

LabVIEW User Manual



LabVIEW User Manual

Contents

- 1. LabVIEW User Manual 46
- 2. LabVIEW Overview 46
- 3. LabVIEW New Features and Changes 47
 - 3.1 LabVIEW 2025 Q3 47
 - 3.2 LabVIEW 2025 Q1 51
 - 3.3 LabVIEW 2024 Q3 53
 - 3.4 LabVIEW 2024 Q1 54
 - 3.5 LabVIEW 2023 Q3 55
 - 3.6 LabVIEW 2023 Q1 58
 - 3.6.1 New Properties, Methods, and Events 60
 - 3.7 LabVIEW 2022 Q3 60
 - 3.7.1 New and Changed Properties, Methods, and Events 61
 - 3.7.1.1 New Properties, Methods, and Events 61
 - 3.7.1.2 Changed Properties and Methods 62
- 4. Updates and Changes for LabVIEW Extended Support Versions 62
 - 4.1 LabVIEW 2021 62
 - 4.2 LabVIEW 2020 64
 - 4.2.1 Interfaces 64
 - 4.2.2 Web Services 66
 - 4.2.3 Application Builder 66
 - 4.2.4 Environment 67
 - 4.2.5 Import Shared Library Wizard 69
 - 4.2.6 Block Diagram 70
 - 4.2.7 VIs and Functions 72
 - 4.2.7.1 New 72
 - 4.2.7.2 Changed 74
 - 4.2.8 Classes, Properties, Methods, and Events 74
 - 4.3 LabVIEW 2019 76
 - 4.3.1 Sets and Maps 76
 - 4.3.2 Shortcut Menu Items 78
 - 4.3.3 Front Panel 78
 - 4.3.4 Application Builder 79
 - 4.3.5 Project Items 80
 - 4.3.6 Execution Highlighting 81

4.3.7	History Probes	81
4.3.8	Handling Errors in Case Structures	82
4.3.9	Text Overflow	82
4.3.10	Environment	83
4.3.11	VIs and Functions	84
4.3.11.1	New	84
4.3.11.2	Changed	86
4.3.12	New Properties	87
5.	Upgrading LabVIEW	87
5.1	1. Back Up	87
5.2	2. Gather Test Data	88
5.3	3. Install	89
5.4	4. Convert VIs	89
5.5	5. Validate Behavior	90
5.6	Compatibility Considerations	91
5.6.1	LabVIEW 2025 Q1	91
5.6.2	LabVIEW 2025 Q1	92
5.6.3	LabVIEW 2024 Q1	92
5.6.4	LabVIEW 2023 Q3	92
5.6.5	LabVIEW 2023 Q1	93
5.6.6	LabVIEW 2022 Q3	93
5.6.7	LabVIEW 2021	94
5.6.8	LabVIEW 2020	94
6.	Finding Your Way in LabVIEW	95
6.1	LabVIEW Environment Overview	98
6.2	Navigating the Controls and Functions Palettes	98
6.3	Menus and Toolbars	99
6.4	Navigation Window	100
6.5	Keyboard Shortcuts	101
6.6	Quick Drop Keyboard Shortcuts	108
6.7	Securing the LabVIEW Environment	109
6.8	Customizing Your Work Environment	110
6.9	Text Characteristics	113
6.10	Nigel, AI Advisor	115
7.	Examples, VI Templates, Project Templates, and Sample Projects	118
8.	Installing LabVIEW	120
8.1	Installing LabVIEW on Windows	120
8.2	Installing LabVIEW on Linux	121
8.2.1	LabVIEW Linux Containers for CI/CD Workflows	121

9. LabVIEW Concepts	122
9.1 LabVIEW Functionalities	122
9.1.1 Boolean Controls and Indicators	122
9.1.2 Container Controls and Indicators	124
9.1.3 Dragging and Dropping in LabVIEW	125
9.1.4 Building the Block Diagram	126
9.1.5 Block Diagram Objects	126
9.1.6 Numeric Data	133
9.1.7 Using Polymorphic Units	141
9.1.8 Compiled Object Cache	146
9.1.9 Types of Distributions	147
9.2 Structure of LabVIEW File System	147
9.2.1 Organization of LabVIEW	147
9.2.2 Grouping Data Using Strings, Clusters, Arrays, and Collections	150
9.3 LabVIEW Subpalettes	151
9.3.1 Graph Subpalette	151
9.3.2 Controls Subpalette	152
9.3.3 Datatypes Subpalette	152
9.3.4 3D Graph Subpalette	153
9.4 LabVIEW Controls and Indicators	153
9.4.1 I/O Name Controls and Indicators	153
9.4.2 List, Table, and Tree Controls and Indicators	157
9.4.3 .NET and ActiveX Controls and Indicators	158
9.4.4 Numeric Controls and Indicators	159
9.4.5 Refnum Controls and Indicators	160
9.4.6 Ring and Enumerated Type Controls and Indicators	161
9.4.7 String and Path Controls and Indicators	162
9.4.8 Variant and Class Controls and Indicators	164
9.4.9 Creating Custom Controls, Indicators, and Type Definitions	165
9.4.10 XControls in LabVIEW	174
9.4.10.1 Using XControls	174
9.4.10.2 Using XControl Abilities	174
9.4.10.3 Using the XControl Facade Ability	175
9.4.10.4 Using XControls in VIs	177
9.5 Functions in LabVIEW	177
9.5.1 Functions Overview	177
9.5.2 Polymorphic Functions	179
9.5.3 Using Comparison Functions	184
9.6 VIs in LabVIEW	188

9.6.1	Express VIs	188
9.6.1.1	Using the Dynamic Data Type	189
9.6.1.2	Creating SubVIs from Express VIs	191
9.6.1.3	Creating Express VIs	192
9.6.1.4	Creating Documentation for Express VIs	193
9.6.1.5	Locating Express VI Components	194
9.6.1.6	Designing Configuration Dialog Box VIs	195
9.6.1.7	Designing Source VIs	197
9.6.1.8	Selecting Express VI Parameter Options	198
9.6.2	Handling Variant Data	199
9.6.3	Creating VIs and SubVIs	201
9.6.4	Using Built-In VIs and Functions	201
9.6.5	SubVIs in LabVIEW	201
9.6.5.1	Creating SubVIs from Sections of a VI	201
9.6.5.2	Displaying SubVIs and Express VIs as Icons or Expandable Nodes	202
9.6.5.3	Creating Modular Code	203
9.6.5.4	Watching for Common Operations	203
9.6.5.5	Building the Connector Pane	204
9.6.5.6	Designing SubVI Front Panels	206
9.6.6	Viewing the Hierarchy of VIs	207
9.6.7	Recursive VIs	208
9.6.8	Polymorphic VIs	208
9.6.9	Malleable VIs	211
9.6.9.1	Using a Malleable VI for Different Data Types	212
9.6.9.2	Using a Malleable VI for Different Classes	212
9.6.10	Choosing between Malleable VIs and Polymorphic VIs	213
9.6.11	VI Icons	214
9.6.11.1	Creating Icons	214
9.6.11.2	Using Icons	216
9.6.11.3	Saving Icons	217
9.6.11.4	Using Glyphs	218
9.6.11.5	Using Icon Templates	219
9.6.11.6	Using Layers	219
9.6.12	Saving VIs	220
9.6.13	Customizing VIs	223
9.6.14	Running and Debugging VIs	225
9.6.15	Running VIs	226
9.6.16	Setting up the VI Revision History	227
9.6.17	Preventing Undefined Data	228

9.6.18	Avoiding Incorrect Dependencies	228
9.6.19	Executing a VI before or after a Build	229
9.6.20	Connector Pane Changes That Reset the Prototype of a VI (Shared Libraries)	229
9.6.21	Picture Plots VIs	230
9.6.22	Picture Functions VIs	231
9.6.23	Graphics Formats VIs	233
9.6.24	Sound VIs	233
9.6.25	Graphics and Sound VIs	238
9.6.26	Documenting and Printing VIs	238
9.6.27	3D Picture Control VIs	239
9.6.28	Documenting VIs	239
9.6.29	Printing VIs	240
9.6.30	Programmatically Controlling VIs	245
9.6.31	Dynamically Loading and Calling VIs	245
9.6.32	Asynchronously Calling VIs	246
9.6.33	Callback VIs	249
9.6.34	Emailing Data from VIs	250
9.6.35	Anatomy of a VI and Common VI Objects	251
9.6.36	VI Scripting	258
9.6.37	Capabilities of the VI Server	259
9.6.38	Building VI Server Applications	260
9.6.39	Manipulating Application and VI Settings	261
9.6.40	Editing and Running VIs on Remote Computers	264
9.6.41	VI Scripting Information in the Context Help Window	264
9.6.42	VI Execution Speed	265
9.6.43	VI Memory Usage	273
9.6.44	VI Code Complexity	308
9.6.45	Reentrancy: Allowing Simultaneous Calls to the Same SubVI	309
9.6.46	Using Matrices	313
9.6.47	Porting and Localizing VIs	317
9.6.48	Porting VIs among Platforms	318
9.6.49	Localizing VIs	322
9.6.50	Debugging Techniques	329
9.6.50.1	Correcting Broken VIs	337
9.6.50.2	Handling Errors	338
9.6.50.3	Using the Standard Functionality for error in Parameters	341
9.6.50.4	Using the Standard Functionality for error out Parameters	342
9.6.51	Predefined Command Line Operations	342
9.6.52	Windows Embedded Standard Targets	349

9.6.52.1	Working with Windows Embedded Standard Targets	349
9.6.52.2	Running VIs on Windows Embedded Standard Targets	351
9.6.52.3	Deploying Installers to Windows Embedded Standard Targets	352
9.7	Block Diagram within LabVIEW	353
9.7.1	Designing the Block Diagram	353
9.7.2	Using Wires to Link Block Diagram Objects	354
9.7.3	Theory of Operation	358
9.7.4	Coercion Dots	360
9.8	Creating a Project in LabVIEW	365
9.8.1	Managing a Project in LabVIEW	365
9.8.2	Working with Projects and Targets	368
9.8.3	Using Folders in LabVIEW Projects	369
9.8.4	Using Libraries in LabVIEW Projects	370
9.8.5	Using Packed Project Libraries in LabVIEW Projects	374
9.8.6	Managing Dependencies in LabVIEW Projects	376
9.8.7	Replacing Project Items	378
9.8.8	Resolving Project Conflicts	378
9.8.9	Adding Targets to a LabVIEW Project (Windows)	381
9.8.10	Working with Application Instances	382
9.8.11	Planning and Designing a Project	383
9.9	Source Control in LabVIEW	384
9.9.1	Using Source Control in LabVIEW	384
9.9.2	Selecting a Source Control Provider	385
9.9.3	Source Control Operations in LabVIEW	386
9.9.4	Facilitating Source Control by Separating Compiled Code from VIs and Other File Types	388
9.10	Build Specifications	389
9.10.1	Building and Distributing Applications	389
9.10.2	Using Build Specifications	390
9.10.3	Creating Build Specifications	392
9.10.4	Preparing Files for Build Specifications	393
9.10.5	Configuring Build Specifications	393
9.10.6	Referencing Source Files in Stand-Alone Applications and Shared Libraries	394
9.10.7	Managing Build Specifications	395
9.11	Applications in LabVIEW	395
9.11.1	Managing Applications	395
9.11.2	Features of Built Applications	396
9.11.3	Installing Stand-Alone Applications and Shared Libraries	396
9.11.4	Using the LabVIEW Run-Time Engine	397

9.11.5	Using ActiveX with LabVIEW	398
9.11.6	Deploying LabVIEW Applications on Windows Vista and Later Operating Systems	406
9.11.7	Deploying Builds on Windows Vista and Later Operating Systems	408
9.11.8	Deploying Installers to Windows Embedded Standard Targets	409
9.11.9	Windows Connectivity	410
9.12	Loops and Structures in LabVIEW	411
9.12.1	Loops and Other Structures	411
9.12.2	Controlling Repetitive Operations Using Loops	413
9.12.3	For Loops: Repeating Operations a Set Number of Times	413
9.12.4	While Loops: Repeating Operations Until a Condition Occurs	414
9.12.5	Transferring Values between Loop Iterations	415
9.12.6	Error Registers: Handling Errors on Parallel Iterations	416
9.12.7	Shift Registers: Passing Values between Loop Iterations	416
9.12.8	Feedback Nodes: Storing Data between Loop Iterations	417
9.12.9	Executing One or More Sections of Code Using Structures	419
9.12.10	Case Structures: Executing a Section of Code Based on Input Values	420
9.12.11	Sequence Structures: Executing Sections of Code Sequentially	421
9.12.12	Disable Structures: Preventing Code from Executing	422
9.12.13	In Place Element Structures: Increasing Memory Efficiency	425
9.12.14	Controlling Timing in LabVIEW Applications (Real-Time, Windows)	426
9.12.15	Characteristics of Timed Structures (Real-Time, Windows)	427
9.12.16	Understanding Absolute and Relative Time in Timed Structures	429
9.12.17	Configuring Timed Loops and Timed Loops with Frames (Real-Time, Windows)	431
9.12.18	Configuring Timed Sequence Structures (Real-Time, Windows)	435
9.12.19	Configuring Settings of a Timed Structure (Real-Time, Windows)	436
9.12.20	Setting the Priority of a Timed Structure (Real-Time, Windows)	442
9.12.21	Selecting a Timing Source for a Timed Structure (Real-Time, Windows)	444
9.12.22	Setting the Input Options of a Timed Structure Dynamically (Real-Time, Windows)	446
9.12.23	Naming Timed Structures (Real-Time, Windows)	447
9.12.24	Synchronizing the Start of Timed Structures (Real-Time, Windows)	448
9.12.25	Aborting the Execution of a Timed Structure (Real-Time, Windows)	450
9.13	Events in LabVIEW	451
9.13.1	Types of Events	451
9.13.2	Event-Driven Programming	452
9.13.3	Determining Which Type of User Interface Events to Use	453
9.13.4	Choosing How the Event Structure Monitors For Events	454
9.14	Grouping Data in LabVIEW	455

9.14.1	Grouping Data Using Strings, Clusters, Arrays, and Collections	455
9.14.2	Grouping Data with Strings	456
9.14.2.1	Tables	457
9.14.3	Grouping Data with Arrays and Clusters	460
9.14.3.1	Arrays	460
9.14.3.2	Clusters	465
9.14.4	Grouping Data with Collections	467
9.14.4.1	Collection Sets	467
9.14.4.2	Collection Maps	468
9.14.5	Format Specifier Syntax	469
9.15	Graphs and Charts in LabVIEW	476
9.15.1	Graphs and Charts	476
9.15.2	Types of Graphs and Charts	477
9.15.2.1	XY Graphs	478
9.15.2.2	Waveform Charts	479
9.15.2.3	Waveform Graphs	480
9.15.2.4	Intensity Charts	482
9.15.2.5	Intensity Graphs	484
9.15.2.6	Digital Waveform Graphs	485
9.15.2.7	Mixed Signal Graphs	488
9.15.3	Customizing Graphs and Charts	493
9.15.4	Using Anti-Aliased Line Plots	509
9.15.5	Using Scale Legends	509
9.15.6	Dynamically Formatting Graphs	510
9.15.7	Using Smooth Updates	510
9.15.8	Using the 2D Picture Control	510
9.15.9	Using the 3D Picture Control	511
9.15.10	Using Meshes to Draw 3D Objects	516
9.16	File I/O Operations in LabVIEW	522
9.16.1	File I/O	522
9.16.2	Basics of File I/O	522
9.16.3	Extending Data File Support using DataPlugins (Windows)	523
9.16.4	Saving Memory using Disk Streaming	524
9.16.5	Improving Efficiency by Disabling File Buffering for Binary File I/O Functions	525
9.16.6	Determining Which File Format to Use	526
9.16.7	File Paths	527
9.16.8	Binary Files	528
9.16.8.1	Creating Binary Files	529
9.16.9	Configuration Files	529

9.16.10	Datalog Files	530
9.16.10.1	Creating Datalog Files	531
9.16.11	Spreadsheet Files	532
9.16.11.1	Creating Spreadsheet Files	533
9.16.12	Text Files	535
9.16.12.1	Creating Text Files	536
9.16.13	XML Files	537
9.16.13.1	Converting Data to and from XML	539
9.16.13.2	Parsing XML Files in LabVIEW	540
9.16.14	Zip Files	543
9.16.15	Windows Configuration Settings File Format	543
9.16.16	Recording Data for Reuse	547
9.16.17	TDM Data Model	551
9.16.17.1	TDM versus TDMS	553
9.16.17.2	Interacting with TDM Files (Windows)	554
9.16.17.3	Interacting with TDMS Files	555
9.16.17.4	Standard versus Advanced TDMS Functions	556
9.16.17.5	File Format Versions	557
9.16.17.6	Synchronous versus Asynchronous File Operations	557
9.16.17.7	File Buffering with TDMS Files	558
9.16.17.8	Reading TDMS Data Asynchronously	559
9.16.17.9	Writing TDMS Data Asynchronously	560
9.16.18	Waveforms	560
9.16.18.1	Reading Waveforms from Files	560
9.16.18.2	Writing Waveforms to Files	561
9.17	Classes and Objects in LabVIEW	562
9.17.1	LabVIEW Object-Oriented Programming	562
9.17.2	Classes and Objects	563
9.17.3	Creating LabVIEW Classes	565
9.17.4	Developing LabVIEW Classes	578
9.17.5	Comparing LabVIEW Class Objects	584
9.17.6	LabVIEW Interfaces	585
9.17.7	Using LabVIEW Classes and Interfaces in an Application	587
9.18	LabVIEW Front Panel	589
9.18.1	Front Panel Controls and Indicators	589
9.18.2	Building the Front Panel	590
9.18.3	Configuring Front Panel Objects	591
9.18.3.1	Configuring the Front Panel	593
9.18.4	Designing User Interfaces	598

9.18.5	Controlling Front Panel Objects	602
9.18.6	Troubleshooting Front Panel Data Binding	603
9.19	Variables in LabVIEW	604
9.19.1	Local Variables, Global Variables, and the Feedback Node	604
9.19.2	Local Variables	605
9.19.3	Global Variables	606
9.19.4	Read and Write Variables	607
9.19.5	Using Local and Global Variables Carefully	607
9.19.6	Understanding Shared Variable Technology	610
9.19.7	Publishing Latest Values with Shared Variables	612
9.19.7.1	Configuring Shared Variables	613
9.19.7.2	Configuring Scaling for Shared Variables	614
9.19.7.3	Creating Shared Variables	618
9.19.7.4	Choosing a Method of Reading and Writing Shared Variables	619
9.19.7.5	Reading and Writing Shared Variables Programmatically	624
9.19.7.6	Reading and Writing Shared Variables with Shared Variable Nodes	629
9.19.7.7	Reading and Writing Shared Variables with the Variable Web Service	632
9.19.7.8	Reading and Writing Shared Variables with Controls and Indicators	633
9.19.7.9	Binding Shared Variables and Other NI-PSP Data Items to Controls and Indicators	633
9.19.7.10	Variable Identifier URLs	634
9.19.7.11	Variable Identifier URL Escape Codes	638
9.19.7.12	Using Variable Methods	639
9.19.7.13	Writing Shared Variables with Existing Data Sources	640
9.19.8	Sharing Live Data Programmatically	641
9.19.9	DataSocket Functions and Variant Data	643
9.19.10	Reading and Writing Live Data Using DataSocket Functions	644
9.19.11	Buffering Data	646
9.19.12	Sending Shared Variable Data over the Network Immediately	648
9.19.13	Making Shared Variables Available on a Network	649
9.20	Data Transfer in LabVIEW	651
9.20.1	Transferring Data	651
9.20.2	Data Communication Methods in LabVIEW	651
9.20.3	Transferring Data over a Network	657
9.20.4	LabVIEW as a Network Client and Server	660
9.20.5	Streaming Data and Sending Commands between Applications	660
9.20.6	Designing Applications Using Network Streams	663
9.20.7	Streaming Data between Applications	663

9.20.8	Sending Commands between Applications	665
9.20.9	Displaying Information about Endpoints	668
9.20.10	Network Stream Endpoints	669
9.20.11	Specifying Network Stream Endpoint URLs	670
9.20.12	Organizing Network Stream Endpoint Names	673
9.20.13	Connecting Network Stream Endpoints Together	674
9.20.14	Allocating Memory for Network Stream Endpoint Buffers	676
9.20.15	Replicating Channels	677
9.20.16	Type Propagation	677
9.20.17	Type Descriptors	678
9.20.17.1	Transferring Data between Two Loops Running in Parallel	689
9.21	Formulas and Equations in LabVIEW	690
9.21.1	Formulas and Equations	690
9.21.1.1	Formula Nodes	691
9.21.1.2	Expression Nodes	692
9.21.1.3	Feedback Node	693
9.22	Communication With LabVIEW Applications	697
9.22.1	Communicating with a LabVIEW Application from Web Clients (Real-Time, Windows)	697
9.22.2	Overview: Web-based Communication with a LabVIEW Application (Real-Time, Windows)	700
9.22.3	Developing Web Services (Real-Time, Windows)	703
9.22.3.1	Components of a Web Service (Real-Time, Windows)	704
9.22.3.2	Developing HTTP Method VIs (Real-Time, Windows)	707
9.22.3.3	Processing POST Data, Form Data, and Uploaded Files (Real-Time, Windows)	709
9.22.3.4	Accessing HTTP Request Information (Real-Time, Windows)	710
9.22.3.5	Configuring HTTP Headers, Streaming, and Buffering (Real-Time, Windows)	711
9.22.3.6	Developing Startup VIs to Initialize and Communicate with HTTP Method VIs (Real-Time, Windows)	712
9.22.3.7	Understanding the Web Services Architecture and File Locations (Real-Time, Windows)	714
9.22.3.8	Hosting Web Services (Real-Time, Windows)	715
9.22.3.9	Publishing Web Services (Real-Time, Windows)	718
9.22.3.10	Transferring Data between Web Services and Clients (Real-Time, Windows)	720
9.22.3.11	Sending Data to a Deployed Application Using URLs (Real-Time, Windows)	721
9.22.3.12	Understanding URL Mappings and Query Strings in Web Services (Real-Time, Windows)	722
9.22.3.13	Using the POST HTTP Method (Real-Time, Windows)	724

9.22.3.14	Uploading Files Using Web Services (Real-Time, Windows) . . .	725
9.22.3.15	Importing Web Services	726
9.22.4	Application Web Server (Real-Time, Windows)	727
9.22.5	Storing Values During HTTP Sessions when Using the Application Web Server (Real-Time, Windows)	727
9.22.6	Viewing and Controlling Front Panels Remotely with the Web Server .	728
9.22.7	Securing the Web Server with TLS/SSL (Windows)	731
9.22.8	Using LabVIEW with TCP, UDP, and Other Low-Level Communications Applications	733
9.22.9	Using LabVIEW with TCP/IP and UDP	734
9.22.10	OPC Systems With LabVIEW	738
9.22.10.1	Connecting to OPC Systems Using LabVIEW (Windows Only) .	738
9.22.10.2	Connecting to the LabVIEW OPC Server from Third- Party OPC Clients (Windows Only)	741
9.22.10.3	Developing OPC Clients in LabVIEW (Windows Only)	741
9.22.11	Communicating Data between Parallel Sections of Code Using Channel Wires	742
9.22.11.1	Using LabVIEW with Wireless Devices	746
9.22.12	Connecting Dynamically with Call By Reference Nodes	748
9.22.13	Creating Cycles among Nodes	749
9.22.14	NI Service Locator	750
9.22.15	Multitasking, Multithreading, and Multiprocessing	750
9.22.15.1	LabVIEW Threading Model	751
9.22.15.2	Multitasking in LabVIEW	752
9.22.15.3	Benefits of Multithreaded Applications	755
9.22.15.4	Manually Assigning CPUs	756
9.22.15.5	Pipelining for Systems with Multiple CPUs	758
9.22.15.6	Prioritizing Parallel Tasks	760
9.22.15.7	Suggestions for Using Execution Systems and Priorities	763
9.22.15.8	Multiprocessing and Hyperthreading in LabVIEW	767
9.23	Memory Management in LabVIEW	769
9.23.1	Managing Performance and Memory	769
9.23.2	Using the Profile Performance and Memory Window	770
9.23.3	Effects of Code Complexity on Compile Time	772
9.23.4	Identifying When LabVIEW Compiles a VI	773
9.23.5	Memory Management for Large Data Sets	774
9.23.6	Storing Data and Reducing Data Copies with Data Value References .	777
9.23.7	How LabVIEW Stores Data in Memory	778
9.23.8	Flattened Data	786
9.24	.NET in LabVIEW	789
9.24.1	Using .NET with LabVIEW	789

9.24.2	.NET Framework Environment	790
9.24.3	Requirements for Using .NET Framework Assemblies in LabVIEW	791
9.24.4	Creating .NET Objects	792
9.24.5	.NET Properties and Methods	793
9.24.6	.NET Events	793
9.24.7	.NET Data Types	794
9.24.8	Configuring a .NET Framework Client Application	795
9.24.9	Using .NET Assemblies with LabVIEW	796
9.24.10	Loading .NET Framework Assemblies	797
9.24.11	Distributing .NET Assemblies with .NET Client Applications	798
9.25	Touch Panel in LabVIEW	800
9.25.1	Creating a Project Using the Touch Panel Project Template	800
9.25.2	Navigation Controls	800
9.25.3	Protecting Data Using Write Filters	801
9.25.4	Supported Touch Panel Targets	802
9.25.5	Using Touch Panel VI Templates	802
10.	LabVIEW How To	803
10.1	Setting up LabVIEW	803
10.1.1	About LabVIEW Evaluation and Activation (Windows)	803
10.1.2	Configuring Your LabVIEW User Name	804
10.1.3	Logging In and Logging Out of LabVIEW (Windows)	805
10.2	Creating your project	805
10.2.1	Creating a LabVIEW Project	805
10.2.2	Selecting a Control	806
10.2.3	Adding Folders to a Project	806
10.2.4	Adding Hyperlinks to a Project	808
10.2.5	Adding Items to a Project	809
10.2.6	Adding LLBs to a Project	810
10.2.7	Adding Targets to a Project (Windows)	811
10.2.8	Finding Items in a Project	812
10.2.9	Sorting Items in a Project	813
10.2.10	Removing Items from a Project	813
10.2.11	Removing Dependencies from a Project	814
10.2.12	Saving a Project	814
10.2.13	Selecting an Application Instance	815
10.2.14	Setting the Default Palette File for a Project Library	815
10.2.15	Showing and Hiding Items in a Project	816
10.2.16	Viewing Files in a Project	816
10.2.17	Working with Project Libraries	817

10.2.17.	Creating a Project Library	817
10.2.17.	Creating a Project Library from a Project Folder	818
10.2.17.	Configuring Access Options in Project Libraries	818
10.2.17.	Protecting Project Libraries	820
10.2.17.	Converting an LLB to a Project Library	821
10.2.17.	Converting VIs in a LabVIEW Project to a Packed Project Library	822
10.2.17.	Editing a VI That Calls a VI in a Packed Project Library	822
10.2.17.	caveats and Recommendations for Packed Project Libraries	823
10.2.18	Additional Project Options	823
10.2.18.	Accessing Recently Opened Files	823
10.2.18.	Disabling the Getting Started Window	824
10.2.18.	Recovering Backup Files Manually	824
10.2.18.	LabVIEW File Extensions	824
10.2.18.	Renaming Files and Project Items	827
10.3	Customizing LabVIEW	828
10.3.1	Setting Font Styles	828
10.3.2	Selecting Custom Colors	828
10.3.3	Configuring User-Defined Colors	829
10.3.4	Setting the Color in a Color Box	830
10.3.5	Setting LabVIEW Options	831
10.3.6	Adding a Directory to VI Search Path	831
10.3.7	Restoring Default Menu Shortcuts	832
10.3.8	Setting Menu Shortcuts	832
10.3.9	Setting Path Options	833
10.3.10	Changing Passwords (Windows)	833
10.4	Customizing your Palette	834
10.4.1	Adding Palette Categories	834
10.4.2	Adding Palette Items to the Favorites Category	834
10.4.3	Adding VIs and Controls to the User and Instrument Drivers Subpalettes	835
10.4.4	Adding VIs or Objects from the Functions or Controls Palette	835
10.4.5	Adding VIs or Objects to the Functions or Controls Palette	836
10.4.6	Automatically Updating Palette Sets	837
10.4.7	Changing Palette Formats	838
10.4.8	Subpalettes	839
10.4.8.1	Creating .NET Subpalettes	839
10.4.8.2	Creating ActiveX Subpalettes	840
10.4.8.3	Creating and Editing Subpalette Descriptions	840
10.4.8.4	Creating Subpalettes in a Palette Set	841

10.4.9	Deleting Empty Slots on a Palette	841
10.4.10	Editing a Palette Set	841
10.4.11	Editing Icon Short Names in a Palette Set	842
10.4.12	Inserting Rows or Columns in a Palette Set	843
10.4.13	Linking Subpalettes to HTML Files or Compiled Help Files	843
10.4.14	Moving Subpalettes in a Palette Set	844
10.4.15	Organizing Palette Categories	844
10.4.16	Sharing Palette Sets	845
10.4.17	Showing and Hiding Palette Categories	845
10.4.18	Configuring Text Comparison for File Recovery	845
10.5	Source Controls with LabVIEW	846
10.5.1	Configuring LabVIEW to Use Source Control	846
10.5.2	Configuring Source Control for LabVIEW Projects	847
10.5.3	Checking out Files from Source Control	848
10.5.4	Undoing File Check-out from Source Control	849
10.5.5	Checking in Files to Source Control	849
10.5.6	Getting the Latest Version of Files in Source Control	850
10.5.7	Comparing Files in Source Control	851
10.5.8	Viewing Revision History in Source Control	852
10.5.9	Adding Files to Source Control	852
10.5.10	Removing Files from Source Control	853
10.5.11	Renaming Files in Source Control	853
10.5.12	Building Stand-Alone Applications with Source Control VIs	854
10.6	LabVIEW Classes	855
10.6.1	Creating a LabVIEW Class or Interface	855
10.6.2	Converting the Contents of a Control to a LabVIEW Class	856
10.6.3	Defining the Private Data of a LabVIEW Class	856
10.6.4	Creating a Member VI in a LabVIEW Class or Interface	857
10.6.5	Viewing the Inheritance Hierarchy of a LabVIEW Class or Interface	863
10.6.6	Changing the Inheritance of a LabVIEW Class or Interface	864
10.6.7	Adding Items to a LabVIEW Class or Interface	865
10.6.8	Changing the Wire Appearance of a LabVIEW Class or Interface	865
10.6.9	Creating Dynamic Dispatch Terminals	866
10.6.10	Viewing and Opening Implementations of Dynamic Dispatch Member VIs	866
10.6.11	Setting the Access Scope of Member VIs	868
10.6.12	Locking and Unlocking LabVIEW Classes or Interfaces	869
10.6.13	Setting the Default Probe for a LabVIEW Class or Interface	870
10.7	Block Diagrams	871
10.7.1	Reading from Binary Files	871

10.7.2	Writing to Binary Files	871
10.7.3	Using Configuration File VIs	872
10.7.4	Data Logging	873
10.7.4.1	Reading from Datalog Files	873
10.7.4.2	Writing to Datalog Files	873
10.7.4.3	Logging Front Panel Data	874
10.7.4.4	Changing the Log-File Binding	874
10.7.4.5	Clearing the Log-File Binding	874
10.7.4.6	Datalog Type	875
10.7.4.7	Deleting a Datalog Record	875
10.7.4.8	Enabling Data Logging	875
10.7.4.9	Retrieving Logged Front Panel Data Using a SubVI	876
10.7.4.10	Selecting the Default Data Directory	877
10.7.4.11	Viewing Logged Data Interactively	878
10.7.5	LabVIEW Measurement Files	878
10.7.6	Reading Data from LabVIEW Measurement Files	879
10.7.7	Writing Data to LabVIEW Measurement Files	879
10.7.8	Reading from Spreadsheet Files	880
10.7.9	Writing to Spreadsheet Files	880
10.7.10	Streaming Data to Spreadsheet Files	881
10.7.11	TDM/TDMS Files	883
10.7.12	Streaming External Data to a TDMS File (Windows)	883
10.7.13	Reading from Text Files	884
10.7.14	Writing to Text Files	885
10.7.15	Converting LabVIEW Data to XML	885
10.7.16	Converting XML to LabVIEW Data	885
10.7.17	Automatically Cleaning Up the Block Diagram	886
10.7.18	Creating Controls and Indicators on the Block Diagram	887
10.7.19	Creating and Editing User-Defined Constants	888
10.7.20	Displaying Terminals as Icons or Data Types	889
10.7.21	Removing and Rewiring Block Diagram Objects	889
10.7.22	Inserting Block Diagram Objects	890
10.7.23	Replacing Block Diagram Objects	890
10.7.24	Resizing Nodes	891
10.7.25	Reusing Snippets of Code	891
10.7.26	Selecting the Instance of a Polymorphic VI Manually	892
10.7.27	Using Wires	893
10.7.27.1	Automatically Wiring Objects	893
10.7.27.2	Configuring Automatic Wiring	894

10.7.27.3	Deleting Wires	894
10.7.27.4	Displaying Terminals for VIs, Functions, and Express VIs	895
10.7.27.5	Moving Wires	895
10.7.27.6	Routing Wires	896
10.7.27.7	Selecting Wires	897
10.7.27.8	Wiring Block Diagram Objects Manually	897
10.7.27.9	Wiring Techniques to Avoid	898
10.7.27.10	Wiring Structures	899
10.7.27.11	Assigning More Than One Value to a Sequence Local Terminal .	900
10.7.27.12	Failing to Wire a Tunnel in All Cases of a Structure	900
10.7.27.13	Overlapping Tunnels	901
10.7.27.14	Wiring from Multiple Frames of a Stacked Sequence Structure .	901
10.7.27.15	Wiring Tunnels Automatically in Structures with Multiple Cases	902
10.8	Searching Options in LabVIEW	903
10.8.1	Finding a Block Diagram Terminal from a Front Panel Object	903
10.8.2	Finding a Front Panel Object from a Block Diagram Terminal	903
10.8.3	Finding All Instances of a VI or Type Definition	903
10.8.4	Finding an Object on the Palettes	904
10.8.5	Finding an Object or Terminal Local Variable	904
10.8.6	Finding Property Nodes Linked to an Object or Terminal	905
10.8.7	Finding VIs in a Directory	905
10.8.8	Finding and Replacing VIs, Objects, and Text	905
10.8.9	Finding and Replacing Text in VIs	906
10.8.10	Finding and Replacing Block Diagram Objects	907
10.9	LabVIEW Misc	908
10.9.1	Selecting a Tool	908
10.9.2	Navigating Large Front Panels and Block Diagrams	909
10.9.3	Changing Controls to Indicators and Indicators to Controls	909
10.9.4	Clearing Front Panel Indicators	910
10.9.5	Shortcut menu plug-ins	910
10.9.5.1	Understanding Shortcut Menu Plug-Ins	910
10.9.5.2	Part 1: Specifying the Objects that Shortcut Menu Plug-Ins Affect	912
10.9.5.3	Part 2: Building the Shortcut Menu Using Plug-Ins	912
10.9.5.4	Part 3: Executing Shortcut Menu Plug-Ins	914
10.9.5.5	Identifying Refnum Types and Menu Item Tags of Objects	915
10.9.5.6	Troubleshooting Shortcut Menu Plug-Ins	915
10.9.5.7	Tips and Tricks for Creating Shortcut Menu Plug-Ins	916
10.9.6	Adding or Reducing Space on the Front Panel or Block Diagram	917

10.9.7 Reading from One Front Panel Object and Writing to Two Block Diagram Locations	918
10.9.8 Writing to One Front Panel Object from Two Block Diagram Locations	918
10.10 VIs	919
10.10.1 Dragging and Dropping VIs, Pictures, and Text	919
10.10.2 Customizing Styles for your Current VI	920
10.10.2.1 Setting the Control and Indicator Style for the Current VI	920
10.10.2.2 Setting the Alignment Grid Size for the Current VI	921
10.10.2.3 Setting Printing Options for the Current VI	921
10.10.2.4 Setting Revision History Options for the Current VI	922
10.10.3 Creating a Recursive VI	922
10.10.4 Searching in the VI Hierarchy	923
10.10.5 Viewing the Hierarchy of VIs	923
10.10.6 Polymorphic VIs	925
10.10.6.1 Building Polymorphic VIs	925
10.10.6.2 Editing the Shortcut Menus of a Polymorphic VI	927
10.10.6.3 Adding or Removing an Instance in a Polymorphic VI	927
10.10.7 Building Malleable VIs	928
10.10.8 Debugging Malleable VIs	929
10.10.9 Saving VIs	929
10.10.10 Creating Password-Protected VIs	930
10.10.11 Locking and Unlocking VIs	931
10.10.12 Common Mass Compile Status Messages	931
10.10.13 LLBs	933
10.10.13.1 Creating LLBs	933
10.10.13.2 Converting Directories to LLBs	933
10.10.13.3 Converting LLBs to Directories	934
10.10.13.4 Merging VIs and LLBs	934
10.10.13.5 Marking VIs as Top-Level VIs in an LLB	935
10.10.14 Mass Compiling VIs	935
10.10.15 Removing Block Diagrams from VIs	937
10.10.16 Removing Files from LLBs	937
10.10.17 Reverting to the Last Saved Version of a VI	938
10.10.18 Saving Changes to VIs That Ship with LabVIEW	938
10.10.19 Saving for a Previous Version	938
10.10.19.1 Defining the Save Version for all VIs	939
10.10.19.2 Using the Save for Previous Version Option	940
10.10.20 Saving Individual VI Files	941
10.10.21 Saving VIs in Existing LLBs	942
10.10.22 Customizing VIs	942

10.10.23	Changing a VI Window Title	942
10.10.24	Copying Browse Paths and Keywords	943
10.10.25	Preparing Example VIs to Appear in the NI Example Finder	943
10.10.26	Setting Execution Properties	946
10.10.27	Setting Window Appearance and Size	946
10.10.28	Running VIs	947
10.10.29	Launching a VI from the Command Line	948
10.10.30	LabVIEW-Defined Command-Line Arguments	949
10.10.31	Adding Comments to the Revision History	949
10.10.32	Resetting the VI Revision History and Revision Number	950
10.10.33	Viewing the VI Revision History	950
10.10.34	Debugging VIs	951
10.10.34.1	Disabling Debugging Tools	951
10.10.34.2	Disabling Sections of Block Diagrams	951
10.10.34.3	Displaying Warnings	952
10.10.34.4	Execution Highlighting	952
10.10.34.5	Single-Stepping through a VI	953
10.10.34.6	Suspending Execution	954
10.10.35	Memory Usage	955
10.10.35.1	Profiling VI Execution Time and Memory Usage	955
10.10.35.2	Identifying Buffer Allocations in LabVIEW Applications	956
10.10.35.3	Optimizing Execution Speed for Built Applications	956
10.10.35.4	Choosing between Editor Responsiveness and VI Execution Speed	958
10.10.35.5	Resolving Out of Memory Errors When Compiling a VI	960
10.10.36	VI Strings	960
10.10.36.1	Exporting and Importing VI Strings	960
10.10.36.2	Example of Exported VI Strings	961
10.10.37	Comparing and Merging VIs	963
10.10.37.1	Comparing VIs and VI Hierarchies in LabVIEW	963
10.10.37.2	Comparing VIs using the LabVIEW Command Line Interface	967
10.10.37.3	Configuring Source Control with LVCompare.exe	968
10.10.37.4	Configuring Source Control with LVMerge.exe	970
10.10.37.5	Merging VIs	971
10.10.37.6	Merging LLBs	972
10.10.38	VI Metrics	974
10.10.38.1	Using the VI Metrics Window	974
10.10.38.2	Analyzing VIs with the VI Metrics Window	974
10.10.38.3	Block Diagram Metrics	975
10.10.38.4	Shared Library Metrics	975

10.10.38	Globals/Locals Metrics	975
10.10.38	SubVI Interface Metrics	976
10.10.38	User Interface Metrics	976
10.10.39	Setting Undo	976
10.10.40	Configuring Shortcuts for the Quick Drop Dialog Box	977
10.10.41	Managing Unfinished Tasks	978
10.10.42	Creating VIs from Templates and Sample Projects	978
10.10.43	Creating Template VIs	979
10.10.44	Separating Compiled Code from VIs and Other File Types	980
10.10.45	Asynchronously Calling VIs	981
10.10.46	Asynchronously Calling a VI and Collecting the Results	982
10.10.47	Asynchronously Calling a VI without Collecting the Results	984
10.10.48	Asynchronously Calling Multiple Instances of a VI for Parallel Execution	985
10.10.49	Combining Option Flags into a Single Option Value	987
10.10.50	Configuring the VI Server	989
10.10.51	Configuring VI Calls	989
10.10.52	Controlling Front Panel Objects Programmatically from a SubVI	990
10.10.53	Creating a Strictly Typed VI Reference to Call VIs Dynamically	991
10.10.54	Creating a VI Server Application	992
10.10.55	Creating Control Reference SubVIs from Property Nodes and Invoke Nodes	993
10.10.56	Dynamically Calling a VI	994
10.10.57	Specifying Which VIs You Can Control Remotely	994
10.10.58	VI Scripting	995
10.10.58	VI Scripting	995
10.10.58	Enabling VI Scripting	995
10.10.58	Common VI Scripting Tasks	996
10.10.58	Obtaining a VI Reference	1002
10.10.58	Obtaining a VI Object Reference Using VI Scripting	1002
10.10.58	Obtaining References to Objects in a Known Target VI Using VI Scripting	1003
10.10.58	Obtaining References to Objects in an Unknown Target VI Using VI Scripting	1005
10.10.58	Adding an Object to a VI Using VI Scripting	1008
10.10.58	Configuring a Connector Pane Using VI Scripting	1011
10.10.58	Creating a New VI from a Template Using VI Scripting	1014
10.10.58	Closing References to VIs and VI Objects Using VI Scripting	1016
10.10.58	Navigating a VI Using VI Scripting	1018
10.10.58	Navigating from a Node to a Wire Using VI Scripting	1020
10.10.58	Navigating from a Wire to a Node Using VI Scripting	1022

10.10.58	Navigating across a Structure Using VI Scripting	1024
10.10.59	VI Scripting Tutorial	1026
10.10.58T17	Tutorial: Creating and Opening a VI Using VI Scripting	1026
10.10.58T18	Tutorial: Adding an Object to a VI Using VI Scripting	1028
10.10.58T19	Tutorial: Repositioning an Object in a VI Using VI Scripting	1029
10.10.58T20	Tutorial: Wiring Objects Using VI Scripting	1031
10.10.59	Adding New Visible VIs Entries	1034
10.10.60	Editing Existing Visible VIs Entries	1035
10.10.61	Removing Visible VIs Entries	1035
10.10.62	Calling Non-Modal Windows Programmatically	1036
10.10.63	Importing a Web Service as a Library of VIs	1036
10.11	Objects	1037
10.11.1	Aligning Objects	1037
10.11.2	Coloring Objects and Backgrounds	1038
10.11.3	Copying and Pasting Objects	1039
10.11.4	Copying Colors from One Object to Another	1040
10.11.5	Creating Transparent Objects	1040
10.11.6	Displaying Hidden Front Panel Objects	1041
10.11.7	Distributing Objects	1041
10.11.8	Grouping and Locking Objects	1042
10.11.9	Reordering Objects	1043
10.11.10	Replacing a Front Panel Object from the Clipboard	1043
10.11.11	Replacing Front Panel Objects	1044
10.11.12	Scaling Front Panel Objects	1044
10.11.13	Setting the Tabbing Order of Front Panel Objects	1045
10.11.14	Changing Values of Numeric Objects with a Scale	1046
10.11.15	Displaying Radixes on Numeric Objects	1046
10.12	Controls and Indicators	1047
10.12.1	Configuring Controls and Indicators	1047
10.12.2	I/O Controls and Indicators	1048
10.12.3	Variant Control and Indicator	1049
10.12.4	Boolean Controls	1049
10.12.5	Configuring I/O Controls	1049
10.12.6	Displaying All Hidden Controls and Indicators	1050
10.12.7	Displaying Scroll Bars of Controls and Indicators	1050
10.12.8	Hiding Front Panel Controls and Indicators	1051
10.12.9	Customizing Controls and Indicators that are Parts of Controls and Indicators	1051
10.12.10	Adding Custom Controls and Indicators to Front Panels and Block Diagrams	1053

10.12.10	Changing Data Ranges of Numeric Controls and Indicators	1054
10.12.10	Changing Fill Options for Slide Controls and Indicators	1055
10.12.10	Changing Format and Precision of Numeric Controls and Indicators	1055
10.12.11	Importing Graphics into Boolean Controls and Indicators	1056
10.12.15	Creating Custom Controls and Indicators from the Front Panel	1058
10.12.15	Editing Saved Custom Controls and Indicators	1058
10.12.17	Caveats and Recommendations when Creating Custom Controls and Indicators	1059
10.12.18	Controls	1060
10.12.18	Setting Keyboard Shortcuts for Controls	1060
10.12.18	Container Controls	1061
10.12.18	Configuring Subpanel Controls	1061
10.12.18	Loading a Front Panel in a Subpanel Control	1062
10.12.18	Tab Controls	1063
10.12.18	Refnum Controls	1067
10.12.18	Ring and Enum Controls	1072
10.12.18	Changing the Value of a Path Control	1076
10.12.18	Configuring Browse Options of Path Controls	1076
10.12.18	Configuring Combo Box Controls to Allow Undefined String Values	1077
10.12.18	Editing Items in Combo Box Controls	1077
10.12.18	XC Controls	1078
10.12.18	Configuring Run-Time Drag-and-Drop Behavior for Text-Based Controls	1085
10.12.18	Implementing Custom Drag-and-Drop Behavior	1087
10.12.18	Step 1: Identify Which Controls to Use as the Drag Source and Drop Target	1089
10.12.18	Step 2: Prepare the VI to Detect When the User Causes Drag-and-Drop Events to Occur	1089
10.12.18	Step 3: Detect When the User Aborts a Drag-and-Drop Operation	1091
10.12.18	Step 4: Start the Drag-and-Drop Operation by Providing the Drag Data	1091
10.12.18	Step 5: Verify that the Drop Target Can Accept the Drag Data	1093
10.12.18	Step 6: Update the Drop Target with the Drag Data	1094
10.12.19	Creating Custom Controls and Indicators Using the New Dialog Box	1096
10.13	Arrays and Clusters	1096
10.13.1	Arrays	1096
10.13.1	Array, Matrix & Cluster Controls and Indicators	1096
10.13.1	Adding Elements to Arrays	1097
10.13.1	Changing Array Default Values	1097
10.13.1	Changing Array Dimensions	1098

10.13.1.5	Clearing an Array	1098
10.13.1.6	Creating Array Controls and Indicators	1099
10.13.1.7	Default Sizes and Values of Arrays	1099
10.13.1.8	Determining the Size of Arrays	1100
10.13.1.9	Displaying an Array in a Table	1100
10.13.1.10	Displaying Elements in Arrays	1101
10.13.1.11	Displaying or Hiding Index Displays	1102
10.13.1.12	Selecting, Cutting, Copying, and Pasting Array Cells	1102
10.13.1.13	Setting Array Default Values	1103
10.13.1.14	Creating Array Constants	1104
10.13.1.15	Default Data in Arrays	1104
10.13.1.16	Deleting Elements, Rows, and Columns within Arrays	1105
10.13.1.17	Inserting Elements, Rows, and Columns into Arrays	1105
10.13.1.18	Programmatically Deleting Elements, Rows, Columns, and Pages within Arrays	1105
10.13.1.19	Programmatically Inserting Elements, Rows, Columns, and Pages into Arrays	1106
10.13.1.20	Replacing Elements, Rows, Columns, and Pages within Arrays	1107
10.13.2	Clusters	1109
10.13.2.1	Clusters	1109
10.13.2.2	Changing Cluster Controls to Indicators and Cluster Indicators to Controls	1109
10.13.2.3	Creating Cluster Controls and Indicators	1109
10.13.2.4	Modifying Cluster Element Order	1110
10.13.2.5	Resizing Clusters	1110
10.13.2.6	Setting Cluster Default Values	1111
10.13.2.7	Bundling Elements into Clusters	1111
10.13.2.8	Creating Cluster Constants	1112
10.13.2.9	Replacing Cluster Elements	1112
10.13.2.10	Unbundling Elements from Clusters	1113
10.13.2.11	Unbundling Elements from Clusters by Name	1113
10.13.3	Converting an Array to a Cluster	1114
10.13.4	Converting a Cluster to an Array	1115
10.13.5	Moving Arrays and Clusters	1115
10.13.6	Tabbing through Elements of an Array or Cluster	1115
10.14	Matrices	1116
10.14.1	Matrices	1116
10.14.2	Creating Matrix Controls, Indicators, and Constants	1116
10.14.3	Converting an Array to a Matrix	1117
10.14.4	Converting a Matrix to an Array	1117

10.15	Formula Nodes, Expression Nodes, and Script Nodes	1118
10.15.1	Formula Node and Expression Node Functions	1118
10.15.2	Precedence of Operators in Formula Nodes and Expression Nodes	1120
10.15.3	Array Manipulation in Formula Nodes	1121
10.15.4	Changing a Formula Node Terminal Type	1122
10.15.5	Creating Formula Nodes	1122
10.15.6	Formula Node Syntax	1123
10.15.7	Removing a Terminal from a Formula Node	1129
10.15.8	Scope Rules for Declaring Variables in Formula Nodes	1129
10.15.9	Creating Expression Nodes	1130
10.15.10	Expression Node Syntax	1131
10.15.11	Script Nodes	1131
10.15.12	Adding Scroll Bars to Script Nodes	1131
10.15.13	Calling the MATLAB® Software from LabVIEW	1131
10.15.14	Changing Script Servers in Script Nodes	1132
10.15.15	Clearing Scripts	1133
10.15.16	Configuring the Data Type of Script Node Terminals	1133
10.15.17	Debugging Scripts	1133
10.15.18	Importing or Exporting Your Scripts	1134
10.15.19	Showing and Hiding Line Numbers in Script Nodes	1134
10.16	Variables	1135
10.16.1	Read and Write Variables	1135
10.16.2	Local Variables	1135
10.16.2.1	Changing the Data Direction of a Local Variable	1135
10.16.2.2	Creating Local Variables	1136
10.16.2.3	Memory Considerations when Using Local Variables	1136
10.16.3	Global Variables	1137
10.16.3.1	Changing the Data Direction of a Global Variable	1137
10.16.3.2	Creating Global Variables	1137
10.16.3.3	Creating Template Global Variables	1138
10.16.3.4	Memory Considerations when Using Global Variables	1138
10.16.4	Shared Variables	1139
10.16.4.1	Configuring Firewalls and Network Address Translating Routers for Shared Variables	1139
10.16.4.2	Scaling Shared Variables	1140
10.16.4.3	Resources Exposed by the Variable Web Service	1142
10.16.4.4	Finding Out Basic Information about a Shared Variable	1143
10.16.4.5	Reading the Value of a Variable	1144
10.16.4.6	Writing to a Variable	1145
10.16.4.7	Writing to Multiple Variables Simultaneously	1145

10.16.4.Creating a Variable Group	1147
10.16.4.Deleting a Variable Group	1149
10.16.4.Reading a Variable Group	1149
10.16.4.Getting Variable Updates	1149
10.16.4.Editing Multiple Shared Variables	1150
10.16.4.Creating Bound Variables	1151
10.16.4.Importing Shared Variable Configuration from a Spreadsheet	1151
10.16.4.Exporting Shared Variable Configuration to a Spreadsheet	1152
10.17 Numeric Data	1152
10.17.1 Numeric Conversion	1152
10.17.2 Changing Numeric Representation	1153
10.17.3 Changing the Numeric Representation of a Control Refnum	1154
10.17.4 Configuring Output Settings for Numeric Functions	1154
10.17.5 Numeric Data Types	1156
10.17.6 Complex Numbers	1156
10.17.7 Floating-Point Numbers	1157
10.17.8 Integers	1157
10.17.9 Numeric Data Types Table	1158
10.17.10 Unsigned Integers	1159
10.17.11 Fixed-Point Data	1160
10.17.11.1 Fixed-Point Numbers	1160
10.17.11.2 Configuring Fixed-Point Numbers	1160
10.17.11.3 Caveats and Recommendations for Using Fixed-Point Numbers	1163
10.17.11.4 Adapting to Cycles of Fixed-Point Data in Loops with Shift Registers	1163
10.17.11.5 Displaying and Hiding the Overflow Status LED for Fixed-Point Numbers	1164
10.17.11.6 Functions and Functions that Support the Fixed-Point Data Type	1165
10.17.12 Variant Data	1168
10.17.12.1 Variant Data	1168
10.17.12.2 Adding Attributes to Variant Data	1169
10.17.12.3 Checking Variants for Known Data Types	1169
10.17.12.4 Converting Flattened Data to Variant Data	1171
10.17.12.5 Converting LabVIEW Data to Variant Data	1171
10.17.12.6 Converting Variant Data to LabVIEW Data	1172
10.17.12.7 Deleting Attributes from Variant Data	1173
10.17.12.8 Retrieving Attributes from Variant Data	1173
10.17.13 Units	1174
10.17.13.1 Associating Units with a Numeric Control or Indicator	1174

10.17.13	Available Units in LabVIEW	1175
10.17.13	Creating Polymorphic Units	1180
10.17.13	Unit Syntax	1180
10.17.13	Cutting, Copying, and Pasting Digital Data	1181
10.17.13	Exporting Data and Images	1182
10.17.13	Inserting and Deleting Rows and Columns in Digital Data Controls	1183
10.18	Comparing functions in LabVIEW	1183
10.18.1	Comparing Arrays	1183
10.18.2	Comparing Boolean Values	1184
10.18.3	Comparing Characters	1184
10.18.4	Comparing Clusters	1184
10.18.5	Comparing Numeric Values	1185
10.18.6	Comparing Strings	1185
10.18.7	Determining Range Values and Coercing Data	1185
10.18.8	Determining Range Values and Coercing Data in Arrays	1186
10.18.9	Setting Comparison Functions to Compare Elements or Aggregates	1187
10.19	Showing and Hiding Labels and Digital Displays	1187
10.20	Importing Background Images for Panes	1187
10.21	Importing Graphics on Linux	1188
10.22	Importing Graphics on macOS	1189
10.23	Importing Graphics on Windows	1190
10.24	Editing the Labels of Buttons	1190
10.25	Graphs and Charts	1192
10.25.1	Getting Started with Graphs and Charts	1192
10.25.2	Adding a Plot to a Plot Legend	1193
10.25.3	Adding a Scroll Bar to a Plot Legend	1193
10.25.4	Adding Markers to X- and Y-Scales on a Graph or Chart	1194
10.25.5	Adding Multiple X- and Y-Scales on Graphs or Charts	1194
10.25.6	Changing the Format and Precision of Graph and Chart Axes	1195
10.25.7	Clearing the Data on Graphs and Charts	1195
10.25.8	Configuring a Graph or Chart	1196
10.25.9	Configuring the X- and Y-Axes on a Graph or Chart	1196
10.25.10	Customizing an Analog Plot Using the Plot Legend	1197
10.25.11	Scrolling through a Graph or Chart	1198
10.25.12	Setting the Number of Plots in a Plot Legend	1198
10.25.13	Zooming and Moving the View of a Graph or Chart	1199
10.25.14	Adding Annotations to Graphs	1200
10.25.15	Adding Cursors to Graphs	1201
10.25.16	Adding Plot Areas to a Mixed Signal Graph	1203

10.25.17	Changing the Plot Legend View in a Digital Waveform Graph	1203
10.25.18	Creating a Graph	1204
10.25.19	Customizing a Plot on a Digital Waveform Graph Using the Plot Legend	1205
10.25.20	Customizing a Plot on a Mixed Signal Graph Using the Plot Legend . .	1206
10.25.21	Deleting Cursors from Graphs	1208
10.25.22	Displaying Digital Lines and Buses in a Digital Waveform Graph	1208
10.25.23	Displaying Special Planes on the XY Graph	1209
10.25.24	Graphing Digital Data	1213
10.25.25	Graphing Multiple Data Types on a Mixed Signal Graph	1213
10.25.26	Graphing Waveform Data	1214
10.25.27	Graphing XY Data	1215
10.25.28	Ignoring Time Stamp and Plot Labels in Graphs	1215
10.25.29	Mapping Graph Coordinates to Plot Image Coordinates	1216
10.25.30	Moving Plots and Buses to Different Plot Areas in a Mixed Signal Graph	1217
10.25.31	Plotting Data on 2D Graphs	1217
10.25.32	Plotting Data on 3D Graphs	1218
10.25.33	Plotting Data on a 3D Surface Graph	1219
10.25.34	Plotting X, Y, and Z Data on an ActiveX 3D Surface Graph	1220
10.25.35	Replaying the Animation for the 3D Comet Graph	1221
10.25.36	Charting Waveform Data	1221
10.25.37	Creating a Chart	1222
10.26	Changing the Numeric Representation of a Waveform	1222
10.27	Acquiring or Simulating a Signal	1223
10.28	Analyzing a Signal	1223
10.29	Displaying a Signal	1223
10.30	Filtering a Signal	1224
10.31	Merging Signals	1224
10.32	Splitting Signals	1225
10.33	Instrument Driver Modification Instructions	1225
10.34	Creating Instrument Driver Projects	1229
10.35	Creating Instrument Driver VIs	1231
10.36	Displaying Pixels Not Visible in a 2D Picture Control	1233
10.37	Resetting 2D Picture Controls	1233
10.38	Using Sound Input and Sound Output Devices	1233
10.39	Creating a 3D Scene with the 3D Picture Control	1234
10.40	Configuring a Camera for a 3D Scene	1235
10.41	Configuring a Light Source for a 3D Scene	1236
10.42	Changing the Surface Appearance of a 3D Object	1238
10.43	Creating Relationships between 3D Objects	1241

10.44 Adding Text to a 3D Scene	1243
10.45 Configuring a 3D Scene Window	1244
10.46 Changing the Mechanical Action of a Boolean Object	1246
10.47 Configuring a Mixed Checkbox	1247
10.48 Creating a Radio Buttons Control	1247
10.49 Creating and Operating Boolean Controls and Indicators	1248
10.50 Building Digital Tables	1248
10.51 Listboxes, Tree Controls, and Tables	1250
10.51.1 Listboxes, Tree Controls, and Tables	1250
10.51.2 Creating a Tree Control	1250
10.51.3 Creating Listboxes	1251
10.51.4 Creating Multicolumn Listboxes	1252
10.51.5 Deleting Rows in Listboxes and Tree Controls	1252
10.51.6 Disabling or Enabling Listbox and Tree Control Items	1253
10.51.7 Displaying Components in Listboxes, Tables, Tree Controls, and Digital Data Controls	1253
10.51.8 Editing Data in Tables	1254
10.51.9 Editing Listbox and Tree Control Cells at Run Time	1254
10.51.10 Entering Data in Tables	1255
10.51.11 Indenting Tree Control Items	1255
10.51.12 Inserting Items in Listbox and Tree Controls	1256
10.51.13 Inserting or Deleting Rows or Columns in Listboxes and Multicolumn Listboxes	1256
10.51.14 Inserting or Deleting Rows or Columns of Data in Tables	1256
10.51.15 Limiting Tree Control Item Selection to Siblings	1257
10.51.16 Opening and Closing Tree Control Items	1257
10.51.17 Resizing Rows and Columns in Listboxes, Tables, and Tree Controls	1258
10.51.18 Selecting Listbox and Tree Control Items at Run Time	1258
10.51.19 Selecting Parent Items in Tree Controls	1259
10.51.20 Selecting Table Cells, Rows, or Columns	1259
10.51.21 Setting Case Sensitivity in Listboxes and Tree Controls	1260
10.51.22 Setting Column Widths at Run Time	1260
10.51.23 Setting Expand/Contract Symbols in Tree Controls	1260
10.51.24 Setting the Number of Items You Can Select in Listboxes and Tree Controls at Run Time	1261
10.51.25 Using Tables	1261
10.51.26 Adding Header Text to Listboxes, Tables, and Tree Controls	1262
10.51.27 Adding Symbols to Listboxes and Tree Controls	1262
10.51.28 Coloring Listboxes, Multicolumn Listboxes, Tree Controls, and Tables	1263
10.51.29 Configuring Cells for Multiple Line Input	1263

10.52 Structures	1264
10.52.1 Adding Structures to the Block Diagram	1264
10.52.2 Resizing Structures	1264
10.52.3 Adding Objects inside Structures	1265
10.52.4 Replacing One Structure with Another Structure	1265
10.52.5 Removing Structures without Deleting Objects in the Structure	1266
10.53 Loops	1266
10.53.1 Automatically Concatenating Arrays Leaving Loops	1266
10.53.2 Processing Individual Elements in an Array or a Collection with a Loop	1266
10.53.3 Conditionally Writing Values to Loop Output Tunnels	1271
10.53.4 Checking Boolean Values on Each Iteration	1271
10.53.5 Changing the Continuation Behavior of Loops	1272
10.53.6 Controlling the Timing on Loops	1273
10.53.7 Handling Errors Using Conditional Terminals on Loops	1273
10.53.8 Specifying the Number of Times Operations in For Loops Execute	1274
10.53.9 Stopping For Loops When a Condition Occurs	1274
10.53.10 Parallel Iterations: Improving For Loop Execution Speed	1275
10.53.11 Enabling Parallel For Loop Iterations	1276
10.53.12 Verifying That a For Loop Can Run with Parallel Iterations	1276
10.53.13 Configuring Parallel For Loop Iterations to Tweak Performance	1277
10.53.14 Performance Considerations for Parallel For Loop Iterations	1279
10.53.15 Specifying Conditions for While Loops	1279
10.53.16 Preventing Code Execution in the First While Loop Iteration	1280
10.53.17 Example: Handling Notifier Operation Functions in While Loops	1281
10.53.18 Passing One Value to the Next Loop Iteration	1282
10.53.19 Passing Multiple Values to the Next Loop Iteration	1282
10.53.20 Resetting the Initial Value of Shift Registers	1283
10.53.21 Replacing Shift Registers with Tunnels	1284
10.53.22 Passing the Last Value from the Previous Execution to the Feedback Node	1286
10.53.23 Resetting the Input Value on a Feedback Node Each Time a Loop Executes	1286
10.53.24 Replacing Shift Registers with a Feedback Node	1287
10.53.25 Changing the Direction of the Feedback Node Arrow	1288
10.54 Subdiagram and Case Structures	1289
10.54.1 Adding and Duplicating Subdiagrams	1289
10.54.2 Moving between Subdiagrams	1290
10.54.3 Deleting Subdiagrams	1291
10.54.4 Rearranging Subdiagrams	1291
10.54.5 Creating a Case Structure to Execute Conditional Code	1292

10.54.6	Entering Values for each Case Structure Subdiagram	1293
10.54.7	Avoiding Wiring Unwired Output Tunnels on Case Structures	1295
10.54.8	Handling Errors Using Case Structures	1295
10.54.9	Adding Cases between Case Structure Subdiagrams	1296
10.54.10	Specifying a Default Subdiagram for Case Structures	1297
10.54.11	Swapping Case Structure Subdiagrams	1297
10.54.12	Shifting Case Structure Subdiagrams	1298
10.54.13	Passing Data through Frames of a Flat Sequence Structure	1299
10.54.14	Passing Data through Frames of a Stacked Sequence Structure	1299
10.54.15	Determining When to Use Sequence Structures	1300
10.54.16	Configuring Conditions for Conditional Disable Structures	1301
10.54.17	Customizing Conditions for Conditional Disable Structures	1305
10.54.18	Reducing Data Copies with In Place Element Structures	1305
10.54.19	Caveats and Recommendations for Using In Place Element Structures	1306
10.54.20	Adding and Deleting Timed Structure Frames (Real-Time, Windows) .	1307
10.54.21	Merging Timed Structure Frames (Real-Time, Windows)	1307
10.54.22	Displaying and Hiding the Terminals of a Timed Structure Node (Real-Time, Windows)	1307
10.55	Events	1308
10.55.1	Adding and Duplicating Event Cases	1308
10.55.2	Configuring Events Handled by the Event Structure	1308
10.55.3	Creating Custom Events	1309
10.55.4	Dynamically Registering for Events	1312
10.55.5	Dynamically Modifying Event Registration	1314
10.55.6	Step 1: Create a Placeholder for the Event You Want to Occur Only Under Certain Circumstances	1314
10.55.7	Step 2: Design an Event Case to Specify When You Want LabVIEW to Register for the Event	1316
10.55.8	Step 3: Unregister for the Event before You Unregister for All Events .	1317
10.55.9	Finding a Control Associated with an Event Case	1319
10.55.10	Locking Front Panels	1319
10.55.11	Viewing Enqueued Events at Run Time	1320
10.55.12	Caveats and Recommendations when Using Events in LabVIEW	1320
10.55.13	Avoid Placing Two Event Structures in One Loop	1322
10.55.14	Ensure That Event Structures Handle Events whenever Events Occur .	1323
10.55.15	Handling a Latched Stop Boolean Control in an Event Structure	1325
10.55.16	One Case Handling Multiple Filter Events	1326
10.55.17	One Case Handling Multiple Notify Events	1327
10.55.18	Prompting Users with Dialog Boxes in a Mouse Down? Event Case . .	1327
10.55.19	Use Events for Direct User Interaction Only	1327

10.55.20	Use Only One Event Structure whenever Possible	1327
10.55.21	Using Events versus Using the Wait for Front Panel Activity Function	1328
10.55.22	Using Events with Latched Boolean Controls	1328
10.55.23	Using Events with Subpanel Controls	1328
10.56	Strings	1329
10.56.1	Strings	1329
10.56.2	Converting Numbers into Strings	1329
10.56.3	Formatting Strings	1330
10.56.4	Inserting Non-Displayable Characters in Strings	1330
10.56.5	Replacing Text in a String at a Specified Point	1331
10.56.6	Searching for and Replacing Text in a String	1331
10.56.7	Special Characters for Match Pattern	1332
10.56.8	Special Characters for Match Regular Expression and Search and Replace String	1336
10.56.9	Splitting Strings	1344
10.56.10	Using Format Specifiers in Numeric Strings	1345
10.56.11	Entering a Carriage Return into a String	1345
10.56.12	Entering a Linefeed into a String	1346
10.56.13	Entering a Tab Character into a String	1346
10.56.14	Entering or Changing Text in a String Control	1346
10.56.15	Selecting and Entering Strings in Combo Box Controls at Run Time	1347
10.56.16	Specifying Custom Values for Combo Box Strings	1347
10.57	Adding Color Ramps to Dials, Knobs, and Gauges	1348
10.58	Adding Sliders or Needles	1349
10.59	Adding Text Labels to Numeric Objects with Scale	1349
10.60	Changing Scale Limits	1350
10.61	Changing Scrolling Ranges of Scroll Bar Controls	1350
10.62	Characters Numeric Controls and Indicators Accept	1351
10.63	Configuring the Time and Date in a Time Stamp Control	1352
10.64	Displaying Higher Digits of Precision in a Time Stamp Control	1353
10.65	Placing Arbitrary Markers on Scales	1354
10.66	Scaling Slide and Rotary Controls and Indicators	1354
10.67	Using Scroll Bar Controls	1355
10.68	Using the Increment and Decrement Buttons	1356
10.69	Valid Format Specifiers for Numeric Data Types	1356
10.70	Empty Paths	1357
10.71	Symbolic Paths	1358
10.72	Display Options	1359
10.72.1	Backslash ('\') Codes Display	1359
10.72.2	Configuring String Controls and Indicators	1361

10.72.3	Enable Wrapping	1362
10.72.4	Hex Display	1362
10.72.5	Limit to Single Line	1362
10.72.6	Normal Display	1363
10.72.7	Password Display	1363
10.72.8	Update Value while Typing	1363
10.72.9	Labeling	1363
10.72.10	Aligning Labels and Captions to Preset Positions	1364
10.72.10	Changing Captions Programmatically	1364
10.72.10	Changing Text Characteristics	1365
10.72.10	Creating Captions	1366
10.72.10	Creating Free Labels	1367
10.72.11	Editing Labels	1368
10.72.11	Labeling Objects	1368
10.72.11	Locking Labels and Captions	1369
10.72.11	Moving Owned Labels and Captions	1370
10.72.11	Rotating Text Labels	1370
10.72.20	Creating a Dialog Box	1371
10.72.21	Aligning Objects Using the Alignment Grid	1372
10.72.22	Creating Splitter Bars and Panes	1372
10.72.23	Creating Status Bars	1373
10.72.24	Creating Toolbars	1373
10.72.25	Configuring Splitter Bars	1374
10.72.26	Configuring Panes	1374
10.72.27	Viewing Run-Time Front Panel Bounds in Edit Mode	1374
10.72.28	Creating Template Controls	1375
10.72.29	Defining a Minimum Window Size	1375
10.72.30	Deleting Objects	1376
10.72.31	Maintaining Window Proportions with Screen Resolution	1376
10.72.32	Moving Objects	1377
10.72.33	Resizing Labels	1378
10.72.34	Resizing Multiple Objects to the Same Size	1378
10.72.35	Resizing Objects	1379
10.72.36	Selecting Objects	1380
10.72.37	Setting a Single Front Panel Object to Scale	1381
10.72.38	Setting All Objects on a Front Panel to Scale	1381
10.72.39	Fitting a Single Front Panel Object to a Pane	1382
10.72.40	Using Decorations on the Front Panel	1382
10.73	Creating Type Definitions and Strict Type Definitions	1382

10.74	Editing Saved Type Definitions and Saved Strict Type Definitions	1383
10.75	Manually Updating Type Definition Instances	1383
10.76	Reestablishing the Link to a Type Definition or Strict Type Definition	1384
10.77	SubVI and Express VI	1385
10.77.1	Displaying SubVIs and Express VIs as Icons or Expandable Nodes	1385
10.77.2	Inserting and Removing Inputs and Outputs	1385
10.77.3	Express VIs	1386
10.77.3.1	Creating an Express VI from an Existing VI	1386
10.77.3.2	Creating an Express VI from an Existing Express VI	1387
10.77.3.3	Creating an Express VI from a Blank VI	1387
10.77.3.4	Building Configuration Dialog Box VIs	1388
10.77.3.5	Displaying Configuration Settings in the Context Help Window	1389
10.77.3.6	Validating Express VIs	1391
10.77.3.7	Adding a Custom Title Option for Express VI Instances	1391
10.77.3.8	Renaming and Moving Express VIs	1392
10.77.3.9	Converting Data to the Dynamic Data Type	1392
10.77.3.10	Converting Dynamic Data to Other Data Types	1393
10.77.3.11	Mapping Data Using the Sensor Mapping Express VI	1393
10.77.4	SubVIs	1394
10.77.4.1	Placing SubVIs on Block Diagrams	1394
10.77.4.2	Replacing SubVI Nodes with SubVI Contents	1395
10.77.4.3	Opening and Editing SubVIs	1396
10.77.4.4	Creating SubVIs from Selections	1396
10.77.4.5	Avoiding SubVI Selections That Create Cycles	1396
10.77.4.6	Avoiding SubVI Selections That Contain Terminals inside Structures	1397
10.77.4.7	Avoiding SubVI Selections of More than 28 Objects that Require Terminals	1397
10.78	Terminals	1398
10.78.1	Assigning Terminals to Controls and Indicators	1398
10.78.2	Confirming Terminal Connections	1399
10.78.3	Deleting Terminal Connections on the Connector Pane	1399
10.78.4	Opening SubVI Front Panels During Execution	1399
10.78.5	Determining the Current Instance of a SubVI	1400
10.79	Selecting a Connector Pane Pattern	1401
10.80	Setting Required, Recommended, and Optional Inputs and Outputs	1401
10.81	Icons	1402
10.81.1	Creating Icons	1402
10.81.2	Creating a Library Icon	1402
10.81.3	Creating a VI Icon	1404

10.81.4	Creating a Custom-Shaped Icon	1406
10.81.5	Synchronizing with the Icon Library Manually	1407
10.82	Menus	1407
10.82.1	Customizing Menus	1407
10.82.2	Associating Existing Menus	1408
10.82.3	Customizing Menus Programmatically	1408
10.82.4	Customizing Menus Statically	1410
10.82.5	Handling Menu Selections	1411
10.82.6	Showing and Hiding Menu Bars	1412
10.82.7	Viewing Application Items	1412
10.82.8	Customizing Run-Time Shortcut Menus	1412
10.82.9	Associating Existing Run-Time Shortcut Menus	1413
10.82.10	Customizing Run-Time Shortcut Menus for Front Panels Programmatically	1413
10.82.11	Customizing Run-Time Shortcut Menus for Controls Programmatically	1414
10.82.12	Customizing Run-Time Shortcut Menus for Controls Statically	1415
10.82.13	Disabling Run-Time Shortcut Menus	1416
10.82.14	Handling Run-Time Shortcut Menu Selections	1416
10.83	Troubleshooting	1417
10.83.1	Probes	1417
10.83.1.1	Caveats and Recommendations When Using Custom Probes	1417
10.83.1.2	Creating Custom Probes	1418
10.83.1.3	Creating Probes after a VI Runs	1420
10.83.1.4	Using Indicators to View Probed Data	1421
10.83.1.5	Using Supplied Probes	1421
10.83.1.6	Using the Generic Probe	1425
10.83.1.7	Using the Probe Tool	1425
10.83.2	Managing Breakpoints	1427
10.83.3	Controlling Error Notification	1428
10.83.4	Defining Custom Error Codes	1429
10.83.5	Defining a Custom Error Code from the Error Ring	1430
10.83.6	Defining Custom Error Codes to Distribute throughout Your Application	1431
10.83.7	Distributing Custom Error Codes in Text Files	1432
10.83.8	Defining Custom Error Codes Using the General Error Handler VI	1432
10.83.9	Finding Error Code Descriptions	1433
10.83.10	Generating Error Cluster Reports	1434
10.83.11	Handling Errors Using Case Structures	1435
10.83.12	Handling Errors Using Conditional Terminals on Loops	1435
10.83.13	Selecting Built-In Errors from the Error Ring	1436

10.83.1	Setting up Error I/O in a VI	1436
10.83.1	Using Error Clusters	1437
10.83.1	Debugging Applications and Shared Libraries	1437
10.83.1	Troubleshooting the Build Process	1439
10.84	Documenting in LabVIEW	1440
10.84.1	Documenting VIs	1440
10.84.2	Creating a Bookmark Manager	1440
10.84.3	Creating and Editing VI Descriptions	1441
10.84.4	Creating Object Descriptions and Tip Strips	1441
10.84.5	Formatting Text in the Context Help Window	1442
10.84.6	Generating HTML Documentation for VIs, Templates, and Objects	1442
10.84.7	Generating RTF Documentation for VIs, Templates, and Objects	1443
10.84.8	Generating Text Documentation for VIs, Templates, and Objects	1444
10.84.9	Creating Compiled Help Files	1445
10.84.10	Linking VIs to Custom Documentation	1445
10.84.11	Naming Conventions for Image Files that LabVIEW Generates	1447
10.84.12	Printing Documentation for VIs, Templates, and Objects	1448
10.84.13	Printing Documentation for VIs in Multiple Application Instances	1448
10.84.14	Creating HTML Documents and Embedding Front Panel Images	1449
10.84.15	Embedding a Front Panel Image in an Existing HTML Document	1450
10.84.16	Embedding a VI in an Existing HTML Document	1452
10.85	Printing in LabVIEW	1453
10.85.1	Enhancing VI Printouts	1453
10.85.2	Generating Images of Individual Front Panel Objects	1454
10.85.3	Printing the Front Panel of a VI after the VI Runs	1455
10.85.4	Printing VIs Programmatically	1455
10.85.5	Troubleshooting Printer Errors	1456
10.85.6	Using a SubVI to Print Data from a Higher Level VI	1457
10.86	Reports in LabVIEW	1457
10.86.1	Generating Reports	1457
10.86.2	Formatting Data for Report Generation	1458
10.86.3	Using Tokens in Reports	1458
10.87	Adding New Exported VIs Entries	1458
10.88	Editing Existing Exported VIs Entries	1459
10.89	Removing VIs from the Exported VIs List	1459
10.90	Adding New Machine Access List Entries	1459
10.91	Editing Existing Machine Access List Entries	1460
10.92	Removing Machine Access List Entries	1460
10.93	Setting Machine Access for the VI Server	1461

10.94 Removing Browser Access List Entries	1461
10.95 Calling Shared Libraries in LabVIEW	1461
10.96 Building a Function Prototype	1463
10.97 Completing the.c File	1466
10.98 Importing Functions from a Shared Library File	1467
10.99 Example: Importing Functions from a Shared Library File	1470
10.10 Troubleshooting the Import Shared Library Wizard	1473
10.10 Using the Flatten To String Function	1474
10.10 Creating Dotted Properties and Methods	1476
10.10 Creating Properties and Methods	1476
10.10 Creating Properties and Methods with the Class Browser Window	1478
10.10 Registering and Handling .NET and ActiveX Events	1479
10.10 Creating .NET Controls in a .NET Container	1481
10.10 Creating .NET Objects on the Block Diagram	1482
10.10 Using Static .NET Properties and Methods	1484
10.10 Loading .NET Framework 2.0, 3.0, and 3.5 Assemblies in LabVIEW	1484
10.11 Loading Changes to a .NET Framework Assembly in Memory	1486
10.11 Specifying Which Version of a .NET Assembly to Use	1486
10.11 Debugging Calls to .NET Framework Assemblies	1488
10.11 ActiveX	1489
10.113 Accessing an ActiveX-Enabled Application	1489
10.113 Accessing ActiveX Properties	1489
10.113 Activating LabVIEW as an ActiveX Server	1490
10.113 ActiveX Container Overview	1490
10.113 Calling an ActiveX Object from LabVIEW	1491
10.113 COM/Automation and LabVIEW Data Types	1492
10.113 Creating New Controls in an ActiveX Container	1492
10.113 Creating New Documents in an ActiveX Container	1493
10.113 Editing ActiveX Documents	1493
10.113 Inserting an ActiveX Object on the Front Panel	1494
10.113 Inserting Existing Controls or Documents in an ActiveX Container	1494
10.113 Selecting an ActiveX Class	1495
10.113 Using ActiveX Property Pages	1495
10.113 Using the ActiveX Control Property Browser	1495
10.114 Editing Existing Property and Invoke Nodes	1496
10.115 Linking Property Nodes and Invoke Nodes to Terminals	1496
10.116 Connectivity in LabVIEW	1497
10.116 Developing and Distributing an Application	1497
10.116 Building a Universal Framework on macOS	1503

10.116.3	Creating Packages for Distribution	1504
10.116.4	Configuring Run-Time Languages for Applications and Shared Libraries	1505
10.116.5	Copying LabVIEW Work Environment Options	1507
10.116.6	Creating an About Dialog Box for a Stand-Alone Application	1508
10.116.7	Enabling the Web Server in Stand-Alone Applications	1509
10.116.8	Retrieving Application and Shared Library Versions Programmatically (Windows)	1509
10.116.9	Using a Configuration File with a Stand-Alone Application	1510
10.116.10	Characteristics of LabVIEW-Built .NET Framework Assemblies	1511
10.116.11	Characteristics of LabVIEW-Built Shared Libraries	1513
10.116.12	Caching Installer Distribution Components (Windows)	1514
10.116.13	Caveats and Recommendations for Building Installers	1514
10.116.14	Including Additional National Instruments Installers in an Installer Build	1516
10.116.15	Installing LabVIEW-Built Installers Silently	1518
10.116.16	Installing LabVIEW-Built Installers	1519
10.116.17	Using the TDM Excel Add-In (Windows)	1519
10.116.18	Working with the Application Manifest	1520
10.116.19	Running Operations Using the Command Line Interface for LabVIEW	1521
10.116.20	Creating Custom Command Line Operations	1523
10.116.21	Changing the Default Ports for TCP-Based NI-PSP (NI Linux Real-Time, Windows)	1526
10.116.22	JSONP Support	1527
10.116.23	Using Buffering with DataSocket Functions	1527
10.116.24	Programmatically Opening and Closing Data Connections	1529
10.116.25	Writing Live Data Programmatically	1529
10.116.26	Specifying the Buffer Size Using DataSocket Functions	1530
10.116.27	Reading the Buffered Data a Server Writes	1530
10.116.28	Monitoring the Buffer Size of a Data Connection	1531
10.116.29	Reading Live Data Programmatically	1532
10.116.30	Controlling a Front Panel Control Remotely Using the dstp, opc, ftp, http, or file Protocol	1532
10.116.31	Enabling the dstp, opc, ftp, http, or file Connection for a Control	1534
10.116.32	Reading Front Panel Data Using the dstp, opc, ftp, http, or file Protocol	1535
10.116.33	Writing Front Panel Data Using the dstp, opc, ftp, http, or file Protocol	1536
10.116.34	Web Service	1537
10.116.34.1	Integrating Static Content into a Web Service (Real-Time, Windows)	1537
10.116.34.2	Testing and Debugging a Web Service on the NI Web Server (Windows)	1539
10.116.34.3	Setting Up the NI Web Server for Web Services (Windows)	1540

10.116.304	Configuring Web Services Security when Using the NI Web Server (Windows)	1541
10.116.345	Publishing a Web Service through a Package (Windows)	1543
10.116.346	Publishing a Web Service through a Stand-Alone Application (Real-Time, Windows)	1544
10.116.347	Publishing a Web Service through an Installer (Windows)	1546
10.116.348	Publishing and Unpublishing a Stand-Alone Web Service (Real-Time, Windows)	1546
10.116.349	Testing and Debugging a Web Service on the Application Web Server (Real-Time, Windows)	1547
10.116.350	Setting Up the Application Web Server for Web Services	1548
10.116.301	Configuring Web Services Security when Using the Application Web Server (Real-Time, Windows)	1549
10.116.302	Configuring the Root Directory of the Web Server	1552
10.116.340	Tutorials	1552
10.116.351	Adding New Browser Access List Entries	1570
10.116.352	Allowing and Denying Web Browser Access	1570
10.116.353	Editing Existing Browser Access List Entries	1571
10.116.354	Front Panels	1571
10.116.355	Controlling an Application or Front Panel Remotely Using a Browser or LabVIEW	1571
10.116.356	Enabling and Configuring the Web Server for Remote Front Panels	1572
10.116.357	Enabling TLS/SSL Security for Remote Front Panels (Windows)	1573
10.116.358	Errors When Connecting to Remote Front Panels	1573
10.116.359	Functionality Not Supported in Viewing and Controlling Remote Front Panels	1574
10.116.360	Locking and Unlocking Remote Front Panels	1575
10.116.361	Managing Client Traffic to a Front Panel	1576
10.116.362	Publishing VIs and Stand-Alone Applications on the Web	1576
10.116.363	Sending an Email from a VI	1577
10.116.364	Setting Time Limits on Client Control	1578
10.116.365	Specifying Which VIs Are Visible on the Web	1579
10.116.366	Syntax for Browser Address Lists	1579
10.116.367	Syntax for the VIs Lists	1580
10.116.368	Viewing an Application or Front Panel Remotely Using a Browser	1582
10.116.369	Viewing an Application or Front Panel Remotely Using LabVIEW	1583
10.116.370	Web Server Syntax Examples	1584
10.116.400	Creating a TCP Client	1584
10.116.401	Creating a TCP Server	1585
10.116.500	Domains	1586
10.116.501	Using the Domain Account Manager (Windows)	1586

10.116.502	Adding a Network Domain (Windows)	1586
10.116.503	Creating a Local Domain (Windows)	1587
10.116.504	Creating and Configuring Domain Groups (Windows)	1587
10.116.505	Creating and Configuring Domain Users (Windows)	1588
10.116.506	Destroying a Local Domain (Windows)	1589
10.116.507	Exporting and Importing a Local Domain Using the Domain Account Manager (Windows)	1590
10.116.508	Importing a Lookout Account (Windows)	1591
10.116.509	Logging In to a Domain (Windows)	1592
11.	LabVIEW Measurement Concepts	1593
11.1	Overview of Virtual Instruments and Measurements	1593
11.2	Controlling Instruments	1593
11.3	Instruments and DAQ Devices	1594
11.3.1	General-Purpose DAQ	1594
11.3.2	Instrument I/O	1596
11.3.3	Waveform Control and Digital Waveform Control	1597
11.3.4	Data and String Manipulation Techniques	1599
11.3.5	Comparing DAQ Devices and Computer-Based Instruments for Data Acquisition	1601
11.3.6	Configuring DAQ Devices, Instruments, and Other Devices	1601
11.4	Types of Instruments	1603
11.4.1	Types of Instruments	1603
11.4.2	GPIB Communication	1603
11.4.3	Modular Instruments	1604
11.4.4	Serial Port Communication	1604
11.4.5	Ethernet Communication	1606
11.4.6	USB Communication	1606
11.4.7	Other Types of Communication	1606
11.5	Instrument Drivers	1607
11.5.1	Using Instrument Drivers	1607
11.5.2	Types of Instrument Drivers	1608
11.5.3	Finding and Installing Instrument Drivers	1609
11.5.4	Accessing Instrument Driver VIs	1609
11.5.5	Getting Started with Instrument Drivers	1610
11.5.6	Interactively Testing Instrument Driver VIs	1611
11.5.7	Creating Instrument Control Applications	1611
11.5.8	Modifying an Existing Instrument Driver	1612
11.5.9	Creating Instrument Drivers	1613
11.5.10	Instrument Control Error Handling	1613
11.6	Measurement Applications	1614

11.6.1	Creating a Typical DAQ Application	1614
11.6.2	Creating a Typical FieldPoint Application	1616
11.6.3	Using Properties with Measurement Applications	1616
11.7	VISA in LabVIEW	1616
11.7.1	Using VISA in LabVIEW	1616
11.7.2	Selecting a Resource Name	1617
11.7.3	Verifying VISA Communication	1618
11.7.4	Creating a Typical VISA Application	1618
12.	LabVIEW Shared Libraries	1619
12.1	Calling Shared Libraries	1619
12.1.1	Configuring the Call Library Function Node	1620
12.1.2	Specifying the Location of Shared Libraries on Disk	1631
12.1.3	Calling External APIs	1633
12.1.4	Example 1: Call a Shared Library That You Built	1636
12.1.5	Example 2: Call a Hardware Driver API	1638
12.1.6	Example 3: Call the Win32 API	1640
12.1.7	Unicode Versions and ANSI Versions of Functions	1643
12.1.8	Additional Windows Examples of LabVIEW Calls to DLLs	1646
12.2	Debugging Shared Libraries and Calls to Shared Libraries	1646
12.2.1	Module Definition Files	1650
12.2.2	Using Arrays and Strings in the Call Library Function Node	1651
12.3	Importing Shared Libraries	1655
12.3.1	Advantages of the Import Shared Library Wizard Compared to the Call Library Function Node	1655
12.3.2	Supported Data Types for the Import Shared Library Wizard	1656
12.3.3	Using the Import Shared Library Wizard	1659
12.4	LabVIEW Manager Functions in Shared Libraries	1660
12.4.1	Using LabVIEW Manager Functions in Shared Libraries	1660
12.4.2	LabVIEW Manager Data Types	1663
12.4.3	LabVIEW Manager Function Errors	1668
12.4.4	Memory Manager	1672
12.4.5	File Manager	1676
12.4.6	Support Manager	1680
13.	Signal Processing and Analysis	1681
13.1	Signal Generation in LabVIEW	1681
13.2	IIR Filter Characteristics in LabVIEW	1682
13.3	Curve Fitting in LabVIEW	1682
13.4	Spline Interpolation	1683
13.5	Basic Linear Algebra Subroutines	1684
13.5.1	Introduction to Basic Linear Algebra Subroutines	1684

13.5.2	Choosing Between BLAS VIs and Linear Algebra VIs	1685
13.6	Solving ODEs in LabVIEW	1686
13.7	Solving PDEs in LabVIEW	1687
13.8	Point-By-Point Analysis	1688
13.8.1	Point-by-Point Analysis and Array-Based Analysis in	1688
13.8.2	Performing Analysis without Buffers of Data	1690
13.8.3	Case Study of Point-by-Point Analysis	1691
13.8.3.1	Case Study of Point-By-Point Analysis Application Built in LabVIEW	1691
13.8.3.2	Point-By-Point Analysis of Train Wheels	1691
13.8.3.3	Overview of the LabVIEW Point-By-Point Solution	1692
13.8.3.4	Characteristics of a Train Wheel Waveform	1692
13.8.3.5	Analysis Stages of the Train Wheel	1693
14.	Accessing Scanned I/O Data	1696
14.1	Using I/O Variables (Real-Time, Windows)	1696
14.2	Accessing Scanned I/O Data (Real-Time, Windows)	1706
14.3	NI Scan Engine Faults (Real-Time, Windows)	1709
14.4	Reading and Writing I/O Variable Containers as Arrays (Real-Time, Windows)	1711
15.	PID and Fuzzy Logic	1713
15.1	Tutorial: Using the Fuzzy System Designer	1713
15.1.1	Creating Input Linguistic Variables	1714
15.1.2	Creating Output Linguistic Variables	1716
15.1.3	Creating a Rule Base with Pre-Generated Rules	1717
15.1.4	Modifying Pre-Generated Rules	1718
15.1.5	Specifying a Defuzzification Method	1719
15.1.6	Testing the Fuzzy System	1720
15.1.7	Controlling the Fuzzy System	1720
15.2	Tutorial: Modifying a Fuzzy System with the Fuzzy Logic VIs	1721
15.2.1	Observing the Fuzzy System	1721
15.2.2	Loading the Fuzzy System	1722
15.2.3	Modifying a Linguistic Variable	1722
15.2.4	Modifying Membership Functions	1723
15.2.5	Modifying a Rule	1724
15.2.6	Creating Antecedents	1725
15.2.7	Creating Consequents	1726
15.2.8	Combining the Antecedents and Consequents for a Rule	1726
15.2.9	Saving the Fuzzy System	1728
15.3	Fuzzy System	1729
15.3.1	Multi-Loop Control	1729
15.3.2	Non-Overlapping Input Terms	1729

15.3.3	Partially Overlapping Input Terms	1731
15.4	Fuzzy Controllers	1733
15.4.1	Dual-Input Fuzzy Controllers	1733
15.4.2	Fault Protection	1734
15.4.3	Filtering Control Inputs	1735
15.4.4	Fully Overlapping Input Terms	1735
15.4.5	Fuzzy Controllers	1737
15.4.6	Fuzzy Controllers in Closed-Loop Control Structures	1738
15.4.7	Fuzzy Logic	1739
15.4.8	Fuzzy System Design Overview	1740
15.4.9	Fuzzy Systems	1740
15.4.10	Fuzzy-PI Controllers	1741
15.4.11	Gain Scheduling	1742
15.4.12	I/O Characteristics of Fuzzy Controllers	1744
15.5	PID Algorithms	1745
15.6	PID Control	1747
15.7	PID Relay Autotuning Technique	1748
15.8	PID Topologies	1749
15.9	Programmatically Autotuning a Previously Implemented PID Controller	1750
15.10	Programmatically Implementing and Autotuning a PID Controller	1751
15.11	Setpoint Profile Generation	1752
15.12	Singletons as Output Terms with Fully Overlapping Input Terms	1753
15.13	Specifying a Consequent Implication Method	1755
15.14	Specifying an Antecedent Connective	1756
15.15	Undefined Input Term Intervals	1758
15.16	Using Fuzzy Controllers with PID Controllers	1760
16.	PID Autotuning	1762
16.1	Implementing Autotuning with the Autotuning Wizard	1762
16.2	Implementing PID Controllers with the PID VIs	1763
16.3	Implementing the PID Algorithm with the PID VIs	1764
16.4	Internal Model Control Autotuning Method	1767
16.5	Lead/Lag	1768
16.6	Logging Parameter Values After Autotuning	1768
16.7	Online Autotuning Techniques	1769
16.8	Advanced PID Algorithm	1770
16.8.1	Autotuning Methods for Testing PID Parameters	1773
16.8.2	Autotuning PID Controllers (PID and Fuzzy Logic)	1774
16.9	Relay Feedback Autotuning Technique	1775
16.10	Reproducing a Given I/O Characteristic	1775

16.11 Step Closed Loop Autotuning Technique	1777
16.12 Step Open Loop Autotuning Technique	1777
16.13 Updating PID Parameters After Autotuning	1777
16.14 Ziegler-Nichols Autotuning Method	1779
17. Defuzzification	1781
17.1 Center of Area (CoA)	1781
17.2 Center of Maximum (CoM)	1782
17.3 Center of Sums (CoS)	1784
17.4 Chien-Hrones-Reswick Autotuning Method	1784
17.5 Cohen-Coon Autotuning Method	1786
17.6 Complete Rule Bases	1786
17.7 Control Output Rate Limiting	1788
17.8 Converting Between Percentage of Full Scale and Engineering Units	1788
17.9 Creating a Rule Base	1788
17.10 Creating Linguistic Variables	1790
17.11 Creating Membership Functions	1791
17.12 Defuzzification Methods	1797
17.13 Designing a Control Strategy	1799
17.14 Different Overlapping Degrees of Membership Functions for Output Terms	1802
17.15 Mean of Maximum (MoM)	1803
17.16 Modified Center of Area (mCoA)	1804
17.17 Selecting a Defuzzification Method	1805
17.18 Using the Mean of Maximum (MoM) Defuzzification Method	1806
17.19 Wide and Small Membership Functions for Output Terms	1809
18. Distributed System Manager Overview	1811
18.1 Distributed System Manager Components	1812
18.1.1 File Menu	1813
18.1.2 Actions Menu	1813
18.1.3 View Menu	1814
18.1.4 Help Menu	1815
18.2 Forcing I/O Variable Values (NI Scan Engine)	1815
18.3 I/O Servers	1817
18.4 Observing Trend View of Variables	1817
18.5 Programmatically Accessing Published Values	1817
18.6 Saving and Restoring a System Manager Workspace	1818
18.7 Shared or Network Variables	1818
18.8 Shared Variable Engine	1819
18.9 Synchronizing Time for Networked Computers	1819
18.10 Systems	1820

18.11 VI States	1820
18.12 Viewing and Setting the Scan Engine Mode (NI Scan Engine)	1821
18.13 Viewing and Clearing Faults (NI Scan Engine)	1822
18.14 Alarms and Events	1822
18.14.1 Acknowledging Alarms and Events	1822
18.14.2 Alarm Probe View	1823
18.14.3 Alarms and Events View	1823
18.14.4 Alerts Probe View	1824
18.14.5 CPU/Memory Probe View	1824
18.14.6 Date Time Probe View	1825
18.14.7 I/O Variable Probe View	1825
18.14.8 OpenCSRs Probe View	1826
18.14.9 Process Probe View	1827
18.14.10 Scan Engine Probe View	1827
18.14.11 ServerCertificates Probe View	1829
18.14.12 Variable Probe View	1829
18.14.13 Web Server Probe View	1830
18.14.14 Web Server URL Mapping View	1830
18.14.15 Web Service Probe View	1831
18.14.16 Configure Web Services Security View	1832
18.14.17 Configuring Alarms for Shared Variables (DSC Module)	1832
18.14.18 Monitoring Alarms and Events	1833
18.14.18.1 Monitoring and Editing Data Item Ranges	1833
18.14.18.2 Monitoring Data Items in Data Item Ranges	1834
18.14.18.3 Monitoring Variables with the Watch List View	1834
18.15 Creating Certificates and Processes	1835
18.15.1 Creating a Certificate from a CSR	1835
18.15.2 Creating a Certificate Signing Request	1835
18.15.3 Creating a Self-Signed Certificate	1836
18.15.4 Creating Processes	1837
18.15.5 Creating, Modifying, or Deleting Shared or Network Variables	1837
18.15.6 Custom VI-based Server - On Input Change Wizard	1838
18.15.7 Custom VI-based Server - Periodic Wizard	1839
18.15.8 Customizing the System Manager Workspace	1840
18.15.9 Processes	1841

Index

0

LabVIEW User Manual

The LabVIEW User Manual provides detailed descriptions of the product functionality and the step by step processes for use.

Looking for Something Else?

For information not found in the User Manual for your product, such as specifications and API reference, browse *Related Information*.



RELATED INFORMATION

[*LabVIEW and Microsoft Windows Compatibility*](#)

[*NI Hardware and Software Operating System Compatibility*](#)

[*Download LabVIEW*](#)

[*License Setup and Activation*](#)

[*LabVIEW Release Notes*](#)

[*LabVIEW Programming Reference Manual*](#)

[*LabVIEW Wiki Style Guide*](#)

[*NI Learning Center*](#)

LabVIEW Overview

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a programming environment used to create applications using the G programming language.

Definition

In contrast to text-based programming languages that use instructions to determine the order of program execution, LabVIEW uses dataflow programming. In data flow programming, the flow of data through the nodes on the block diagram determines the execution order of the VIs and functions. VIs, or virtual instruments, are LabVIEW programs that imitate physical instruments.

In LabVIEW, you build a user interface by using a set of tools and objects. The user interface is known as the front panel. After you build the front panel, you add code using graphical representations of functions to control the front panel objects. You add this graphical code, also known as G code or block diagram code, to the block diagram. The block diagram somewhat resembles a flowchart. The block diagram, front panel, and graphical representations of code compose a VI.

Key Features

- Wide range of analysis functions
- Interactive, programmable display elements
- Automated instrument and data acquisition drivers
- Industry-standard protocols and support for other languages
- High-Speed Measurement and Control Systems with FPGAs
- Measure Physical Systems with Sensors or Actuators
- Validate or Verify Electronic Designs
- Develop Production Test Systems
- Design Smart Machines or Industrial Equipment



RELATED INFORMATION

[Installing, Updating, Repairing, and Removing NI Software](#)

LabVIEW New Features and Changes

Learn about updates, including new features and behavior changes, introduced in each version of LabVIEW.

Discover what is new in the latest releases of LabVIEW.



RELATED INFORMATION

[LabVIEW Release Notes](#)

LabVIEW 2025 Q3 Features and Changes

Released June 2025

Introducing Nigel™, AI Advisor

Nigel is your trusted test and measurement advisor, trained on the NI platform and ready to help when you need it. Get VI design advice, learn LabVIEW concepts, and more!

Troubleshoot VIs Using New Debug Window

The new **Debug** window added in LabVIEW 2025 Q3 combines probe and breakpoint management in a single location, and replaces the **Probe Watch** window and **Breakpoint Manager** window.

The **Debug** window lists breakpoints and probes for all VIs in memory, and allows you to add, remove, rename, highlight, and export data for probes and breakpoints. You can also

use the window to enable and disable breakpoints, switch probe styles, and create custom probes.

You can use the **Debug** window to customize the window appearance and select if probe values appear directly on wires in your VI block diagram during execution.

[Idea submitted by NI Discussion Forums member Citabria]

Logging for Unwired Errors

LabVIEW 2025 Q3 gives you the option to record unwired errors in a log file, independent of whether you use the **Automatic Error Handling** dialog to report them.

Select All Text Using Ctrl+A

LabVIEW 2025 Q3 enables use of the keyboard shortcut Ctrl+A to select all text whenever editing a text field, including in labels, string controls, and dialogs.

[Idea submitted by NI Discussion Forums member littlesphaeroid]

Quickly Open Files Using Ctrl+Shift+O

LabVIEW 2025 Q3 adds the **Quick Open** dialog, which you can open using the keyboard shortcut Ctrl+Shift+O. The dialog displays recently-opened files that you can open, as well as all files in any project you have open.

[Idea submitted by NI Discussion Forums member AristosQueue (NI)]

Preserve Values by Default When Placing Diagram Disable Structures

In LabVIEW 2025 Q3, when placing a Diagram Disable Structure to enclose a section of code, LabVIEW matches output tunnels with input tunnels to wire values through in the Enabled frame by default. If LabVIEW cannot find the source for a refnum or LabVIEW class output tunnel, the tunnel is not configured with Use Default if Unwired.

[Idea submitted by NI Discussion Forums member Jim_Kring]

Added Context Help Notes Section for VIs

LabVIEW 2025 Q3 adds a Notes section to the Context Help for VIs to highlight settings that might be of interest when reviewing VI block diagrams. This section includes:

- Reentrancy, if the VI is configured for shared clone reentrant execution or preallocated clone reentrant execution.
- Execution Priority, if the VI is configured to a priority other than normal priority.

- Preferred Execution System, if the VI is configured to a Preferred Execution System other than same as caller or standard.

[Idea submitted by NI Discussion Forums member Mr._Jim]

New Context Help Notes Section for Call Library Function Node

LabVIEW 2025 Q3 adds a Notes section to the Context Help for Call Library Function Nodes. This section includes:

- Function prototype
- Library name or path

Left-Aligned Text for Improved Readability in Diagram Objects

In LabVIEW 2025 Q3, the text in some block diagram objects is left-aligned, for improved readability. The objects changed are:

- Bundle by Name
- Unbundle by Name
- Unbundle/Bundle Elements on In Place Element Structure
- Event Structure Data Nodes
- Get Waveform Components
- Build Waveform
- Get Digital Waveform Components
- Build Digital Waveform
- Get Digital Data Components
- Build Digital Data

[Idea submitted by NI Discussion Forums member PJM_LabVIEW]

Use LabVIEW Command Line Interface Without Enabling VI Server TCP Port

LabVIEW 2025 Q3 allows you to use the LabVIEW Command Line Interface (CLI) without enabling the VI Server TCP Port. If users do not enable the port, the CLI configures LabVIEW on a per-operation basis so it can execute the specified operation. These per-operation configurations do not persist after the operation executes.

All existing workflows that previously enabled the VI Server TCP port (with the default or user-defined port number) continue to work as they did in previous releases.

Generate VI Comparison Reports with the LabVIEW Command Line Interface

LabVIEW 2025 Q3 adds a new command to the LabVIEW Command Line Interface for generating VI Comparison reports. Use the `CreateComparisonReport` operation name along with the names of the two VIs you want to compare to output an HTML, XML, Word, or text file summarizing the differences between the files.

Support for .NET (Core) 8.0 Objects in Register Event Callback Function

The `Register Event Callback` function now supports handling .NET 8.0 objects in LabVIEW 2025 Q3.



NOTE

The .NET Container and building .NET assemblies are not yet supported with .NET 8.0.

Support for Windows Long Paths

LabVIEW 2025 Q3 adds support for Windows long paths.

[Idea submitted by NI Discussion Forums member JonathanLindsey]

Support for Additional Data Types in Array Export Commands

In LabVIEW 2025 Q3, the **Export Data to Clipboard** and **Export Data to Excel** commands in the right-click menu of arrays are enabled for arrays of paths as well as one-dimensional arrays of clusters containing only numeric, string, and path elements.

New Vertical Scrollbar for Free Labels

In LabVIEW 2025 Q3, free labels have a **Visible Items»Vertical Scrollbar** option in the right-click menu to show and hide a vertical scrollbar.



NOTE

If you create a VI with the vertical scrollbar shown on a free label and open it in LabVIEW 2025 Q1, the label will not respond to mouse wheel inputs correctly.

[Idea submitted by NI Discussion Forums member SteveChandler]

New Default Option for Compiled Code in Source Distributions

In LabVIEW 2025 Q3, new source distributions on desktop targets use the compiled code setting saved in each source file by default. Previously, the default was to embed compiled code in all source files.

Additional Changes

For hardware and software compatibility information, refer to the *Related Information*.



RELATED INFORMATION

[Nigel, AI Advisor](#) on page 115

[Debug Window - LabVIEW Programming Reference Manual](#)

[LabVIEW Linux Containers for CI/CD Workflows](#) on page 121

LabVIEW supports Linux containers for streamlining CI/CD workflows and other development automation tasks.

[.NET Events](#) on page 793

[LabVIEW and Microsoft Windows Compatibility](#)

[NI Hardware and Software Operating System Compatibility](#)

LabVIEW 2025 Q1 Features and Changes

Released January 2025

.NET (Core) 8.0 Support

LabVIEW 2025 Q1 supports loading and running .NET 8.0 assemblies on Windows. The following features are also supported with .NET 8:

- Building applications, packed libraries, and source distributions.
- Defining and using static properties and methods.



NOTE

The .NET Container, registering VIs for event callbacks, and building .NET assemblies are not yet supported with .NET 8.

Python Support

LabVIEW 2025 Q1 supports Python 3.11 and 3.12.



NOTE

LabVIEW 2025 will continue to support Python 3.9 and 3.10.

Changes to Compare VIs and Other LabVIEW Files

LabVIEW 2025 Q1 adds the HTML format for VI comparison reports. These reports include:

- Annotated images of differences, including different colors for additions, modifications, and removals.

- Images of changed VI icons and connector panes.
- Collapsible sections to help focus on the remaining differences during reviews.

Interactive comparison of VIs also has new annotations and buttons to copy images of differences to the clipboard. You can easily use these features in change comments, such as in Pull Requests with Git.

Project, library, and class files now contain better property labels and less redundant information. It makes it easier for you to review changes using third-party tools for text files.

New Row Options for Listbox, Multicolumn Listbox, Table, and Tree Controls

LabVIEW 2025 Q1 adds the following new options to listbox, multicolumn listbox, table, and tree controls:

- Added Word Wrap option. This option causes text in a cell to display in multiple lines if the column is not wide enough to display the complete text. This option will work best when used in combination with Autosize Row Height.
- Added alternating row background color option. This option makes it easier to associate text with the correct row, especially when the control is not showing horizontal lines. In high contrast themes, this feature is disabled when using system colors for both the text and the background color.

New Properties, Methods, and Events

LabVIEW 2025 Q1 includes the following new properties, methods, and events:

- The Project and Library classes include the following new property:
 - Save Version (read only) — Returns the save version of the project or library.
- The Project class includes the following new method:
 - Set Project Save Version — Sets the save version of the project. Version is specified as a short form LabVIEW version string, such as 24.0 representing LabVIEW 2024 Q1. The oldest supported version is 17.0. This method will load all project items. The items will prompt to be saved on close if not already saved in a compatible version.
- The Library class includes the following new method:
 - Set Library Save Version — Sets the save version of the library. Version is specified as a short form LabVIEW version string, such as 24.0 representing LabVIEW 2024 Q1. The oldest supported version is 17.0. This method will load all library and class items. The items will prompt to be saved on close if not already saved in a compatible version.
- The Listbox, Multicolumn Listbox, Table, and Tree classes include the following new properties:
 - Word Wrap — Determines whether the height of a row is based on explicit lines or wrapped text.

- Alternating Row Colors — Row background colors alternate to make row size and boundaries more obvious.
- Alternate Row Background Color — Background color used to draw alternate rows.

Updated LabVIEW Font Type Definition

The LabVIEW font type definition (`LVFontTypeDef.ct1`) now uses a color box instead of a numeric color value.

[Idea submitted by NI Discussion Forums member Petru_Tarabuta]

Additional Changes

- The *LabVIEW Style Guide* is now in the *LabVIEW Wiki*. For information about the *LabVIEW Style Guide*, refer to the *Related information*.
- For hardware and software compatibility information, refer to the *Related information*.



RELATED INFORMATION

[LabVIEW and Microsoft Windows Compatibility](#)

[NI Hardware and Software Operating System Compatibility](#)

[LabVIEW Wiki Style Guide](#)

LabVIEW 2024 Q3 Features and Changes

Released July 2024

Preview Feature: .NET Core (8.0) Support in LabVIEW

LabVIEW 2024 Q3 provides limited support for loading and running .NET Core (8.0) assemblies on Windows, including:

- Enabling constructor nodes, property nodes, and invoke nodes to work with .NET Core (8.0)
- Supporting the following four data types:
 - int
 - string
 - bool
 - char

To enable .NET Core (8.0) support in LabVIEW 2024 Q3, navigate to **Tools»Options**, select the **Preview Features** category, and select the **Enable .NET Core (8.0) Support** checkbox.

Saving All VIs in a Project for a Previous Version of LabVIEW

You can use the new **Save Version** option, available in the **Properties** dialog box of properties, libraries, and classes, to configure LabVIEW to save all VIs for a previous version of LabVIEW. If a VI contains features not compatible with the designated version, LabVIEW displays a warning and saves the VI for the earliest version that supports all features in that VI.

Improvements to Comparing VIs

- You can now highlight an item in the differences list or details list with a single click.
- You can use the left and right arrow keys to move between the differences list and the details list, and you can use the up and down arrow keys to move through the differences and details.
- In the **Differences** dialog box, a difference or detail appears italicized if it involves only front panel or block diagram cosmetic changes.
- The diagrams can automatically scale to fit an entire difference.
- You can swap the positions of the compared VIs by selecting **More commands»Swap VI Positions** in the **Differences** dialog box.



RELATED INFORMATION

[Saving for a Previous Version](#) on page 938

You can save VIs, LabVIEW projects and project libraries for a previous version of LabVIEW.

[Showing Differences between Two Compared VIs](#) on page 965

LabVIEW 2024 Q1 Features and Changes

Released January 2024

Support for openSUSE 15.5

LabVIEW 2024 Q1 now supports openSUSE 15.5.

New Package Dependency Management

You can now use JKI Dragon to manage package dependencies directly from your LabVIEW project.

New and Changed Objects

- The `Path` class includes the following new method:

- Browse for Path - Opens a file dialog to allow you to select a path. This method replicates the functionality of clicking the Browse Button of the Path control on the front panel.
- The Open VI Reference function includes the following change:
The `vi_path` input parameter now accepts an existing VI reference in addition to the string and path data types accepted in previous versions. This new option enables workflows such as:
 - Getting a strict reference from a non-strict reference
 - Creating another reentrant instance of a reentrant VI

LabVIEW 2023 Q3 Features and Changes

Released July 2023

Diagram Zoom

In LabVIEW 2023 Q3, you can zoom in or out when editing the block diagram of a VI using the **View** menu. **Ctrl** (Cmd on macOS) and the scroll wheel of the mouse can also be used to zoom in or out. A tip strip displays the zoom level percentage when zooming. To return to 100% zoom, use **View>>Actual Size**.

When you first select **View>Toggle Zoom** from the menu, the diagram will zoom out to fill the entire window. LabVIEW will restore the zoom level to its previous value when the same menu command is used a second time, but the diagram will be centered around the current mouse position.

New keyboard shortcuts include:

- View>Zoom In: Ctrl+= (or Ctrl+=)
- View>Zoom out: Ctrl--
- View>Actual Size: Ctrl-0
- View>Toggle Zoom: F2

[Idea submitted by NI Discussion Forums member tst.]

Double-click to Finish Wire

In previous versions of LabVIEW, double-click on the diagram resulted in creating a wire branch. In LabVIEW 2023 Q3, you can use the double-click gesture to complete the wire and place a connected control or indicator after creating one end of the wire with a data type. Control modifier can be used to create a constant. Void wires will only create wire branches. Right-clicking while creating a wire will display a menu of options:

- Create Constant
- Create Control
- Create Indicator

- Create Wire Branch

Find All Instances When VI Running

In LabVIEW 2023 Q3, you can right-click on a subVI on the diagram, or the icon of a VI (in the top-right corner of the VI window) and use the **Find All Instances** command even when the VI is running. In previous versions of LabVIEW, this command was only available for VIs in edit mode.

In addition, the right-click menu on a subVI may now contain the following commands at run-time:

- Open Malleable VI Front Panel
- Open Front Panel
- Open Generic VI
- Open Polymorphic VI
- Show VI Hierarchy

[Idea submitted by NI Discussion Forums member Technico.]

Quick Change List

In LabVIEW 2023 Q3, you can use the keyboard shortcut **Ctrl-Space** or the right-click shortcut **Quick Change** to display the Quick Change list when editing text in a free label. By selecting an item (such as Numeric), the label will change to that object type, and the label text will be converted to the object's value.

You can also **Edit>Paste Special** command to paste text from the clipboard and immediately display the Quick Change list.

You can also add custom plugins to the Quick Drop list. Refer to the instructions document in **resource\dialog\QuickChange\plugins\prefix_Template** for more information.

[Idea submitted by NI Discussion Forums member patibandla.]

Quick Change Palette

In LabVIEW 2023 Q3, after defining a rectangular selection on the diagram (and before releasing the mouse to finish the selection), you can use the keyboard shortcut **Ctrl-Space** to display a palette of items (such as loops) that can be placed in the position of the selection. In addition, one of the palette items will create a subVI from the selection.

Execution Highlighting

LabVIEW 2023 Q3 introduces Slow, Medium, Fast, and Very Fast options for the speed of Execution Highlighting. Medium speed matches the behavior of Execution Highlighting in the previous versions of LabVIEW.

Click and hold the Execution Highlighting toolbar button to display a menu of speed options. In addition to improving the visibility of the executed code, the animation of execution is given a new visual treatment.

[Idea submitted by NI Discussion Forums member JackDunaway.]

Building Applications and PPLs Improvements

LabVIEW 2023 Q3 has improved cache behavior for packed project libraries and applications.

The first build will populate the cache, and then the subsequent builds will be much faster.

Support for Apple Silicon

LabVIEW 2023 Q3 for macOS runs natively on Apple Silicon. The LabVIEW executable is a universal binary.

Text Editing Shortcuts on macOS and Linux

In LabVIEW 2023 Q3, macOS and Linux users can use the following Emacs-style editor keys when text fields are active:

- **Ctrl-A**: Beginning of line; with **Shift**, select to beginning of line
- **Ctrl-E**: End of line; with **Shift**, select to end of line
- **Ctrl-D**: Delete character forward
- **Ctrl-F**: Forward char; with **Shift**, extends selection
- **Ctrl-B**: Backward char; with **Shift**, extends selection
- **Ctrl-K**: Kill (cut) to end of line; supports append-to-pasteboard behavior for multiple consecutive **Ctrl-Ks**
- **Ctrl-N**: Next line; with **Shift**, extends selection
- **Ctrl-P**: Previous line; with **Shift**, extends selection
- **Ctrl-T**: Transpose characters at cursor
- **Ctrl-W**: Wipe region (cut selection)
- **Ctrl-Y**: Yank (paste)

By default, the **Control** and **Alt** (Meta) keys on Linux can be used interchangeably for menu shortcuts. When the text is active, the editor **Control** key bindings take precedence, but the **Alt** menu bindings remain active for menu shortcuts.

On macOS, **Cmd** key menu shortcuts are not affected by this change.

Ctrl-C, **Ctrl-X**, and **Ctrl-V** retain standard copy, cut, and paste behavior.

Config token **EnableEmacsTextEditingShortcuts=False** can be used to disable text editing bindings.

Cluster Consolidation Behavior

In LabVIEW 2023 Q3, clusters with the same type, but different cluster field names will be represented by different types in the built .NET Interop Assembly.

In the previous versions of LabVIEW, clusters with the same type were represented by the same type in the .NET Interop Assembly irrespective of the cluster field names.

To use the old behavior, add the following to the LabVIEW.ini file:

DotNetAssemblyGeneration.UseClusterFieldNames=False

LabVIEW 2023 Q1 Features and Changes

Released January 2023

LabVIEW Introduces a Tool to Identify and Install NI Drivers for a Project

LabVIEW 2023 Q1 includes a utility to assist you in identifying and installing NI drivers used by a LabVIEW project.

Select **Project>Package Dependencies** from the Project Explorer menu or click the **Package Dependencies** button in the Project Explorer toolbar to open the **Manage NI Package Dependencies** dialog. The dialog examines the project to discover which NI packages are used. You can click the **Save List** button to save a text file that lists the identified packages.

If one or more packages are not currently installed, the **Install** button will be enabled. You can click the **Install** button to launch NI Package Manager, and install the required packages.



NOTE

Building the list of packages to show in NI Package Manager may take several minutes.

Enable the **Show VIs** checkbox to understand the reason a package is listed as a dependency. This shows the VIs referenced by the project that the tool assumes are provided by the package.

**NOTE**

This tool might not correctly identify all the NI packages used by a project. The tool uses information from directories where NI installers install files. However, a LabVIEW project might reference files in a directory that is shared by multiple installers or the tool might be missing information for some installers.

Each time the dialog box opens, the tool attempts to update a component from ni.com to get the most-up-to-date directory information.

If the project uses an FPGA target, the dialog will identify LabVIEW FPGA Module as a dependency. However, the tool is currently unable to recommend a specific driver for the target. If the FPGA Module is installed, FPGA targets in the project will show their model numbers, and you can search for applicable drivers on ni.com.

Applications on Linux Support Custom Name and Icon

Starting in LabVIEW 2023 Q1, you can customize the application name and icon for built applications on Linux. Also, if you do not provide a custom icon, built applications will now use the system default application icon instead of the LabVIEW icon.

To customize the application name and icon, use the **Properties** dialog for the application's build specification. LabVIEW will generate a **.desktop** file along with the executable. You will need to move the **.desktop** file to the appropriate location such as `/usr/share/applications/` for your Linux distribution.

Terminals Height No Longer Changes With Font

Starting with LabVIEW 2023 Q1, terminals for the following functions will always be 16 pixels tall:

- Unbundle/Bundle by Name
- Locals/Globals
- Invoke Node
- Property Node
- Non-icon View for SubVI Calls
- Express VIs
- Event Structure Event Data Node
- Register Event Callback
- Register for Events
- Call Library Function Node Name View
- In Place Element Structure Unbundle/Bundle Elements
- Get Waveform Components

This is a LabVIEW 2023 Q1 feature that has been inspired by the **LabVIEW Idea Exchange**. If the new terminal size introduces wire bends to your VI diagrams, you can use the VI Analyzer Wire Bends test to find them. You will need to remove the wire bends manually.

[Idea submitted by NI Discussion Forums member **Andrey_Dmitriev**.]

Python Support

LabVIEW 2023 Q1 adds support for Python 3.10. In addition, LabVIEW 2023 Q1 supports Anaconda and venv virtual environments with the new Open Virtual Environment Session function.

Support for macOS 13

LabVIEW 2023 Q1 now supports macOS 13.

New Properties, Methods, and Events

LabVIEW 2023 Q1 includes the following new properties, methods, and events.

- The `Application` class includes the following new property:
 - `Application: Version Display Name` - Returns the display name of the application version, such as 2023 Q1.

LabVIEW 2022 Q3 Features and Changes

Released July 2022

Compare VIs within LabVIEW

Compare VIs feature is now available with all editions of LabVIEW 2022 Q3 (Base, Full, and Professional) and not limited to a Professional Edition license.

Python Support

LabVIEW 2022 Q3 supports use of Python Node with Python Object Refnum. Use this type of refnum to pass a python object as return type or input parameter to Python Node.

Changes to Default under Options

In LabVIEW 2022 Q3, the default for **Separate compiled code from new files** has been changed to **ENABLED**.

Call MATLAB Function

You can set a breakpoint on the **Call MATLAB Function** and then use the **step into** debugging command to open the MATLAB(R) editor and execute your script. If you have

multiple versions of MATLAB installed, you can right-click on the function and use **Open in MATLAB** submenu to specify the version called by LabVIEW.

Uninit method of Actor.lvclass

In the Actor Framework, the Actor class now has a new Uninit method. Actors can override this method to release resources acquired during the execution of Pre Launch Init.vi or Actor.vi. This method always executes even if there is an error returned from the earlier methods.

Support for Drivers/Toolkits Independent of LabVIEW Version

Earlier versions of LabVIEW required add-on content such as toolkits and drivers to be located within the LabVIEW directory. Starting with LabVIEW 2022 Q3 and later, LabVIEW will additionally load such content from a shared location called LVAddons. On Windows, the default location for LVAddons is C:\Program Files\NI\LVAddons. Note that only a subset of NI drivers and toolkits will install into this location with their 2022 Q3 releases. After a driver or toolkit transitions to using the LVAddons location, it will work with the newer versions of LabVIEW without needing to be upgraded or reinstalled.

New Help Experience

In LabVIEW 2022 Q3, help links will direct you to the new online LabVIEW help when the system is connected to the internet. When the system is not connected to the internet, help links will direct you to the new NI Offline Help Viewer that is installed with LabVIEW. Your network connection determines whether LabVIEW uses online or offline help. NI Help Preferences application, however, allows users to always use the offline help viewer.



NOTE

LabVIEW 2022 Q3 is the first NI product to adopt a modern help viewer to replace CHM help files. Over time, we will transition other LabVIEW modules and toolkits to the new NI Offline Help Viewer. In the meantime, we will continue to install CHMs for your LabVIEW add-ons.

New and Changed Properties, Methods, and Events

New Properties, Methods, and Events

LabVIEW 2022 Q3 includes the following new properties, methods, and events.

- The Page CClass includes the following new property:
 - Z-Order:Controls[]

Changed Properties and Methods

LabVIEW 2022 Q3 includes the following changed properties and methods:

- The `Pane Class` includes the following changed properties:
 - `All Objects[]` is now `Z-Order:All Objects[]`
 - `Controls[]` is now `Z-Order:Controls[]`
 - `Decorations[]` is now `Z-Order:Decorations[]`
- The `Page Class` includes the following changed properties:
 - `CtlsOnPage` is now `Tabbing Order`
 - `DecosOnPage` is now `Z-Order:Decorations[]`
 - `ObjectsOnPage` is now `Z-Order:All Objects[]`
- The `Page Class` includes the following changed method:
 - `Reorder Controls` method is now `Rearrange Tabbing Order`

Updates and Changes for LabVIEW Extended Support Versions

Browse updates and changes made in LabVIEW versions on extended support.



NOTE

If you cannot find changes for your version, it might be a more recent version, documented as a new feature. Or, your version might not have included user-facing updates. You can find more information about non-visible changes, such as bug fixes, compatibility updates, and stability adjustments or maintenance adjustments, in the product *Release Notes*, available on ni.com.



RELATED INFORMATION

[LabVIEW New Features and Changes](#) on page 47

Learn about updates, including new features and behavior changes, introduced in each version of LabVIEW.

LabVIEW 2021 Features and Changes

Released August 2021

Improvements to the Probe Display

Default probes for integer wires show the radix option in the probe display, which you can use to view data in decimal, hexadecimal, octal, or binary format, or SI notation.

SFTP Support

SFTP VIs and nodes allow users to securely interact with data and files on an SFTP server.

Python Support

LabVIEW 2021 supports Python 3.6 through 3.9 and removes support for Python 2.7.

Run When Opened

When a user launches a VI that is configured to run when opened from a disk location other than standard NI installer locations, such as the LabVIEW 20XX or National Instruments\Shared folder, the user will be prompted to choose whether to run the VI or open it in edit mode.

nicurl Uses System Certificates on Windows

nicurl uses the OS certificate store on Windows rather than its own HTTPS certificate bundle.

OS Support

LabVIEW no longer supports Windows Vista/XP/7/8.x, Windows Server 2003/2008/2012, or any 32-bit Windows operating system.

Hidden Controls

You can choose to show hidden controls while editing by selecting **Edit»Show Hidden Controls While Editing**. Controls are still hidden at run time.

Use LVCompare with Git

You can use LVCompare with Git to diff VIs. You no longer need a wrapper script to convert paths from relative paths to absolute paths before passing them to LVCompare.

Improvements to Workflow for Comparing VIs and VI Hierarchies

When comparing VIs or VI hierarchies, you can filter out various types of diffs interactively. Previously, you had to specify filters prior to launching a diff and had to relaunch the diff to change the filters. These filters include attributes, front panel changes, including changes in object position and size, as well as functional and cosmetic changes on the block diagram.

Image Depth for GetImage and GetImageScaled Methods

The GetImage and GetImageScaled methods now support a 32-bit image depth.

Call MATLAB Functions

Use the MATLAB functions to call MATLAB® code from LabVIEW.

Named Tuples Support

When using the Python Node you can marshal clusters to named tuples so you can reference tuple items by name.

VI Names Listed in VI Abort Dialog

When closing a project, deleting a target, or quitting LabVIEW, if user VIs are running, LabVIEW informs the user which top-level VIs will be aborted as part of that operation.

Improved Wire Routing

When drawing a wire between terminals, auto-routing leaves space between the wire and any unwired terminals so that you can wire the lower terminals without crossing wires.

LabVIEW 2020 Features and Changes

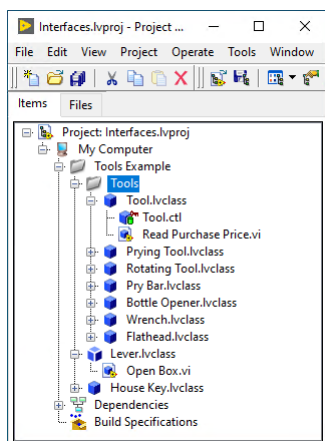
Released May 2020

Improving Code Flexibility Using LabVIEW Interfaces

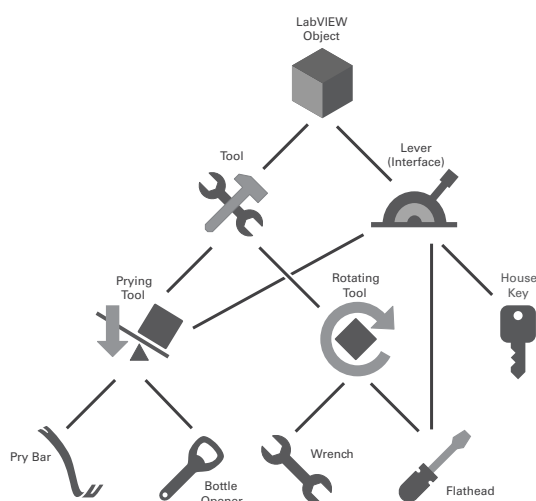
LabVIEW 2020 introduces interfaces. An *interface* can be thought of as a class without a private data control, but that small difference enables an interface to serve entirely different purposes in software architectures than classes. Specifically, interfaces enable a form of multiple inheritance.

An interface declares a role that an object can play without defining how to perform that role. By inheriting from an interface, a class declares that its objects fulfill that role, and the class becomes responsible for specifying how the behaviors are performed. When a class inherits from multiple interfaces, its objects can be passed into multiple software modules requiring different roles.

The following project includes the Tool class with several child classes of different tools. The project also includes the Lever interface. You can see that the Tools class has a control that defines the data in the class, whereas the Lever interface does not have a control because interfaces do not have private data. A class is represented by a solid cube (■). An interface is represented by the faces of a cube (⏏). Interfaces and classes use the same file extension `.lvclass`.



The following image illustrates the inheritance relationship among the Tool classes, the House Key class, and the Lever interface. The Tool class and the Lever interface both inherit from LabVIEW Object. In addition to having their own methods and methods of their respective parent classes, the Prying Tool class and the Flathead class also inherit from the Lever interface because they both can be used as a lever. They inherit the methods of the Lever interface. Since Lever is an interface rather than a class, this multiple inheritance is legal. The three unrelated classes (Prying Tool, Flathead, House Key) have a common ancestor (Lever) other than LabVIEW Object.



You can create an interface in the following ways:

- Right-click **My Computer** in the **Project Explorer** window and select **New»Interface** from the shortcut menu.
- Select **File»New** and select **Other Files»Interface** from the **Create New** list.
- Use the LabVIEW `Class>Create Interface` method.

Refer to the *New and Changed Properties, Methods, and Events* section for more information about the new properties and methods for interfaces.

Refer to the following projects for examples of using interfaces:

- `labview\examples\Object-Oriented Programming\Basic Interfaces\Basic Interfaces.lvproj`
- `labview\examples\Object-Oriented Programming\Actors and Interfaces\Actors and Interfaces.lvproj`

LabVIEW Web Services Enhancements

LabVIEW 2020 provides functionalities for developing LabVIEW Web services and publishing Web services to the NI Web Server, in addition to the Application Web Server that continues to be supported.

The NI Web Server is a production-grade web server that can host user-authored services, such as LabVIEW Web services, and SystemLink services created by NI. The NI Web Server protects web applications against common web security threats, provides high scalability to many enterprise-grade data services, and allows device management.



NOTE

The NI Web Server supports Windows (64-bit) only. The Application Web Server supports both Windows (32-bit and 64-bit) and RT targets.

You can publish a Web service to the NI Web Server through a stand-alone application, package, or package installer. You can establish secure communication between Web clients and LabVIEW Web services by enabling Secure Sockets Layer (SSL) encryption on the NI Web Server or assigning different privileges to each user role.

The default response format to a Web client changes from an XML to a JSON string. If you want to create user interfaces to visualize data and interact with Web services through a web browser, integrate WebVIs into Web services.

For more information about WebVIs, refer to the LabVIEW NXG Web Module Manual.

Application Builder Enhancements

LabVIEW 2020 introduced enhancements to the LabVIEW Application Builder and build specifications.

(Windows) Repairing Packages

If an installation does not run correctly, you can repair the package in NI Package Manager. In LabVIEW, you can configure a custom repair action to execute when NI Package Manager repairs the package. To create a custom repair action, on the **Advanced** page of the **Package Properties** dialog box, right-click in the **Custom actions** table and select **Add Repair Action**. You can choose from the following options:

- **Pre-repair**—Specifies the action to execute before repairing the current package.
- **Post-repair**—Specifies the action to execute after repairing the current package.

- **Post-repair all**—Specifies the action to execute after repairing all packages.

Miscellaneous Application Builder Enhancements

The **Feed** page of the **Package Properties** dialog box includes the following new options:

- **Include all dependencies in feed**—Specifies whether to add all dependencies of a package or only the package to the feed.
- **Append package version to feed location**—Specifies whether to include the package version in the feed. LabVIEW appends a subfolder with the package version to the feed location.
- **Include all dependencies in SystemLink feed**—Specifies whether to publish all dependencies of a package or only the package to the SystemLink feed.

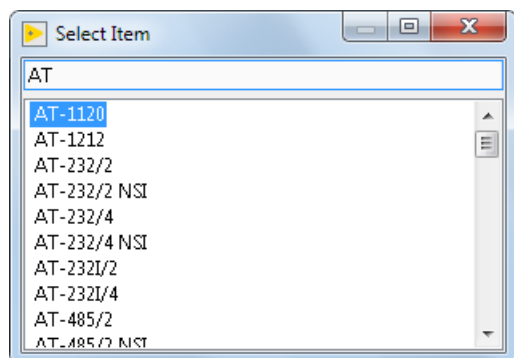
These options are available for Windows only.

Environment Enhancements

Learn about the enhancements to the LabVIEW 2020 environment.

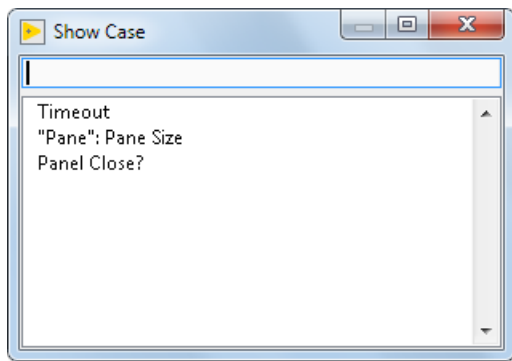
Improvements to Selecting a List Item

In LabVIEW 2020, you can change the value of a list item in a ring control, enum, or I/O control or constant by using the **Select Item** dialog box. Right-click the control or constant on the front panel or constant on the block diagram and select **Select Item**. This dialog box replaces the **Select Item** shortcut menu item.



Improvements to Moving between Subdiagrams

In LabVIEW 2020, you can change the visible frame of a Case, Event, or Conditional Disable structure by using the **Show Case** dialog box. Right-click on the structure border and select **Show Case**. This dialog box replaces the **Show Case** shortcut menu item.

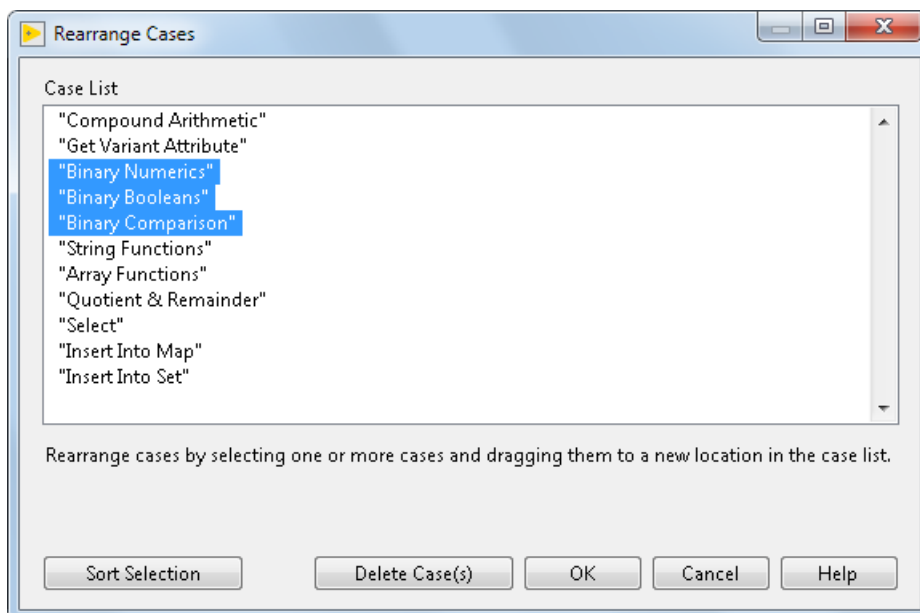


Improvements to Rearranging Cases or Subdiagrams



The **Rearrange Cases** dialog box has been redesigned and includes the following new functionalities:

- Rearrange multiple cases
- Delete one or more cases
- Sort only selected cases within the case list
- Resize the dialog box



[Idea submitted by NI Discussion Forums member Intaris.]

Clearing Data for Variant Data Types

 To clear data for a variant data type, right-click the variant control or indicator and select **Data Operations»Clear Data**.

[Idea submitted by NI Discussion Forums member altenbach.]

Improvements to Displaying Errors

LabVIEW 2020 changes the way a VI or library reports errors when the VI or library is error-free but has a broken dependency. In LabVIEW 2019 and earlier, the **Error list** window displays an error about the immediate dependency of the VI or library. Double-click the error to open the immediate dependency. In LabVIEW 2020, the **Error list** window displays the error: **ROOT CAUSE: Dependency is broken**. Double-click the error to open the broken dependency that causes the error.

Launching the Cluster Size Dialog Box

 In addition to right-clicking, you can double-click the **Array to Cluster** function to open the **Cluster Size** dialog box.

[Idea submitted by NI Discussion Forums member PalanivelThiruvankadam.]

Displaying the Last Item in a Ring Control

You can hide the last item in a ring control so that users cannot select the item while the VI runs. To show or hide the last item, right-click the ring control and select **Show/Hide Last Item When Running**. You can also use the **Last Value Hidden** property to show or hide the last item in a ring control programmatically.

Enhancements to the Import Shared Library Wizard

On the **Configure VIs and Controls** page of the Import Shared Library wizard, use the following new options to apply settings to multiple parameters and functions:

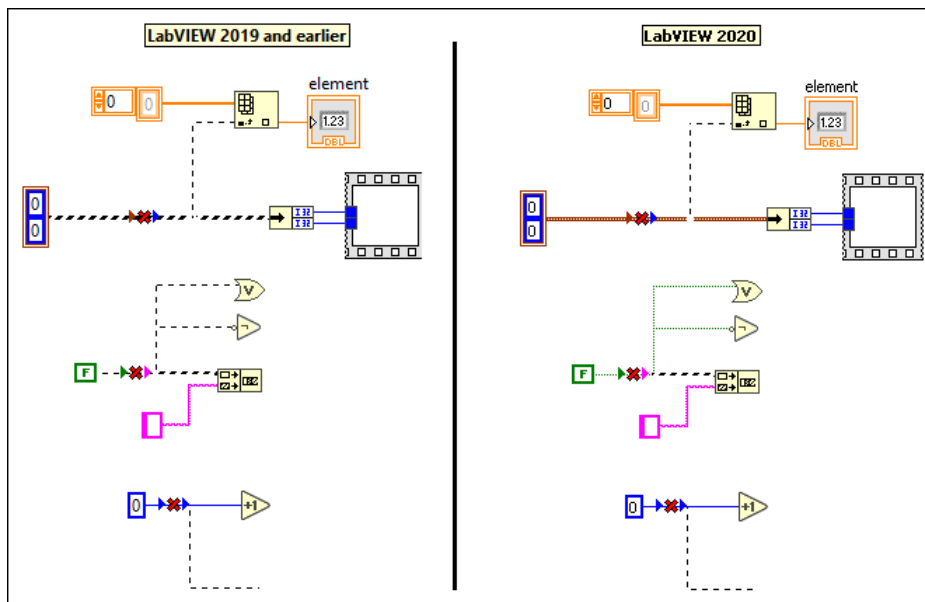
- **Apply to All Matching Parameters**—Applies the type settings of the current parameter to all function parameters that match the declaration in the header file.
- **Apply to All**—Applies the calling convention of the current function to all functions in the shared library.

Block Diagram Enhancements

Learn about block diagram enhancements in LabVIEW 2020.

Cleaning Up Broken Wire Branches

In previous versions of LabVIEW, a broken wire branch causes the entire wire to display as broken and removing the wire branch removes the entire wire. In LabVIEW 2020, only broken wire branches display as broken. When you remove broken wires by selecting **Edit»Remove Broken Wire** or pressing <Ctrl-B>, LabVIEW removes only broken wire branches. LabVIEW removes the entire wire if all branches are broken.



[Idea submitted by NI Discussion Forums member altenbach.]

Hiding the Event Data Node

In an Event Structure, you can show or hide the Event Data Node for a case when all data items are unused. Right-click the structure border and select **Visible Items»Event Data Node for This Case** to show or hide the Event Data Node.

[Idea submitted by NI Discussion Forums member Broken_Arrow.]

Hiding the Iteration Terminal in Loops

In For Loops and While Loops, you can choose to show or hide the iteration terminal. Right-click the loop border and select **Iteration Terminal** to show or hide the terminal. You cannot hide the iteration terminal if the terminal is wired.

[Idea submitted by NI Discussion Forums member somebody_that_i_used_to_k.]

Swapping Wire Positions When Only One Input is Wired

 In previous versions of LabVIEW, you can swap wire positions on a function that has two inputs and both inputs are wired. In LabVIEW 2020, you can swap wire positions on the function when only one input is wired. Press the <Ctrl> key while clicking the wired input to swap wire positions.

[Idea submitted by NI Discussion Forums member altenbach and tst.]

Setting Text on Icons

 In LabVIEW 2020, you can quickly create an icon that contains text by using the **Quick Drop** keyboard shortcut <Ctrl-K>. When you press <Ctrl-Space> and then <Ctrl-K>, LabVIEW adds the filename of the VI as text in the icon. LabVIEW truncates the text if it is too long.

You also can right-click the VI icon and select **Set Icon to VI Name**.

You can customize the text by pressing <Ctrl-Space>, typing the text you want to appear in the icon, and then pressing <Ctrl-K>.

[Idea submitted by NI Discussion Forums member tst.]

Displaying Context Help for Objects in Quick Drop

 The **Context Help** window displays information about the object you select in the **Quick Drop** dialog box.

[Idea submitted by NI Discussion Forums member elset191.]

Improvements to Wiring Terminals for Objects Inserted from Quick Drop

 Objects you add on the selected wire using **Quick Drop** have improved default wiring. When you use the **Quick Drop** keyboard shortcut <Ctrl-I>, LabVIEW wires the object to the most appropriate terminals. The following functions have improved wiring:

- Binary numeric functions, such as Add, Subtract, Multiple, and Divide
- Binary Boolean functions, such as And, Or, Exclusive Or, Not And, Not Or, Not Exclusive Or, and Implies
- Binary Comparison functions, such as Equal?, Not Equal?, Greater?, Less?, Greater Or Equal?, Less Or Equal?
- Compound Arithmetic
- Get Variant Attribute
- String functions, such as Match Pattern, Search and Replace String, and Search/Split String
- Array functions, such as Delete From Array, Insert Into Array
- Quotient & Remainder

- Select
- Insert Into Map
- Insert Into Set

[Idea submitted by NI Discussion Forums member D.]*

VIs and Functions

New VIs and Functions

LabVIEW 2020 introduces new VIs for web services, websockets, Transport Layer Security (TLS), and more.

Web Services VIs

The **Web Services** palette is reorganized and includes the following new subpalettes:

- **Application Web Server**—Contains VIs that specifically support Web services deployed to the Application Web Server, including VIs to configure Embedded Server Pages (ESP) scripting, encrypt and decrypt data transfers, and manage HTTP sessions on the Application Web Server.
- **NI Web Server**—Contains the new **Get Auth Details for NI Web Server** VI, which returns the authentication details for the NI Web Server.

WebSockets VIs

The **Data Communication** palette includes a link to install **WebSockets** VIs for streaming data to or from web user interfaces. On the **Data Communication** palette, click **Install WebSockets Add-on** to install the WebSockets Toolkit from the JKI VI Package Manager (VIPM). The **WebSockets** VIs appear on the **Data Communication»WebSockets** palette.

[Special thanks to Sam Sharp, author of the WebSockets add-on.]

Multiple Errors VIs

The **Dialog & User Interface** palette includes the new **Multiple Errors** subpalette. Use the **Multiple Errors** VIs to convert an error cluster into different formats or to manipulate the attributes of an error cluster.

Transport Layer Security Functions

The TCP palette includes the new Transport Layer Security (TLS) palette. The TLS protocol presents an interface similar to the Transmission Control Protocol (TCP) and is encrypted and authenticated on top of TCP. LabVIEW TLS support reuses the TCP functions. LabVIEW supports TLS version 1.2. TLS functions do not support FPGA targets.

Miscellaneous New VIs and Functions

LabVIEW 2020 includes the following miscellaneous new VIs and functions:

- The following VIs can compute message digest algorithms:
 - The new File Checksum VI on the Advanced File palette computes the message digest on the contents of a file.
 - The new Byte Array Checksum VI on the Data Manipulation palette computes the message digest on a byte array.

LabVIEW can compute the following message digest algorithms:

- SHA-256
- SHA-224
- SHA-512
- SHA-384
- SHA-512/256
- SHA-512/224
- SHA3-224
- SHA3-256
- SHA3-384
- SHA3-512
- The new Get Memory Status VI on the Memory Control palette returns the amount of physical memory that your LabVIEW process uses. This VI also returns the status of system-wide memory usage.
- The new Create NI GUID VI on the Additional String palette generates a globally unique identifier (GUID) string.
- The new Range Limits for Type VI on the Numeric palette returns the maximum and minimum values of the input data type.
- The new Enum to Arrays of Enum VI on the Conversion palette creates an array of enums with the same number of elements as values in the enum.
- The new Get LabVIEW Class Parent And Member VI Information VI on the Data Type Parsing palette retrieves parent and member VI information of the LabVIEW class or interface stored in a variant.

Changed VIs and Functions

Changed VIs and functions in LabVIEW 2020.

- The default data type of the **name or relative path** input of the Build Path function is changed from string to path. When you create a control or constant from this input, the data type is a path. This input still also accepts a string.
- The To More Specific Class and To More Generic Class functions add support for interfaces. Use these functions to cast a wire of one class or interface type to another class or interface type.
- The Assert Same or Descendant Type VI adds support for interfaces. This VI breaks the calling VI unless the input data is the same as or is a descendant of the input class or interface.
- The Call Parent Method node is renamed Call Parent Class Method. Use this node to call the nearest ancestor implementation of a class method.
- The TDMS File Viewer VI and **TDMS File Viewer** dialog box are redesigned to be more intuitive. You can display data from a .tdms file and change visualization settings all in a single page instead of switching between multiple tabs.
- In previous versions of LabVIEW, the Merge Errors function always returns the first error or warning in the input error clusters. In LabVIEW 2020, you can configure the function to return an error cluster that combines all of the input errors and warnings by right-clicking the function and selecting **Retain All Errors**.

New and Changed Properties, Methods, and Events

LabVIEW 2020 includes the following new and changed properties, methods, and events.

New Properties, Methods, and Events

LabVIEW 2020 includes the following new properties, methods, and events:

- The LVClassLibrary class includes the following new properties and methods:
 - Parent Interfaces
 - Parent Libraries
 - Is Interface
 - Write Parent Interfaces
 - Write Parent Library Paths
 - Add Parent Interface
 - Remove Parent Interface
- The Application class includes the following new properties and method:
 - Palettes:Controls Names

- Palettes:Functions Names
 - LabVIEW Class:Create Interface
- The VI class includes the following new property and method:
 - Execution:Retain Wire Values
 - Control VI Apply Changes
- The GObject class includes the following new properties:
 - Grouped
 - Group Member Refs[]
 - Locked
- The Array class and Array Constant class include the following new property:
 - Index Rect
- The Path class includes the following new property:
 - Browse Button Rect
- The Ring class includes the following new properties:
 - Last Value Hidden
 - Flavor
- The TreeControl class includes the following new property:
 - Focus Item
- The MulticolumnListbox class includes the following new properties:
 - Selected Cell
 - Bulk Cells:Font Colors
 - Bulk Cells:Background Colors
- The Project class includes the following new method:
 - Item From Item ID
- The Numeric class, NumericConstant class, and OverridableParameterTerminal class include the new properties for configuring fixed-point behavior for numeric controls and constants and fixed-point output configuration on math primitives.
- The Ring class, Enum class, and ComboBox class include the following new events:
 - Operate Menu Activation
 - Operate Menu Dismissed
- The Control class includes the following new event:
 - Shortcut Menu Dismissed

Changed Methods

LabVIEW 2020 includes the following changed methods:

- The **Consider Parent Classes** parameter of the LVClassLibrary:Has Implementation? method is renamed **Consider Parents**. This method returns an error if the class inherits multiple methods of the same name from parent interfaces and does not provide its own override.
- The Library:Source Scope:Set and Propagate method includes the new **skip prompt?** parameter. Use this parameter to specify whether to ask before applying the access scope setting to all override VIs when the item is a dynamic dispatch VI.

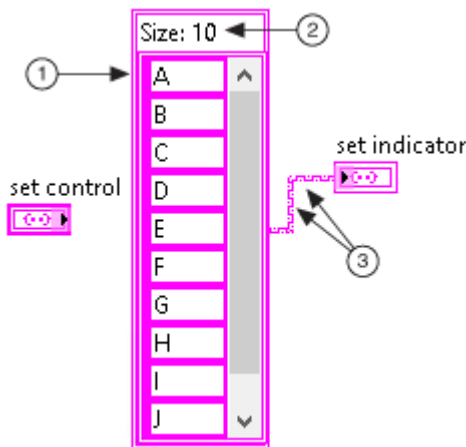
LabVIEW 2019 Features and Changes

Released May 2019

New Fundamental Collection Types: Set and Map

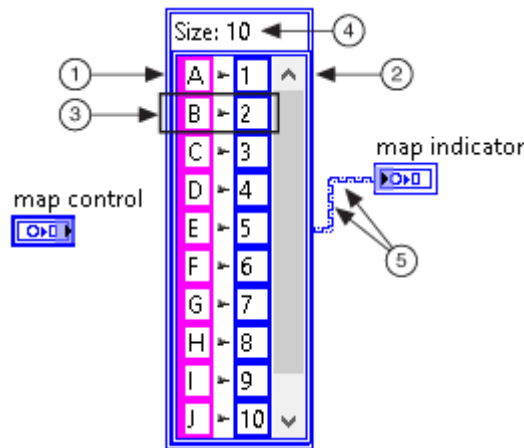
LabVIEW 2019 introduces the following *collection* data types for aggregating collections of homogeneous data: *set* and *map*. Both types maintain unique elements or entries in sorted order, which allows faster search, insertion, modification, and removal operations on the data than unordered data structures, such as arrays, even when the data size is large.

Set —A collection of unique elements of the same data type.



①	Element
②	Size—Number of elements in the set
③	Wires of set—Wire color matches the data type color of the element

Map—A collection of entries consisting of key-value pairs where all the keys are unique. The key and value can each be any data type. Maps are also known as dictionaries because the key is used to look up a value.



①	Key
②	Value
③	Entry—A pair of associated key and value
④	Size—Number of entries in the map
⑤	Wires of map—Wire color matches the data type color of the value

Use the set and map VIs and functions, located on the **Programming»Collection** palette, and the set and map controls and indicators, located on the **Data Containers** palette, to create or manipulate sets and maps.



NOTE

You cannot modify the embedded data in a set or map interactively or programmatically. You can update set or map data only as a whole through operations such as writing to the set or map front panel terminal or right-clicking the set or map and selecting **Data Operations»Copy Data** or **Data Operations»Paste Data**.

Refer to the following VIs and project for examples of manipulating data using sets or maps:

- labview\examples\Collections\Set Collection - Word Counting.vi
- labview\examples\Collections\Map Collection - Comparing Test Results.vi
- labview\examples\Collections\Map Collection - Word Counting.vi

- labview\examples\Design Patterns\Registration Map\Registration Map Usage.lvproj

New Shortcut Menu Items

LabVIEW 2019 includes new shortcut menu items for faster access to common commands on front panel and block diagram objects.

Create Constant, Create Control, and Create Indicator —Creates a constant, control, or indicator from a terminal. These shortcut menu items have long been available and are now duplicated at the top of the shortcut menu.

 **Replace Conversion** —Converts from one numeric conversion function to another numeric conversion function when you right-click a numeric conversion function and select **Replace Conversion**. *[Idea submitted by NI Discussion Forums member EthanStern.]*

 **Create Color Box** —Creates a color box constant, control, or indicator from a color terminal when you right-click a color terminal and select **Create Constant, Create Control, or Create Indicator**. *[Idea submitted by NI Discussion Forums member altenbach.]*

 **Create Cluster from Selection** —Creates a cluster from a selection of constants, controls, or indicators when you right-click a selection of constants, controls, and indicators and select **Create»Cluster from Selection**. *[Idea submitted by NI Discussion Forums member okubik.]*

 **Create Scalar Constant or Control** —Creates a scalar constant or control from an array terminal when you right-click an array terminal and select **Create»Scalar»Constant** or **Create»Scalar»Control**. *[Idea submitted by NI Discussion Forums member altenbach.]*

Replace Comparison —Converts from one comparison function to another comparison function when you right-click a comparison function and select **Replace Comparison**.

Front Panel Enhancements

LabVIEW 2019 introduces new set and map controls and indicators, additional visual styles, and more enhancements for the front panel.

Set and Map Controls and Indicators

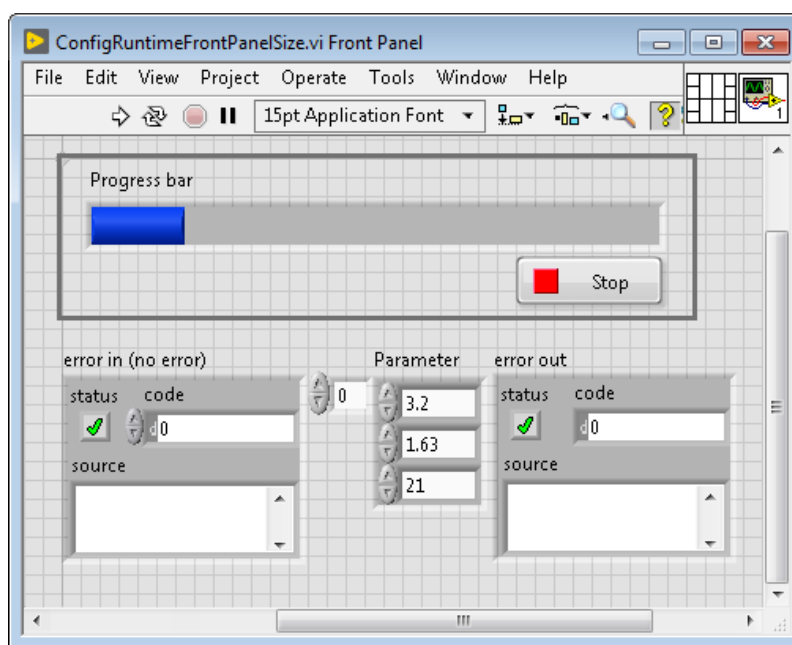
The **Array, Matrix & Cluster** palette is renamed **Data Containers**. The palette now includes the new map and set controls and indicators.

More Controls and Indicators in NXG Style

LabVIEW 2019 includes additional controls and indicators on the **NXG Style** palette for you to easily create numeric, Boolean, container, and I/O name controls and indicators with the same visual style as LabVIEW NXG.

Viewing Run-Time Front Panel Bounds in Edit Mode

You can configure a VI to remember the run-time front panel size and scroll position. When the VI is in edit mode, LabVIEW reflects the settings as a rectangular border on the front panel. When the VI is running, the rectangular border vanishes. In the following figure, the rectangular border indicates the position and size of the visible portion of the front panel when the VI runs.



Refer to the *Viewing Run-Time Front Panel Bounds in Edit Mode* topic for instructions about how to view run-time front panel bounds.



NOTE

LabVIEW shows run-time front panel bounds for single-pane front panels only.

[Idea submitted by NI Discussion Forums member altenbach.]

Application Builder Enhancements

LabVIEW 2019 includes enhancements to the LabVIEW Application Builder and build specifications that enable you to distribute your LabVIEW application through the NI Package Manager.

(Windows) Distributing Your Application as Package Installers

You can distribute your applications to clients through NI Package Manager by creating a package installer (.exe). The *package installer* includes all the package dependencies so that your clients can install the package without network access.

To create a package installer, right-click **Build Specifications**, select **New»Package**, and select **Create a Package Installer** on the **Package Installer** page of the **Package Properties** dialog box.

Publishing NI Packages to Feeds

You can publish NI packages to a feed, so that your clients can subscribe to the feed to receive update notifications and install the NI packages via network access.

A *feed* is a collection of package files that includes a manifest containing information on the included packages. NI recommends using feeds when distributing multiple NI packages that have dependencies on one another. With feeds, you can create a single location where you host and maintain all NI packages intended for end user access.

In LabVIEW 2019, you can use the following options on the **Feed** page of the **Package Properties** dialog box to add your NI package to a local feed or publish the NI package to a SystemLink feed.

- **Add to feed**—Adds the NI package to a local feed. Your clients use NI Package Manager to subscribe to the feed to install the NI package.
- **Publish to SystemLink feed**—Publishes the NI package you created in LabVIEW to a feed on the SystemLink server. Your client can find and install the NI package in SystemLink.

Miscellaneous Application Builder Enhancements

The **Package Attribute** page of the **Package Properties** dialog box is renamed to **Package**. The **Package** page includes the following new options:

- **Include the NI Certificates package**—Specifies to include the NI Certificates package, ni-certificates, as a recommended dependency when clients install your application in NI Package Manager.
- **Display in the runtime deployment packages list**—Specifies to display the package in the **Related packages** list on the **Dependencies** page when you select **Runtime deployment packages only** for **Filter options**.
- **Display in NI Package Manager when filtered by products**—Specifies to display this package when you filter packages by selecting **Products only** in NI Package Manager. Enable this option if the package you create is a product that you want to include in the **Browse Products** section of the NI Package Manager.

Replacing Project Items

In LabVIEW 2019, the **Replace with a packed library** shortcut menu is renamed **Replace with**. You can replace a VI, class, or library in a LabVIEW project with another item of the

same type. Right-click a project item in the **Project Explorer** window and select **Replace with** to select a file to replace the original item.

Highlighting the Execution of a Section of Code

LabVIEW 2019 allows you to highlight the execution of a section of code instead of the whole block diagram of a VI.

You can set a point where you want to toggle execution highlighting by right-clicking a wire and selecting **Custom Probe»Toggle Execution Highlighting**. When the VI is running, this probe toggles the execution highlighting at the specified point.



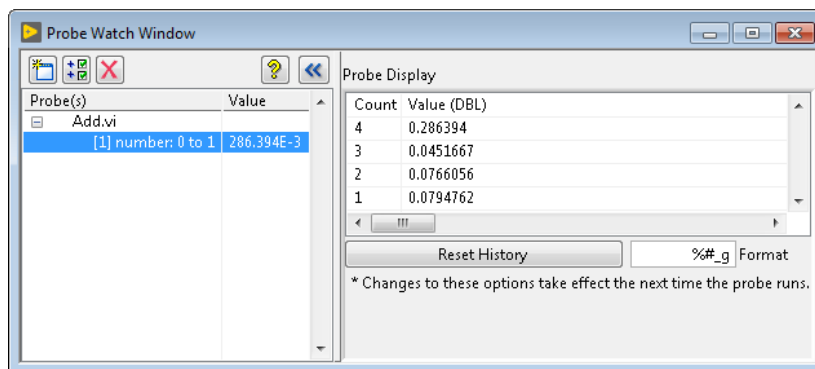
NOTE

The **Toggle Execution Highlighting** custom probe is not supported on RT targets.

[Idea submitted by NI Discussion Forums member therealkilkenny.]

Monitoring the Flow of Data Using History Probes

In LabVIEW 2019, you can display a history of the data that flowed through a wire by placing a history probe on the wire.



You can use history probes with the following data types:

- Boolean
- String
- Path
- I8
- I16
- I32
- I64
- U8

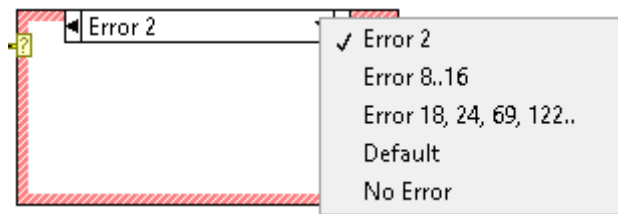
- U16
- U32
- U64
- SGL
- DBL
- EXT
- CSG
- CDB
- CXT

To place a history probe on a wire, right-click the wire and select **Custom Probe** and select the history probe option for your data type.

Enhancements to Handling Errors in Case Structures

Quickly configure a case structure to execute a specific subdiagram for specific errors.

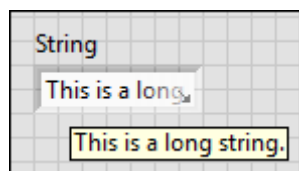
 In LabVIEW 2019, you can configure a case structure to execute a specific subdiagram for specific errors or list of errors by entering error value(s) or ranges in the selector labels using the same syntax as entering numeric values or ranges.



[Idea submitted by NI Discussion Forums member Hueter.]

Indicating Text Overflow in Constants, Controls, and Indicators

 LabVIEW 2019 indicates that visible text is cut off in strings, numerics, timestamps, text rings and enums, and combo boxes by a text fade-out effect with an arrow. Hover over the arrow to display the entire text in a tip strip.



LabVIEW 2019 enables text overflow by default. To disable text overflow, right-click a control, indicator, or constant and deselect **Visible Items»Text Overflow**.

[Idea submitted by NI Discussion Forums member Lavezza.]

Environment Enhancements

LabVIEW 2019 includes the enhancements to the LabVIEW environment to help you work more efficiently.

-  When you double-click a dynamic dispatch subVI node, the **Choose Implementation** dialog box includes the new **Diagram Preview** section that displays a preview of the block diagram of the selected VI. *[Idea submitted by NI Discussion Forums member PrimaryKey.]*
- (Windows) When you build an installer in LabVIEW, the default name of the executable is changed from `setup.exe` to `install.exe`. This behavior change may have potential impact to post-build processes that rely on the `setup.exe` name. To specify a different name for the executable, use the **Installer name** option on the **Product Information** page of the **Installer Properties** dialog box.
- When you select the Error Ring drop-down arrow, the **Select Error** dialog box includes the new **Filter** option that allows you to search by keyword within the specified error code range. Use this option to narrow down the list of error codes to display in the **Error Listing** table.
-  You can set the position of the left and top coordinates of an object on the front panel or block diagram on the **Appearance** page of the **Properties** dialog box. *[Idea submitted by NI Discussion Forums member smmarlow.]*
- Arrange the objects and placement of the front panel and block diagram by using the **Quick Drop** keyboard shortcut <Ctrl-F>. When you press <Ctrl-Space> and then <Ctrl-F>, LabVIEW performs the following clean-up actions:



NOTE

You can also arrange objects on the front panel and block diagram by pressing <Ctrl-U>.

- On the front panel, arranges controls and indicators to be consistent with the connector pane arrangement, and resizes and moves the front panel to a consistent top-left location on the primary monitor.
- On the block diagram, scrolls the block diagram to a reasonable location relative to the top-most and/or left-most block diagram object, and resizes and moves the block diagram to a consistent top-left location on the primary monitor.

VIs and Functions

New VIs and Functions

LabVIEW 2019 introduces new VIs and functions to support map and set collections and more.

Map and Set VIs and Functions

The **Programming** palette includes the new **Collection** subpalette, which includes the following map and set VIs and functions:

- **Map** VIs and Functions—Creates and manipulates maps.
 - Build Map
 - Collection Size
 - Convert Map To Array
 - Empty Collection?
 - In Place Map Access
 - Insert Into Map
 - Look In Map
 - Map Constant
 - Read Map Max & Min Keys
 - Remove From Map
 - **Registration Map** VIs
 - Registration Map: Confirm Registration
 - Registration Map: Register
 - Registration Map: Unregister
- **Set** VIs and Functions—Creates and manipulates sets.
 - Build Set
 - Collection Size
 - Convert Set To Array
 - Empty Collection?
 - Insert Into Set
 - Element of Set?
 - Read Set Max & Min
 - Remove From Set
 - Set Cartesian Product
 - Set Constant

- Set Difference
- Set Intersection
- Set Symmetric Difference
- Set Union

JSONtext VIs

The **Flatten/Unflatten String** palette includes a link to install JSONtext VIs for programming JSON in LabVIEW. On the **Flatten/Unflatten String** palette, click **Install JSONtext Add-on** to install the JSONtext add-on from the JKI VI Package Manager (VIPM). The JSONtext VIs appear on the **Addons»JSONtext** palette.

[Special thanks to Dr. James David Powell, author of the JSONtext add-on.]

Data Type Parsing VIs

The **Data Type Parsing** palette includes the following new VIs:

- **Get Map Collection Information**—Retrieves map information from the data type stored in the input variant data.
- **Get Set Collection Information**—Retrieves set information from the data type stored in the input variant data.
- **Is Error Cluster**—Indicates whether the data type stored in the input variant is an error cluster.

Advanced File VIs

The **Advanced File** palette includes the following new VIs:

- **Create Directory Recursive**—Creates a directory and any parent directories it requires to exist.
- **Create File and Containing Folders**—Creates a file at a specified path and any folders within that path that do not exist.

Numeric VIs and Functions

The **Numeric** palette includes the following new constant and VI:

-  **Not A Number**—Returns the value NaN to the block diagram. *[Idea submitted by NI Discussion Forums member altenbach.]*
- **Random Number (Range)**—Generates a random value from a specified range. You can use this VI with the following data types: U64, I64, and DBL.

Additional String VIs and Functions

The **Additional String** palette includes the following new VIs:

- **1D String Array to Delimited String**—Converts the elements of a one-dimensional string array to a single string with input array elements separated by a delimiting character.
- **Delimited String to 1D String Array**—Converts substrings in a delimited string to the elements of a one-dimensional string array.

Miscellaneous New VIs and Functions

LabVIEW 2019 includes the following miscellaneous new VIs and functions:

- The **Array** palette includes the new **Remove Duplicates From 1D Array** VI. This VI removes duplicate elements from a 1D array. This VI preserves the original order of the elements.
- The **Assert Type** palette includes the new **Assert Structural Type Mismatch** function. This function breaks the calling VI if the input type is the same data type as any of the specified mismatch inputs, ignoring type definitions and type names. Use the **Assert Structural Type Mismatch** function in conjunction with the **Type Specialization** structure to customize sections of code in a malleable VI (.vim) for specific data types or to force a malleable VI to decline specific data types.
- The **Protocols** palette includes the new **Wait for Configured Network** VI. Use this VI to wait until the system can contact the remote host.
- The **Synchronization** palette includes the new **Synchronize Data Flow** VI. This VI passes through the values of the input wires after the upstream code executes. Use this VI to synchronize multiple parallel code paths at a single point of data flow to ensure a certain execution order.
- The **Application Control** palette includes the new **Get Command Line Arguments** VI. This VI returns the user-defined arguments passed from the command line when LabVIEW or a LabVIEW-built application launched. User-defined arguments start after two hyphens (--) surrounded by spaces in the command line.

Changed VIs and Functions

LabVIEW 2019 includes the following changed VIs and functions:

- The **Python Node**, located on the **Connectivity»Python** palette, supports a new data type—Boolean. You also can use the **Python Node** to marshal numeric arrays to NumPy arrays.

New Properties

LabVIEW 2019 includes the following new properties.

- The VI class includes the new `Block Diagram Window:Alignment Grid Size` and `Front Panel Window:Alignment Grid Size` properties. Use these properties to read or write the alignment grid size on the block diagram or front panel of a VI. You must enable VI Scripting to use these properties.

Upgrading Your Version of LabVIEW

If you have large LabVIEW applications that control or monitor critical operations; cannot afford extended downtime; use multiple modules, toolkits, or drivers; or are saved in an unsupported version of LabVIEW, use this rigorous upgrade process to detect and correct upgrade difficulties as efficiently as possible.

Refer to the NI website at ni.com/info and enter the Info Code lifecycle for information about which versions of LabVIEW still receive mainstream support.

Back Up Your VIs and Machine Configuration

Protect a copy of your VIs and the configuration of your development or production machine before upgrading.

Back Up Your VIs

If you back up your VIs before you upgrade LabVIEW, you can quickly revert to the backup copy. Without the backup copy, you can no longer open upgraded VIs in the previous version of LabVIEW without saving each VI for the previous version.

You can back up a set of VIs by submitting VIs to source code control. This action allows you to revert to this version of the VIs if you cannot address behavior changes caused by upgrading the VIs.

Back Up Your Machine Configuration

Installing a new version of LabVIEW updates shared files in ways that sometimes affect the behavior of VIs even in previous versions. However, after you update those shared files, it is very difficult to restore the previous versions of the files. Therefore, consider one of the following methods for backing up the configuration of NI software on your development machine, especially if you are upgrading from an unsupported version of LabVIEW or if downtime for your applications would be costly:

- Create a backup image of the machine configuration—Use *disk imaging software* to preserve the disk state of the machine before you upgrade, including installed

software, user settings, and files. To return the machine to its original configuration after you upgrade, deploy the backup disk image.

- Test the upgrade process on a test machine—Use a test machine, commonly a virtual machine, to test the upgrade process. Although upgrading on a test machine requires more time than creating a backup image, NI strongly recommends this approach if you need to prevent or minimize downtime for machines that control or monitor production. After resolving any issues that result from upgrading on the test machine, you can either replace the production machine with the test machine or replicate the upgrade process on the production machine.



TIP

To minimize the possibility that upgraded VIs on the test machine behave differently than on the development machine, use a test machine that matches the features of the development machine as closely as possible, including CPU, RAM, operating system, and versions of software.

Test and Record the Existing Behavior of Your VIs

When you upgrade VIs, differences between versions of LabVIEW can occasionally change the behavior of the VIs. Run these tests in both versions of LabVIEW and compare the results to detect behavior changes specifically caused by upgrading.

About this task

Therefore, verify that you have current results for any of the following tests:

Procedure

- Mass compile logs that identify the existence of broken VIs

Mass compiling your VIs before you upgrade is particularly useful if multiple people contribute to the development of the VIs or if you suspect that some of the VIs have not been compiled recently. To generate this mass compile log, place a checkmark in the **Log Results** checkbox of the **Mass Compile** dialog box.

- Unit tests that verify whether individual VIs perform their intended functions correctly
- Integration tests that verify whether a project or group of subVIs work together as expected
- Deployment tests that verify whether VIs behave as expected when deployed to a target, such as a desktop or FPGA target
- Performance tests that benchmark CPU usage, memory usage, and code execution speed

You can use the **Profile Performance and Memory** window to obtain estimates of the average execution speeds of the VIs.

- Stress tests that verify whether the VIs handle unexpected data correctly

What to do next

If you changed any VIs as the result of testing, back up the new versions of the VIs before proceeding.

Install LabVIEW, Add-Ons, and Device Drivers

When you upgrade to a new version of LabVIEW, you must install not only the new development system but also modules, toolkits, and drivers that are compatible with the new version. However, if you have previously installed a driver that is LabVIEW version-independent, then that driver does not need to be upgraded when LabVIEW is upgraded. Refer to the Driver Status Page for further information.

Procedure

1. Install LabVIEW, including modules, toolkits, and drivers.
2. Copy files from the `labview\user.lib` directory in the previous version to the `labview\user.lib` directory in your new version of LabVIEW.

`user.lib` contains custom controls and VIs you created in the previous version of LabVIEW.

3. If you used the JKI VI Package Manager (VIPM) to install VI packages in the previous version of LabVIEW, go to the VIPM software and reinstall all packages to LabVIEW.



RELATED INFORMATION

[Installing LabVIEW](#) on page 120

You can install LabVIEW on the Windows or Linux operating system.

Convert Your VIs

Mass compile your VIs in your new version of LabVIEW. You can generate an error log to help you identify VIs that are broken as a result of changes between versions.

Procedure

1. Check out your VIs from source code control.

Source code control backs up your VIs and tracks changes. If you cannot address behavior changes caused by upgrading the VIs, you can revert to a previous version of the VIs.

Include only VIs that you want to convert to the new version of LabVIEW. After upgrading the VIs, you no longer can open the VIs in a previous version of LabVIEW without selecting **File»Save for Previous Version** for each VI or project.

2. Mass compile the VIs.

Mass compiling VIs simultaneously converts and saves the VIs in your new version of LabVIEW.

3. To help identify any problems that arose from upgrading, create a mass compile log by placing a checkmark in the **Log Results** checkbox of the **Mass Compile** dialog box.

4. When you mass compile VIs that contain FPGA or real-time resources, the **Mass Compile** dialog box may report the VIs as non-executable VIs. To check for errors, you must open the VIs under the FPGA or RT target in a LabVIEW project with the required FPGA or real-time resources.

Differences between LabVIEW versions can occasionally cause some VIs to break if they use modified features. To quickly identify and fix broken VIs, complete the following steps:

5. To identify VIs that broke during upgrading, compare the mass compile error log you created in the previous step to the log you created when testing the existing behavior of the VIs.
6. To determine whether updates to LabVIEW caused each VI to break, refer to [Upgrade and Compatibility Issues](#).

Identify and Correct Behavior Changes

The VI Analyzer Upgrade Tests and the upgrade documentation are tools to help you identify whether the improvements and bug fixes in your new version of LabVIEW change the behavior of your VIs.

About this task

NI invests significant effort to avoid changing the behavior of VIs between different versions of LabVIEW. Sometimes improvements and bug fixes occasionally do alter the behavior of VIs. These techniques help you to quickly identify whether the new version of LabVIEW changes the behavior of your VIs.

Procedure

For large sets of VIs, the VI Analyzer Upgrade Tests provide an efficient way to identify many behavior changes caused by upgrading.

1. Download the VI Analyzer Upgrade Tests for all versions of LabVIEW later than your previous version.
2. Refer to the NI website at ni.com/info and enter the Info Code `analyzevi` to download these tests.
3. Open and run the tests by selecting **Tools»VI Analyzer»Analyze VIs** and starting a new VI Analyzer task.

To analyze an entire project at once, select this menu option from the **Project Explorer** window rather than from a single VI. Resolve test failures by referring to [Upgrade and Compatibility Issues](#) for the version of LabVIEW that corresponds to the tests.

To confirm your VIs perform as well as they did in your previous version of LabVIEW, run existing unit tests, integration tests, deployment tests, performance tests, and stress tests in your new version of LabVIEW.

4. Compare the results from your new version of LabVIEW to the results from your previous version. If you identify new behaviors, refer to the upgrade documentation to diagnose the source of the change.

What to do next

Read the upgrade documentation.

- **Upgrade and Compatibility Issues**—Lists changes that may break or affect the behavior of your VIs. Refer to the subsections for each version of LabVIEW beginning with your previous version.



TIP

To quickly locate deprecated objects and other objects mentioned in the *Upgrade and Compatibility Issues* topic, open your upgraded VIs and select **Edit»Find and Replace**.

- **LabVIEW Release Notes**—Lists known issues, bugs fixed since the last release, and other notes about this release.
- **Module and toolkit documentation**—Lists upgrade issues specific to some modules and toolkits, such as the LabVIEW FPGA Module and the LabVIEW Real-Time Module.
- **Driver readme files**—Lists upgrade issues specific to each driver. To view the readme for your Driver, go to ni.com/manuals.



TIP

To determine whether a behavior change resulted from a driver update rather than an update to LabVIEW, test your VIs in the previous version of LabVIEW after installing the new version.

Upgrade and Compatibility Issues

The *Upgrade and Compatibility Issues* pages list specific changes in each version of LabVIEW that may break or alter the behavior of your VIs. Check for possible compatibility issues for each LabVIEW version between your current version and the version that you're upgrading to.

Upgrading to LabVIEW 2024 Q3

Compatibility issues you might encounter when upgrading to LabVIEW 2024 Q3.

Dynamic Dispatch VIs Are Not Renamed When Building Build Specifications

Beginning with LabVIEW 2024 Q3, LabVIEW does not apply prefixes to the names of dynamic dispatch VIs even if you enable the **Apply prefix for all contained items** option on the **Source File Settings** page of the build specification **Properties** dialog box. LabVIEW needs to maintain the original names of dynamic dispatch VIs in order to ensure correct operation.

Upgrading to LabVIEW 2024 Q3

Compatibility issues you might encounter when upgrading to LabVIEW 2024 Q3.

Dynamic Dispatch VIs Are Not Renamed When Building Build Specifications

Beginning with LabVIEW 2024 Q3, LabVIEW does not apply prefixes to the names of dynamic dispatch VIs even if you enable the **Apply prefix for all contained items** option on the **Source File Settings** page of the build specification **Properties** dialog box. LabVIEW needs to maintain the original names of dynamic dispatch VIs in order to ensure correct operation.

Upgrading to LabVIEW 2024 Q1

Compatibility issues you might encounter when upgrading to LabVIEW 2024 Q1.

Multiple Variable Editor

Starting with LabVIEW 2024 Q1, Multiple Variable Editor is no longer supported. Alternatively, you can use import and export functionality to configure multiple variables at once.

Distributed System Manager

Starting with LabVIEW 2024 Q1, you now have the option to skip installing Distributed System Manager.



RELATED INFORMATION

[*Importing Shared Variable Configuration from a Spreadsheet*](#)

[*Exporting Shared Variable Configuration to a Spreadsheet*](#)

Upgrading to LabVIEW 2023 Q3

Compatibility issues you might encounter when upgrading to LabVIEW 2023 Q3.

Changed Keyboard Shortcuts

- **Ctrl--** and **Ctrl++** was used to modify the font size of any selected text. This is now changed to **Ctrl-<** and **Ctrl->**
- **Ctrl** and mouse wheel were used to navigate between frames of structures. This is now changed to **Ctrl+Shift** and mouse wheel.

Upgrading to LabVIEW 2023 Q1

Compatibility issues you might encounter when upgrading to LabVIEW 2023 Q1.

Application: Version Year Property Deprecated

Beginning with LabVIEW 2023 Q1, **Application: Version Year** property is deprecated.

LabVIEW No Longer Supports MathScript

Beginning with LabVIEW 2023 Q1, LabVIEW no longer supports MathScript Node. Refer to the support document [Migrating to LabVIEW MATLAB Functions from MathScript Nodes](#) for more information.

LabVIEW No Longer Supports Control Design and Simulation Module

Beginning with LabVIEW 2023 Q1, LabVIEW no longer supports the LabVIEW Control Design and Simulation Module.

LabVIEW No Longer Supports GPIB on macOS

Beginning with LabVIEW 2023 Q1, LabVIEW no longer supports GPIB on macOS.

Renamed NXG Style Palette and Controls

In LabVIEW 2023 Q1, all controls previously called **NXG Style** are now named **Fuse Design System**. Refer to [Fuse Design System](#) for more information.

Upgrading to LabVIEW 2022 Q3

Compatibility issues you might encounter upgrading to LabVIEW 2022 Q3.

Web Publishing Tool

The web publishing tool no longer offers the embedded option, since it requires a plug-in that is not supported by any modern web browsers.

Upgrading to LabVIEW 2021

Compatibility issues you might encounter upgrading to LabVIEW 2021.

Python Support

LabVIEW 2021 supports Python 3.6 through 3.9 and removes support for Python 2.7.

Run When Opened

When a user launches a VI that is configured to run when opened from a disk location other than standard NI installer locations, such as the LabVIEW 20XX or National Instruments\Shared folder, the user will be prompted to choose whether to run the VI or open it in edit mode.

nicurl Uses System Certificates on Windows

nicurl uses the OS certificate store on Windows rather than its own HTTPS certificate bundle.

OS Support

LabVIEW no longer supports Windows Vista/XP/7/8.x, Windows Server 2003/2008/2012, or any 32-bit Windows operating system.

Upgrading to LabVIEW 2020

Compatibility issues you might encounter upgrading to LabVIEW 2020.

SSL Encryption Is Deprecated For Remote Front Panels

In LabVIEW 2020, Secure Sockets Layer (SSL) encryption is no longer supported when you place a checkmark in the **SSL** checkbox in the **Remote Panel Server** section on the **Web Server** page of the **Options** dialog box. LabVIEW returns an error when you attempt to make a secure connection to a remote front panel server.

Deprecated VIs, Functions, and Nodes

LabVIEW 2020 and later do not support the MD5Checksum File VI. Use the File Checksum VI instead.

Name Change to the Call Parent Method Node

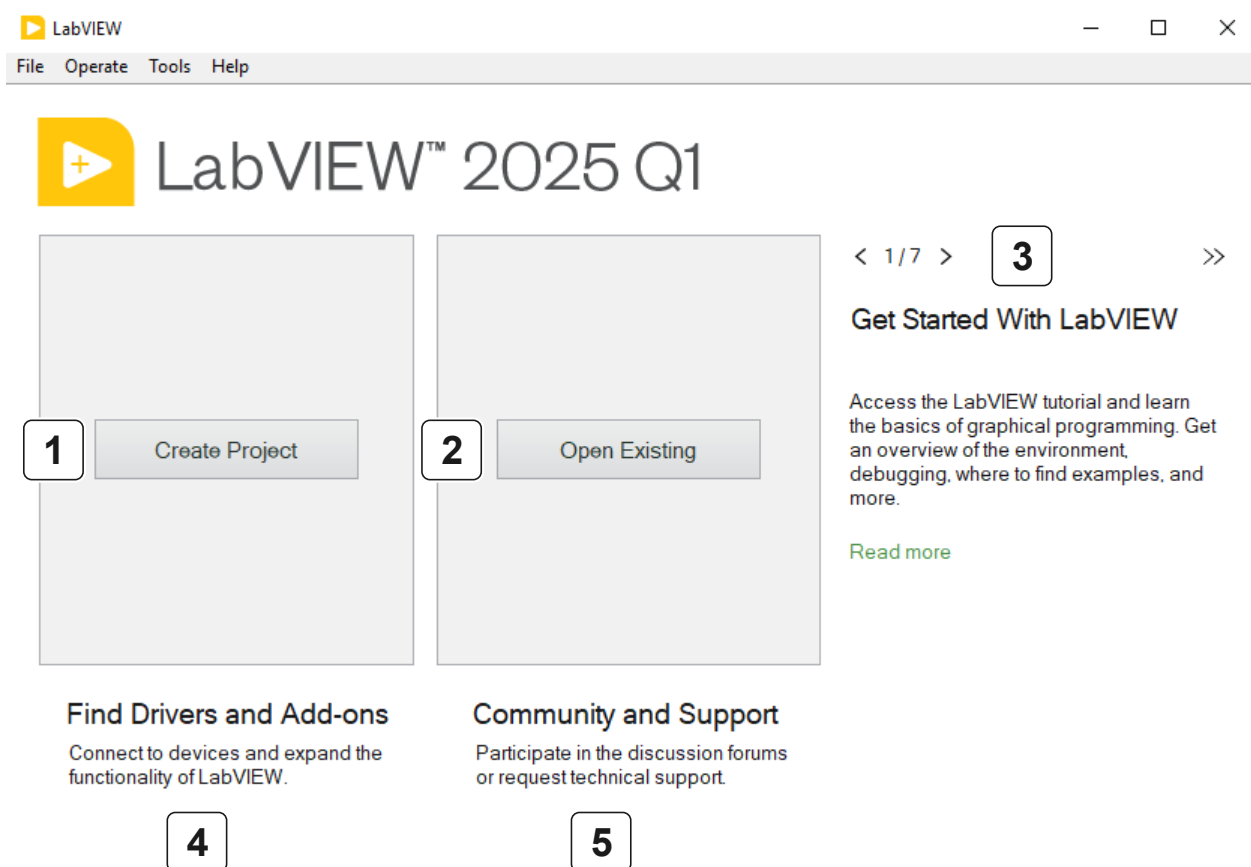
The Call Parent Method node is renamed Call Parent Class Method.

Finding Your Way in LabVIEW

Learn the parts of the LabVIEW environment and how to navigate through them.

Use the **Getting Started** window to create new projects from templates and sample projects or to open existing LabVIEW files as soon as you open LabVIEW. You can access resources to expand the capability of LabVIEW and information to help you learn about LabVIEW.

Figure 1 : LabVIEW Getting Started Window



1 —Create a new project from a template or sample project.

2 —Open an existing project or VI. Try pinning your favorite files so those files are always accessible from the file list.

3 —Browse news and resources from NI.

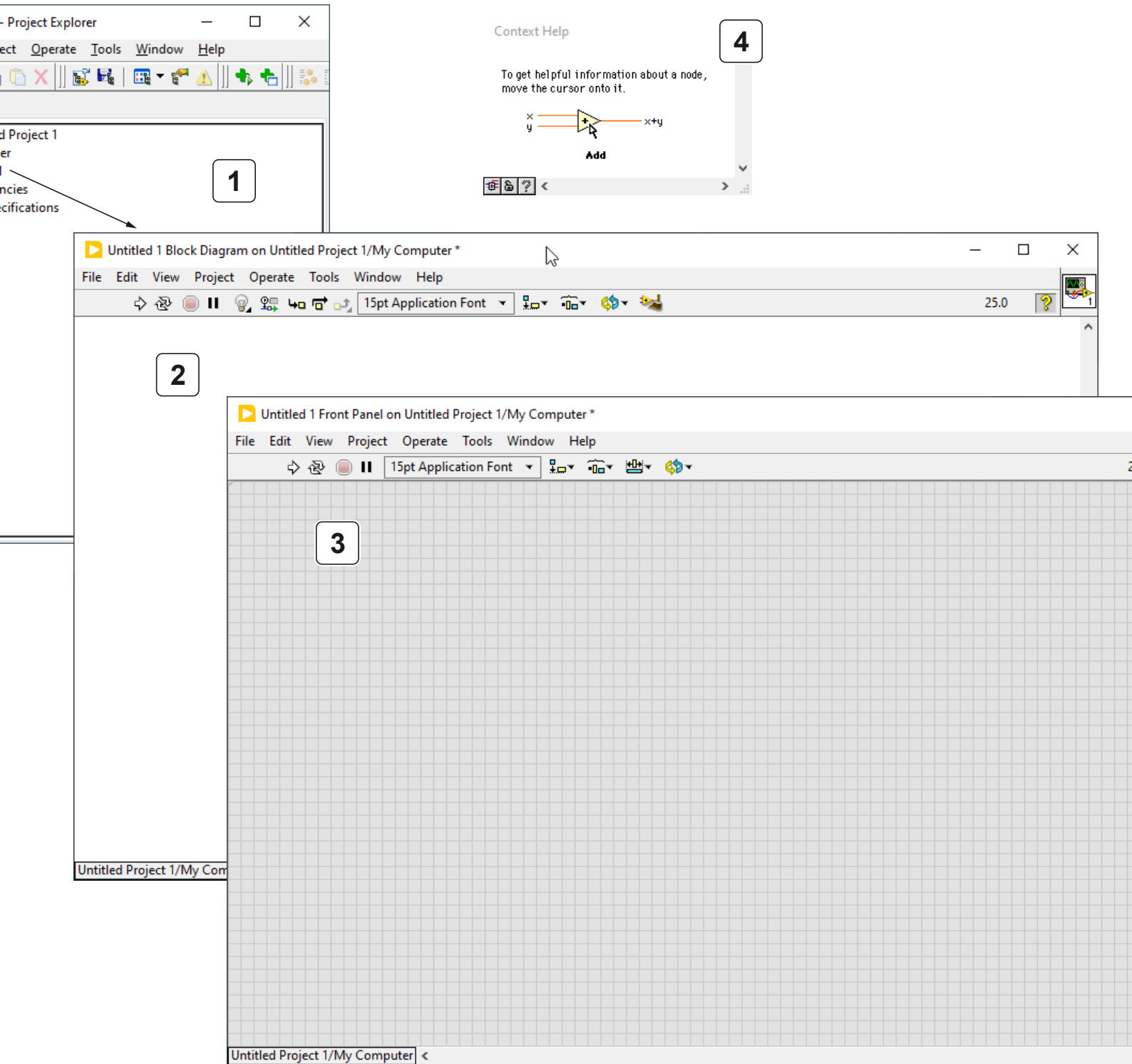
4 —Find Drivers and Add-ons

5 —Community and Support

The **Getting Started** window disappears when you open an existing file or create a new file, and reappears when you close all open front panels and block diagrams. You can display the window by selecting **View»Getting Started Window**.

LabVIEW Editor

Let's explore the components of the LabVIEW editor when we create a blank project with a new VI.



1 —Project Explorer

2 — Block Diagram

3 —Front Panel

4 — Context Help

LabVIEW Environment Overview

Use the LabVIEW palettes, tools, and menus to build the front panels and block diagrams of VIs. LabVIEW includes three palettes: the Controls palette, the Functions palette, and the Tools palette. LabVIEW also includes the Getting Started window, the Context Help window, the Project Explorer window, and the Navigation window. You can customize the Controls and Functions palettes, and you can set several work environment options.



RELATED INFORMATION

- [Menus and Toolbars](#) on page 99
- [Configuring Your LabVIEW User Name](#) on page 804
- [Keyboard Shortcuts](#) on page 101
- [Selecting a Control](#) on page 806
- [Navigating Large Front Panels and Block Diagrams](#) on page 909

Navigating the Controls and Functions Palettes

Click an object on the palette to place the object on the cursor so you can add it to the front panel or block diagram. If you click the thumbtack in the upper left corner of a palette to pin the palette, you also can right-click a VI icon on the palette and select **Open VI** from the shortcut menu to open the VI.

Click the black arrows on the left side of the Controls or Functions palette to expand or collapse palette categories. These arrows appear only if you [set the palette format](#) to **Category (Standard)** or **Category (Icons and Text)**.

Use the following buttons on the Controls and Functions palette toolbars to navigate the palettes, to configure the palettes, and to search for controls, VIs, and functions.

	Up to Owning Palette —Takes you up one level in the palette hierarchy. Click this button and hold the mouse button down to display a shortcut menu that lists each subpalette in the path to the current subpalette. Select a subpalette name in the shortcut menu to navigate to the subpalette. This button appears only if you set the palette format to Icons , Icons and Text , or Text .
	Search —Changes the palette to search mode so you can perform text-based searches to locate controls, VIs, or functions on the palettes. While a palette is in search mode, click the Return button to exit search mode and return to the palette.
	Customize —Provides options for selecting a format for the current palette, showing and hiding categories for all palettes, and sorting items in the Text and Tree formats alphabetically. Select Options from the shortcut menu to display the Controls/Functions Palettes page of the Options dialog box, in which you can select a format for all palettes. This button appears only if you click the thumbtack in the upper left corner of a palette to pin the palette.



Restore Palette Size —Resizes the palette to its default size. This button appears only if you resize the Controls or Functions palette and you click the thumbtack in the upper left corner of a palette to pin the palette.

Menus and Toolbars

Use the menu and toolbar items to operate and modify objects in LabVIEW.

Menus

The menus at the top of a VI window contain items common to other applications, such as **Open**, **Save**, **Copy**, and **Paste**, and other items specific to LabVIEW. Some menu items also list [keyboard shortcuts](#).

(macOS) The menus appear at the top of the screen.



NOTE

Some menu items are unavailable while a VI runs.

Shortcut Menus

All LabVIEW objects have associated shortcut menus, also known as context menus, pop-up menus, and right-click menus. As you create a VI, use the shortcut menu items to change the appearance or behavior of front panel and block diagram objects. To access the shortcut menu, right-click the object.

(macOS) Press <Control>-click to perform the same action as right-click.

While you are editing a VI, the shortcut menu of a control or indicator contains several [common options](#). The menu items above the first two lines in the shortcut menu of a control or indicator are common to all controls and indicators. The menu items below the second line are customized for the object. You can add custom items to the shortcut menu by [creating shortcut menu plug-ins](#).

To perform a common operation on multiple objects simultaneously, select the objects and use the shortcut menu. Some objects may not support all the operations available on the shortcut menu. For example, if you select a numeric control and a structure and select **Create»Reference** from the shortcut menu, LabVIEW creates a reference for only the numeric control.

You also can [customize the shortcut menu items](#) that appear in the VI menu at run time.

Customizing Keyboard Shortcuts

You can change and customize the default LabVIEW keyboard shortcuts. Select **Tools»Options** to display the Options dialog box. Select Menu Shortcuts from the

Category list to set keyboard shortcuts for VI menu items. The **Menu Shortcuts** page also lists the default LabVIEW shortcuts for menu items.



Note LabVIEW does not respond to keyboard shortcuts for dimmed menu items or hidden VI toolbar buttons.

VI Toolbar

Use the buttons on the VI toolbar to run VIs, pause VIs, abort VIs, *debug* VIs, configure fonts, and align, group, and distribute objects.

Project Explorer Window Toolbars

Use the buttons on the Standard, Project, Build, and Source Control toolbars to perform operations in a LabVIEW project. The toolbars are available at the top of the *Project Explorer* window. You might need to expand the Project Explorer window to view all of the toolbars.



Note The **Unit Test Framework** toolbar is available only if you install the LabVIEW Unit Test Framework Toolkit.

You can show or hide toolbars by selecting **View»Toolbars** and selecting the toolbars you want to show or hide. You also can right-click an open area on the toolbar and select the toolbars you want to show or hide.

Navigation Window

The Navigation window displays an overview of the active front panel in edit mode or the active block diagram. Use the **Navigation** window to navigate large front panels or block diagrams. Click an area of the image in the **Navigation** window to display that area in the front panel or block diagram window. You also can click and drag the image in the **Navigation** window to scroll through the front panel or block diagram. Portions of the front panel or block diagram that are not visible appear dimmed in the **Navigation** window.

Select **View»Navigation Window** to display the Navigation window. (Windows) You also can display the window by pressing the <Ctrl-Shift-N> keys. (macOS) Press the <Command-Shift-N> keys. (Linux) Press the <Alt-Shift-N> keys.

Resize the Navigation window to resize the image it displays. LabVIEW retains the **Navigation** window position and size so when you restart LabVIEW, the window appears in the same position and has the same size.



NOTE

If the front panel contains multiple panes, the Navigation window displays an image of the front panel. However, you cannot click or drag the image in the **Navigation** window to navigate the front panel.

Keyboard Shortcuts

The following table lists keyboard shortcuts in the LabVIEW environment. You also can create custom shortcuts for VI menu items.



NOTE

The <Ctrl> key in these shortcuts corresponds to the (macOS) <Option> or <Command> key or to the (Linux) <Alt> key.

Keyboard Shortcut	Description
Object/Movement	
Shift-click	Selects multiple objects; adds object to current selection.
Arrow keys	Moves selected objects one pixel at a time.
Shift-Arrow keys	Moves selected objects several pixels at a time.
Shift-click (drag)	Moves selected objects in one axis.
Ctrl+K	<i>Reorders objects</i> in a stack of objects by moving the selected object one position higher in the stack.
Ctrl+J	<i>Reorders objects</i> in a stack by moving the selected object one position lower in the stack.
Ctrl+Shift+K	<i>Reorders objects</i> in a stack by moving the selected object to the top of the stack.
Ctrl+Shift+J	<i>Reorders objects</i> in a stack by moving the selected object to the back of the stack.
Ctrl-click (drag)	<i>Duplicates selected object.</i>
Ctrl-Shift-click (drag)	<i>Duplicates selected object</i> and moves it in one axis.
Shift-resize	<i>Resizes selected object</i> while maintaining aspect ratio.
Ctrl-resize	<i>Resizes selected object</i> while maintaining center point.
Ctrl-Shift-resize	<i>Resizes selected object</i> while maintaining center point and aspect ratio.
Ctrl-drag in the direction of open space	Adds <i>more working space</i> to the front panel or block diagram.
Ctrl+Alt-drag in the direction of reducing space	Reduces <i>extra working space</i> from the front panel or block diagram.
Ctrl-A	Selects all front panel or block diagram items.
Ctrl-Shift-A	Performs last <i>alignment</i> operation on objects.

Keyboard Shortcut	Description
Ctrl-Shift-D	Performs last <i>distribution</i> operation on objects.
Double-click open space	Adds a <i>free label</i> to the front panel or block diagram if automatic tool selection is enabled.
Ctrl= (or Ctrl+)	Zooms in on the VI block diagram.
Ctrl--	Zooms out on the VI block diagram.
Ctrl-0	Zooms to actual size.
F2 (or Z)	Toggles between the current zoom level and a view of the full block diagram.
Ctrl-mouse wheel	Zooms on the VI block diagram.
Ctrl-Shift-mouse wheel	 NOTE For versions prior to LabVIEW 2023 Q3, use Ctrl-mouse wheel to perform this action. <i>Scrolls through subdiagrams</i> of a Case, Event, or Stacked Sequence structure.
Spacebar	Switches between different selection modes when <i>creating selection rectangles</i> .
Spacebar (drag)	Disables <i>preset alignment positions</i> when moving object labels or captions.
Ctrl-U	Reroutes all existing wires and rearranges objects on the block diagram automatically.
Navigating the LabVIEW Environment	
Ctrl-F	Finds objects or text.
Ctrl-Shift-F	Displays Search Results window or the Find Project Items dialog box with the previous search results.
Ctrl-G	Searches VIs for next instance of an object or text.
Ctrl-Shift-G	Searches VIs for previous instance of VIs, object, or text.
Ctrl-Tab	Cycles through LabVIEW windows in the order the windows appear onscreen. (Linux) The order of the windows depends on the window manager you use.
Ctrl-Shift-Tab	Cycles the opposite direction through LabVIEW windows.
Ctrl-L	Displays the Error list window.
Ctrl-Shift-E	Displays the current VI in the Project Explorer window.
Ctrl-Shift-W	Displays the All Windows dialog box.

Keyboard Shortcut	Description
Ctrl-Shift-B	Displays the Class Browser window.
Ctrl-Shift-N	Opens the Chat with Nigel AI Advisor menu item.
Navigating Front Panels and Block Diagrams	
Ctrl-E	Displays the block diagram or front panel window.
Ctrl-Space	Displays the Quick Drop dialog box. On Chinese keyboards, press the <Ctrl-Shift-Space> keys. (macOS) Press the <Command-Shift-Space> keys.
Ctrl-#	Enables or disables grid alignment. On French keyboards, press the <Ctrl-#> keys. (macOS) Press the <Command-#> keys.
Ctrl-/	Maximizes and restores window.
Ctrl-T	Tiles front panel and block diagram windows.
Ctrl-Shift-N	Displays the Navigation window.
Ctrl-I	Displays the VI Properties dialog box.
Ctrl-Y	Displays the History window.
Navigating the VI Hierarchy Window	
Ctrl-D	Redraws the VI Hierarchy window.
Ctrl-A	Shows all VIs in the VI Hierarchy window.
Ctrl-click VI	Displays the subVIs and other nodes that make up the VI you select in the VI Hierarchy window.
Enter	<i>Finds the next node</i> that matches the search string after initiating a search by typing in the VI Hierarchy window.
Shift-Enter	<i>Finds the previous node</i> that matches the search string after initiating a search by typing in the VI Hierarchy window.
Debugging	
Ctrl-D	Opens the Debug window.
Ctrl-Down arrow	<i>Steps into node.</i>
Ctrl-Right arrow	<i>Steps over node.</i>
Ctrl-Up arrow	<i>Steps out of node.</i>
File Operations	
Ctrl-N	Opens a new, blank VI.
Ctrl-O	Opens an existing VI.

Keyboard Shortcut	Description
Ctrl-Shift-O	Opens the Quick Open dialog, which displays recently-opened files that you can open as well as all files in any project you have open.
Ctrl-W	Closes the VI.
Ctrl-S	Saves the VI.
Ctrl-Shift-S	Saves all open files.
Ctrl-P	Prints the window.
Ctrl-Q	Quits LabVIEW.

Basic Editing

Ctrl-Z	Undoes last action.
Ctrl-Shift-Z	Redoes last action.
Ctrl-X	Cuts selected objects.
Ctrl-C	Copies selected objects.
Ctrl-V	Pastes last cut or copied objects.

Help

Ctrl-H	Displays the Context Help window. (macOS) Press the <Command-Shift-H> keys.
Ctrl-Shift-L	Locks the Context Help window.
Ctrl-? or <F1>	Displays the LabVIEW Help.

Tools and Palettes

Ctrl	Switches to next most useful tool.
Shift	Switches to Positioning tool.
Ctrl-Shift over open space	Switches to Scrolling tool.
Spacebar	Toggles between two most common tools if <i>automatic tool selection is disabled</i> .
Shift-Tab	Enables automatic tool selection.
Tab	Cycles through four most common tools if you <i>disabled automatic tool selection</i> by clicking the Automatic Tool Selection button. Otherwise, enables automatic tool selection.
Arrow keys	Navigates temporary Controls and Functions palettes.
Enter	Navigates into a temporary palette.

Keyboard Shortcut	Description
Esc	Navigates out of a temporary palette.
Shift-right-click	Displays a temporary version of the Tools palette at the location of the cursor.
SubVIs	
Double-click subVI	<i>Displays the subVI front panel.</i>
Ctrl-double-click subVI	<i>Displays the subVI block diagram and front panel.</i>
Drag VI icon to block diagram	<i>Places that VI</i> as a subVI on the block diagram.
Shift-drag VI icon to block diagram	<i>Places that VI</i> as a subVI on the block diagram with constants wired for controls that have non-default values.
Ctrl-right-click block diagram and select VI from palette	Opens the front panel of that VI.
Execution	
Ctrl-R	<i>Runs the VI.</i>
Ctrl-.	Stops the VI, when used while VI is running.
Ctrl-M	Changes to run or edit mode.
Ctrl-Down arrow	<i>Moves key focus inside</i> an array or cluster, when used while VI is running.
Ctrl-Up arrow	<i>Moves key focus outside</i> an array or cluster, when used while VI is running.
Tab	Navigates the controls or indicators according to <i>tabbing order</i> , when used while VI is running.
Shift-Tab	Navigates backward through the controls or indicators, when used while VI is running.
Wiring	
Ctrl-B	Deletes all <i>broken wires</i> in a VI. If you select a structure or area of the block diagram with broken wires, this shortcut deletes the broken wires in only the selected area.
Esc, right-click, or click terminal	While wiring, cancels a wire you started.
Single-click wire	Selects one segment.
Double-click wire	Selects a branch.
Triple-click wire	Selects the entire wire.

Keyboard Shortcut	Description
A	While wiring, temporarily disables automatic wire routing.
W	While wiring, toggles automatic wire connectivity when an object moving in and out of a structure is connected to an object in the structure.
Double-click	While wiring, tacks down wire without connecting it.
Spacebar	While wiring, switches the direction of a wire between horizontal and vertical.
Spacebar	While moving objects, <i>toggles automatic wiring</i> .
Shift-click	Undoes last point where you set a wire.
Ctrl-click input on function with two inputs	Switches the two input wires.

Text

Double-click	Selects a single word in a string.
Triple-click	Selects an entire string.
Ctrl-A	Selects all text in the current text field.
Ctrl-Right arrow	Moves forward in string by one word when using text that has characters that use single bytes, such as Western character sets. Moves forward in string by one character when using text that has characters that use multiple bytes, such as Asian character sets.
Ctrl-Left arrow	Moves backward in string by one word when using text that has characters that use single bytes, such as Western character sets. Moves backward in string by one character when using text that has characters that use multiple bytes, such as Asian character sets.
Home	Moves to the beginning of the current line in a string.
End	Moves to the end of the current line in a string.
Ctrl-Home	Moves to the beginning of the entire string.
Ctrl-End	Moves to the end of the entire string.
Shift-Enter	Adds new items when entering items in enumerated type controls and constants, ring controls and constants, or Case structures. In string constants, press the <Shift-Enter> keys on the keyboard to disable autosizing if it is enabled. If autosizing is disabled, press the <Shift-Enter> keys on the keyboard to add a new line to the constant and jump to that line.
Ctrl-Shift-Enter	Duplicates the visible subdiagram of a Case structure when editing the text of the case selector label.

Keyboard Shortcut	Description
Esc	Cancels current edit in a string.
Ctrl-Enter	Ends text entry.
Ctrl->  NOTE For versions prior to LabVIEW 2023 Q3, use Ctrl+= to perform this action.	Increases the current font size.
Ctrl-<  NOTE For versions prior to LabVIEW 2023 Q3, use Ctrl-- to perform this action.	Decreases the current font size.
Ctrl-"  NOTE For versions prior to LabVIEW 2023 Q3, use Ctrl-0 to perform this action.	Displays the Font Style dialog box.
Ctrl-1	Changes to the Application font in the Font Style dialog box.
Ctrl-2	Changes to the System font in the Font Style dialog box.
Ctrl-3	Changes to the Dialog font in the Font Style dialog box.
Ctrl-4	Changes to the current font in the Font Style dialog box.
Ctrl-B	Bolds text.
Ctrl-I	Italicizes text.
Ctrl-U	Underlines text.
Arrays	
Ctrl-Delete	Deletes the current element and shifts later elements up.
Ctrl-Shift-Delete	Deletes the element before the current element.
Ctrl-Enter	Inserts a new element after the current element.
Ctrl-Shift-Enter	Inserts a new element before the current element.

Keyboard Shortcut	Description
Ctrl-PgUp	Scrolls the visible elements in an array by one page up.
Ctrl-PgDn	Scrolls the visible elements in an array by one page down.
Ctrl-Home	Scrolls to the first element in an array.
Ctrl-End	Scrolls to the last element in an array.
Tab	Moves to the next element if the Auto Tool is on and locked.
Shift-Tab	Moves to the previous element if the Auto Tool is on and locked.



RELATED INFORMATION

[Debug Window - LabVIEW Programming Reference Manual](#)

Quick Drop Keyboard Shortcuts

The following table lists keyboard shortcuts you can use only while the **Quick Drop** dialog box is active. To display the **Quick Drop** dialog box, press the <Ctrl-Space> keys. On Chinese keyboards, press the <Ctrl-Shift-Space> keys. (macOS) Press the <Command-Shift-Space> keys. (Linux) Press the <Alt-Space> keys.

You also can create custom **Quick Drop** keyboard shortcuts using the **Quick Drop Configuration** dialog box. Click the **Configure** button on the **Quick Drop** dialog box to display the **Quick Drop Configuration** dialog box.



NOTE

To apply a shortcut to a specific block diagram or front panel object, you must select the object to which you want to apply the shortcut before opening the **Quick Drop** dialog box.

Keyboard Shortcut	Description
Ctrl-D	Creates controls and indicators for all unwired inputs and outputs of the selected block diagram object(s).
Ctrl-Shift-D	Creates constants for all unwired inputs of the selected block diagram object(s).
Ctrl-F	On the front panel, arranges controls and indicators to be consistent with the connector pane arrangement, and resizes and moves the front panel to a consistent top left location on the primary monitor. On the block diagram, scrolls the block diagram to a reasonable location relative to the top-most and/or left-most block diagram object, and resizes and moves the block diagram to a consistent top left location on the primary monitor.

Keyboard Shortcut	Description
Ctrl-W	Wires a row or multiple parallel rows of selected block diagram objects.
Ctrl-Shift-W	Wires a row or multiple parallel rows of selected block diagram objects and cleans up the selected objects.
Ctrl-R	Removes the selected block diagram object(s) and any wires and constants connected to the selected object(s), and connects wires of identical data types that were wired to the inputs and outputs of the deleted object(s).
Ctrl-T	Repositions the visible labels and captions of top-level front panel and block diagram objects to match the Default label position you specify in the Options dialog box. You also can select multiple objects on the block diagram, display the Quick Drop dialog box, and press <Ctrl-T> to move the labels of only the selected objects.
Ctrl-Shift-T	Repositions the visible labels and captions of top-level front panel and block diagram objects, including terminals contained in subdiagrams, to match the Default label position you specify in the Options dialog box.
Ctrl-P	Replaces the selected front panel or block diagram object(s) with the object you select in the Quick Drop dialog box.
Ctrl-I	Inserts the object you select in the Quick Drop dialog box on the selected wire(s) on the block diagram.
Ctrl-Shift-I	Inserts a single instance of the object you select in the Quick Drop dialog box on multiple selected wires.
Ctrl-B	Changes the VI Server class of the selected Property Node(s), Invoke Node(s), and/or class specifier constant(s) to the class you enter in the Quick Drop window.
Ctrl-Shift-B	Changes the property or method of the selected Property Node(s) or Invoke Node(s), respectively, to the property or method name you enter in the Quick Drop window.

Securing the LabVIEW Environment

You can create a secure LabVIEW environment that users must log into and out of provided there is a security domain configured on the network. Manage domains using the Domain Account Manager. After you create user accounts, LabVIEW can use the user login information to identify LabVIEW users when communicating with remote VI Servers that have an access control list enforced. An access control list displays the access rights granted to specific users and groups.

You also can create a secure LabVIEW network by setting up VI Server access control lists on each LabVIEW environment. Then you can programmatically log in, log out, and monitor user account changes from the block diagram using application property nodes, method nodes, and application events.

Customizing Your Work Environment

You can customize the Controls and Functions palettes, and you can use the Options dialog box to [select a palette format](#) and set other work environment options.

Customizing the Controls and Functions Palettes

You can customize the Controls and Functions palettes in the following ways:

- [Add VIs and controls to the palettes.](#)
- [Edit a palette set](#) to rearrange the built-in palettes, create and move subpalettes, and so on.
- [Add toolkits to the palettes.](#)
- Add items on the Functions palette to the [Favorites category](#).



CAUTION

Do not save your own VIs or other files in `.lib` files installed with LabVIEW in the `vi.lib` directory because LabVIEW uninstalls the `.lib` files, including any VIs and other files you saved in the `.lib` files, when you uninstall or reinstall. Save your VIs and controls in the `user.lib` directory to add them to the **Functions** and Controls palettes.

Adding VIs and Controls to the User and Instrument Drivers Subpalettes

The simplest method for [adding VIs and controls](#) to the Functions and Controls palettes is to [save them in the labview\user.lib directory](#). When you restart LabVIEW, the User Libraries and User Controls palettes contain subpalettes for each directory, LLB, or palette file `.mnu` in `labview\user.lib` and icons for each file in `labview\user.lib`. After you add files to or remove files from specific directories, LabVIEW automatically updates the palettes when you restart LabVIEW.

The Instrument Drivers palette corresponds to the `labview\instr.lib` directory. Save instrument drivers in this directory to add them to the Functions palette.

When you add VIs or controls to the Functions and Controls palettes using this method, you cannot set the name of each subpalette or the exact location of the VIs or controls on the palettes.

Editing Palette Sets

Use the Edit Controls and Functions Palette Set dialog box to edit a palette set. Select **Tools»Advanced»Edit Palette Set** to display the **Edit Controls and Functions Palette Set** dialog box. LabVIEW stores Controls and Functions palette information in the `labview`

\menus directory. The menus directory contains directories that correspond to each palette category.



NOTE

You can add a line to the preference file that sets the `menusDir` preference to an alternative path, one that is unique for each user preference file.

If you edit the palettes, LabVIEW saves the edits to the `labview\menus` directory. LabVIEW saves the edits in a copy of the original palette. The protection of the original palettes ensures that you can experiment with the palettes without corrupting the original palettes. To revert to the original palettes, click the **Restore to Default** button on the Edit Controls and Functions Palette Set dialog box.

You also can use the Palette Editing VIs to create and edit palette sets programmatically. Use the Palette Editing VIs if you want to edit a large number of palettes, create new palettes, or validate the appearance of a palette after you edit it.

The Read Palette VI extracts palette data from a palette file `.mnu`. The **palette data** output contains all information about the palette from the palette file. Select Controls or Functions from the **palette type** input to read data from the palette file for the Controls or Functions palette. Run the Read Palette VI a second time to read data for the second type of palette. Use the Read Palette VI if you want to use an existing palette as a template. You can read the data from an existing palette and customize the data to create a new palette.

Use the Write Palette VI to write palette data to a palette file. Use the **items** input in the **palette data** cluster to specify the information you want to display on the palette. The order of items in the **items** input defines the order of items on the palette. Select Controls or Functions from the **palette type** input to write data to the palette file for the Controls or Functions palette. Run the Write Palette VI a second time to write data for the second type of palette.

You can edit the palette set for a specific target by specifying the target in the **application reference** input of the Write Palette VI. Create a project with the target you want to reference. Use the `Project:Projects[]` property to obtain the target reference from the project. When you have the target reference, use the Application property to obtain the application reference.



Caution If you use the Palette Editing VIs to edit palettes in the `labview\menus` directory, LabVIEW does not save the edits in a copy of the original palette. After you edit the original palettes, you cannot revert to the original palette.

After you edit the palette set, use the Refresh Palettes method to update the palette set.

Refer to the `labview\examples\Palette API` directory for examples of editing the Controls and Functions palette set programmatically.

Find related examples

How LabVIEW Stores Palettes

The .mnu files and .llb files contain one Controls palette and one Functions palette each. In addition, each file contains an icon for the Controls and Functions palettes. You must store each subpalette you create in a separate .mnu file.

For each VI or control, LabVIEW creates an icon on the palette. For each subdirectory, .mnu file, or .llb file, LabVIEW creates a subpalette on the palette.

Building .NET and ActiveX Subpalettes

If you use .NET or ActiveX controls on the front panel, select **Tools»Import» .NET Controls to Palette** or **ActiveX Controls to Palette** to convert a set of .NET or ActiveX controls, respectively, to custom controls and add them to the Controls palette. LabVIEW saves the controls in the `labview\menus\Controls\DotNet & ActiveX` directory by default because all files in this directory automatically appear in the **.NET & ActiveX** palette.

Representing Toolkits and Modules on the Palettes

After you install a module or toolkit in LabVIEW, a palette that represents that module or toolkit appears in a relevant category on the Functions palette. You can add module and toolkit subpalettes to the User Libraries and Favorites palettes to customize your palette view.

Setting Work Environment Options

Select **Tools»Options** to customize LabVIEW. Use the Options dialog box to set options for front panels, block diagrams, paths, performance and disk issues, the alignment grid, palettes, undo, debugging tools, colors, fonts, printing, the History window, and other LabVIEW features.

Use the **Category** list at the left side of the Options dialog box to select among the different categories of options.

How LabVIEW Stores Options

You do not have to edit options manually or know their exact format because the Options dialog box does it for you. LabVIEW stores options in a different location on each platform. However, the file format on each platform is similar to other configuration files. It begins with a LabVIEW section marker followed by the option name and the value, such as `offscreenUpdates=True`.

Windows

LabVIEW stores options in a `labview.ini` file in the `labview` directory.

If you want to use a different options file, specify the file in the shortcut you use to start LabVIEW. For example, to use an options file on your computer named `lvrc` instead of `labview.ini`, right-click the LabVIEW icon on the desktop and select **Properties**. Click the **Shortcut** tab and type `labview -pref lvrc` in the **Target** text box.

macOS

LabVIEW stores options in the LabVIEW Preferences text file at `~/Library/Preferences/LabVIEW x Preferences` in your home directory, where `x` is the LabVIEW version number.

If you want to use a different options file, copy the LabVIEW Preferences file to the LabVIEW folder and make options changes in the Options dialog box. When you launch LabVIEW, it first looks for an options file in the LabVIEW folder. If it does not find the file there, it looks in the User Preferences folder. If it does not find the file there, it creates a new one in the User Preferences folder. LabVIEW writes all changes you make in the **Options** dialog box to the first LabVIEW Preferences file it finds.

Linux

LabVIEW stores options at `/home/<username>/natinst/.config/LabVIEW-x /labview.conf`, where `x` is the LabVIEW version number. If you change an option in the Options dialog box, LabVIEW writes the change to the `/home/<username>/natinst/.config/LabVIEW-x /labview.conf` file.

If you want to use a different options file, specify the file on the command line when you start LabVIEW. For example, to use a file named `lvrc` in the `test` directory instead of `/home/<username>/natinst/.config/LabVIEW-x /labview.conf`, type `labview -pref /test/lvrc`. LabVIEW writes all changes you make in the Options dialog box to the `lvrc` options file. When you specify an options file on the command line, LabVIEW still reads the `/home/<username>/natinst/.config/LabVIEW-x /labview.conf` file in the program directory, but the options file specified on the command line overrides conflicting entries in the program directory.

Text Characteristics

LabVIEW uses fonts already installed on the computer. Use the **Text Settings** pull-down menu on the toolbar to change the attributes of text.

The **Text Settings** pull-down menu contains the following built-in fonts.

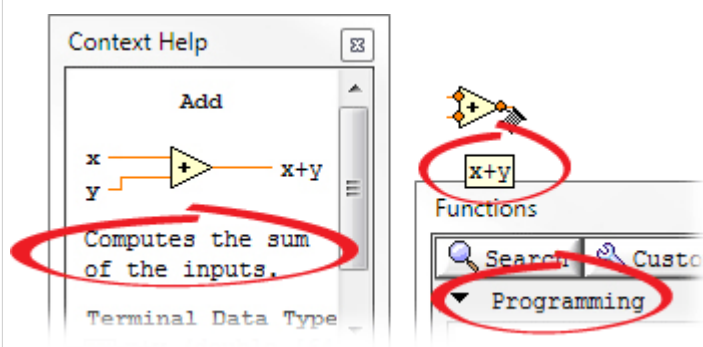


NOTE

The following screenshots do not depict the default fonts, but instead use alternative fonts in order to demonstrate where content is affected by changing the Application, System, and Dialog fonts.

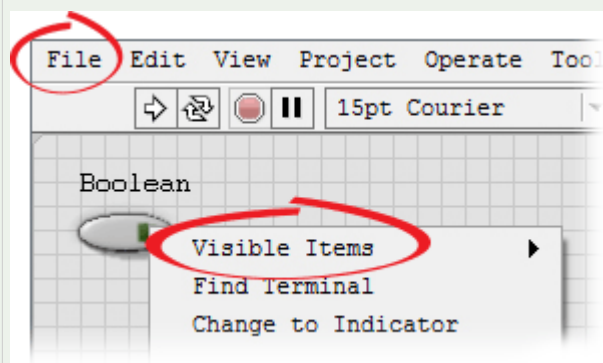
Application Font —Default font used for text in the following locations:

- Controls palettes
- Functions palettes
- context help
- tip strips



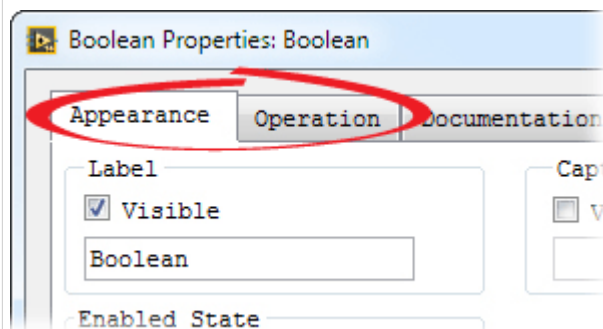
System Font —Default font used for text in the following locations:

- the menu bar
- shortcut menus



Dialog Font —Default font used for text in the following locations:

- dialog boxes
- system controls



If you select objects or text before you make a selection from the **Text Settings** pull-down menu, the changes apply to everything you select. If you select nothing, the changes apply to the default font. Changing the default font does not change the font of existing labels. It affects only those labels you create from that point on.

Select **Font Dialog** from the **Text Settings** pull-down menu on the front panel to apply specific font styles to text you have selected. If you do not select any text, the **Panel Default** option contains a checkmark. If you select **Text Settings»Font Dialog** from the block diagram without selecting any objects, the **Diagram Default** option contains a checkmark. You can set different fonts for the front panel and for the block diagram. For example, you can have a small font on the block diagram and a large one on the front panel.

When you transfer a VI that contains built-in fonts to *another platform*, the fonts correspond as closely as possible.



Tip Right-click the text and select **Size to Text** from the shortcut menu to automatically adjust for font differences between platforms.

The **Text Settings** pull-down menu also has **Size**, **Style**, **Justify**, and **Color** submenu items.

Font selections you make from any of these submenus apply to objects you selected. For example, if you select a new font while you have a knob and a graph selected, the labels, scales, and digital displays all change to the new font.

LabVIEW preserves as many font attributes as possible when you make a change. For example, if you select an object that uses bold Courier font and then select **Arial** from the Text Settings pull-down menu, the object retains the bold font attribute as bold Arial font. If you use the Font Style dialog box to configure selected objects, LabVIEW applies the text characteristics you select to the objects. Text characteristics include Font, Size, Align, Color, and the font attributes. If you select one of the built-in fonts, such as Application Font, System Font, or Dialog Font, to apply to the selected objects, LabVIEW applies the text characteristics of the built-in font to the selected objects.

When you work with objects that have multiple pieces of text, like slides, font changes you make affect the objects or text you currently have selected. For example, if you select the entire slide and select **Style»Bold** from the **Text Settings** pull-down menu, the scale, digital display, and label all change to a bold font. If you select only the label and select **Bold**, only the label changes to a bold font. If you select text from a scale marker and select **Bold**, all the markers change to a bold font.

You should *use standard, consistent fonts* to design a user interface that is easy to use.

Nigel™, AI Advisor

Nigel is a design assistant powered by Microsoft Azure OpenAI. Nigel brings a wealth of knowledge about LabVIEW into your design process to help you create VIs and projects.

Introduced in LabVIEW 2025 Q3

Nigel can help you while you are working in LabVIEW by performing tasks such as:

- Finding VIs and objects in palettes using the search bar
- Finding examples using the search bar to give you a solid starting point for a project
- Providing instructions for completing a specific task
- Explaining key LabVIEW and VI design concepts
- Describing the functionality of existing VIs
- Assisting with finding specific settings or dialog boxes in the LabVIEW environment
- Summarizing documentation topics
- Answering questions about hardware functionality and configuration

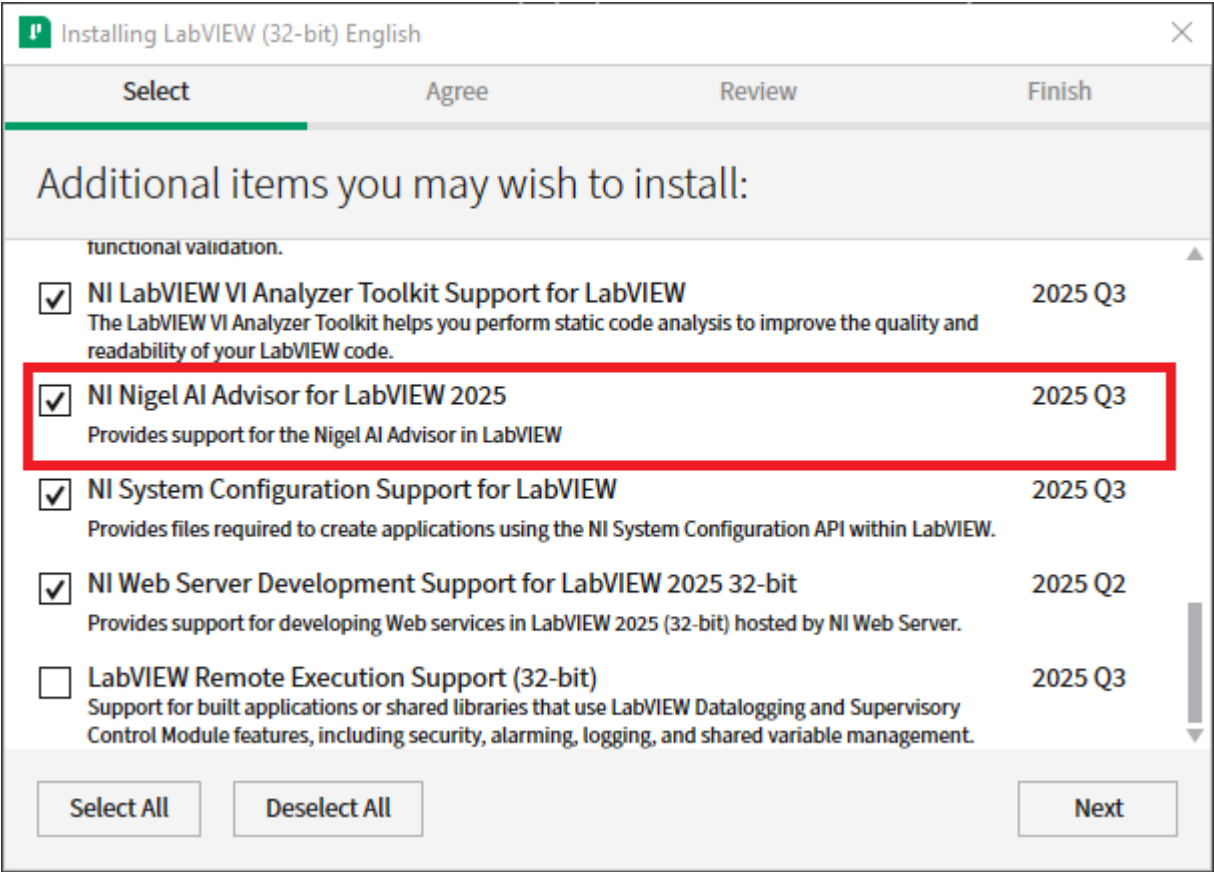


NOTE

Nigel cannot find VIs and objects contained in installed third-party palettes.

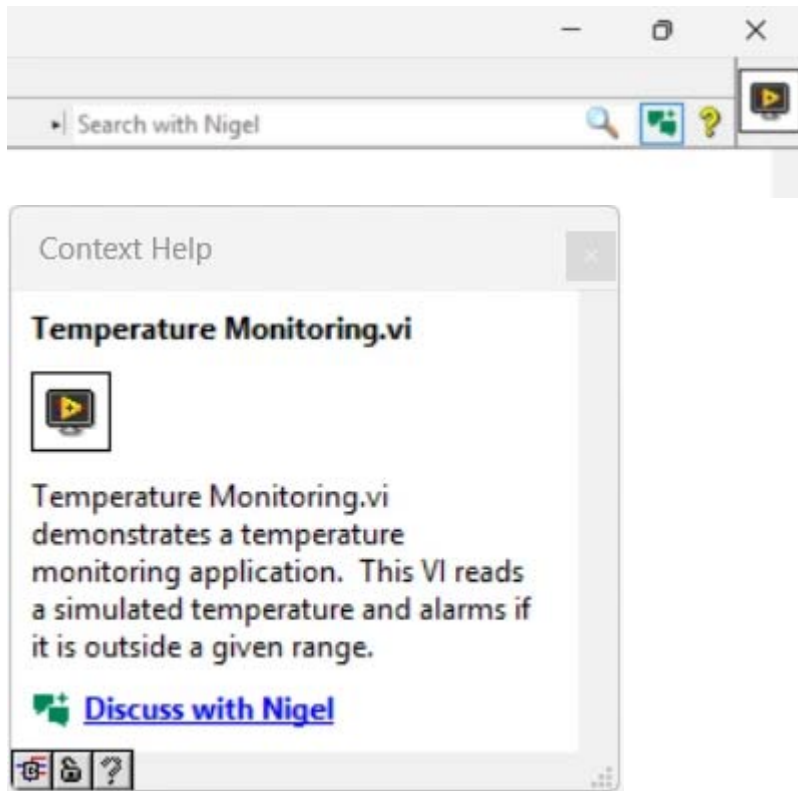
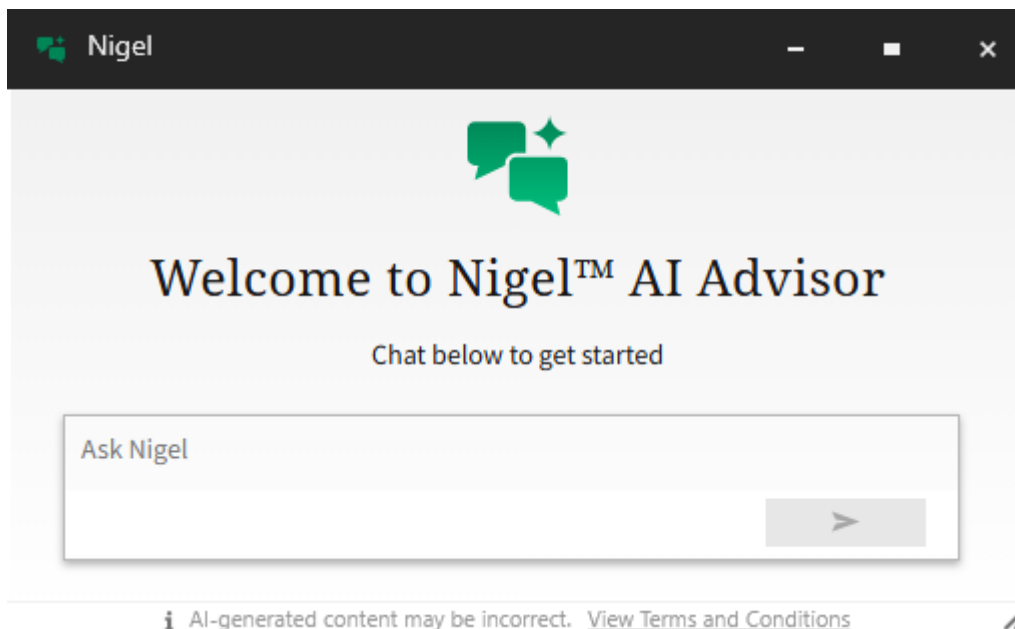
Installing Nigel

Nigel installs by default while installing LabVIEW. If you do not want to install Nigel, you can deselect **NI Nigel AI Advisor for LabVIEW <Version Year>** when setting up your LabVIEW installation.



Accessing Nigel

Nigel is integrated into the LabVIEW IDE, making it easy to open a chat window with Nigel from the menu, toolbar, and Context Help window. You can also use the **Search with Nigel** search bar to generate a drop-down list of AI-powered search results.

Figure 2 : Nigel Access Locations**Figure 3 :** Nigel Chat Window**NOTE**

You can use the keyboard shortcut **Ctrl+Shift+N** to quickly open the **Chat with Nigel AI Advisor** menu item.

After you open Nigel, you must log in to your ni.com user account and ensure it is associated with an active subscription license, multi-seat license, or Software Service Program for LabVIEW Professional.

Rate Limits

Access to Nigel is rate limited per user and is measured as Requests Per Day (RPD). The RPD limit is shared between the TestStand chat and all LabVIEW Nigel search functions.



NOTE

You can only use the **Discuss VI with Nigel** feature 20 times per day.

Disabling Nigel

You can disable Nigel features by editing the following registry keys:

- You can disable the Discuss with Nigel feature by setting the Computer \HKEY_LOCAL_MACHINE\SOFTWARE\National Instruments\Nigel AI Advisor\DisableNigelSharing registry key (DWORD (32-bit), 0 or 1) to 1.
- You can disable all Nigel features, including Discuss with Nigel, chat, and search, by setting the Computer\HKEY_LOCAL_MACHINE\SOFTWARE\National Instruments\Nigel AI Advisor\DisableNigel registry key (DWORD (32-bit), 0 or 1) to 1.



NOTE

Nigel may respond to queries you make in English even if you ask in your local language.



RELATED INFORMATION

[Meet Nigel, NI's New AI Advisor](#)

Examples, VI Templates, Project Templates, and Sample Projects

You can use examples, VI templates, project templates and sample projects to help build VIs and design applications.

Examples

Before you build a new VI, consider searching for an example VI that meets your needs by selecting **Help»Find Examples** to open the NI Example Finder. LabVIEW searches among hundreds of example VIs you can use and incorporate into VIs that you create. You can modify an example to fit an application, or you can copy and paste from one or more

examples into a VI that you create. Browse or search the example VIs with the NI Example Finder by selecting **Help»Find Examples**

You also can access examples using the **Open example** and **Find related examples** buttons located at the bottom of certain VI and function reference topics in the LabVIEW Help. Click the **Open example** button to open the example VI to which the topic refers. Click the **Find related examples** button to open the NI Example Finder and display related example VIs.

You also can right-click a VI or function on the block diagram or on a pinned palette and select **Examples** from the shortcut menu to display a help topic with links to examples for that VI or function.

Refer to ni.com/support for additional example VIs.

LabVIEW VI Templates

If you cannot find an appropriate example VI, open a template VI from the New dialog box and populate the template with built-in VIs and functions from the Functions palette. You also can use the Create Project dialog box, available by selecting **File»Create Project**, to browse customizable templates and sample projects.

The built-in VI templates include the subVIs, functions, structures, and front panel objects you need to get started building common measurement applications. VI templates open as [template name] #.vi, where template name is the name of the template and # is a unique decimal number. Select **File»New** to display the New dialog box, which lists the built-in VI templates.

Application design patterns are available in the **VI»From Template»Frameworks** category under **Design Patterns**. Application design patterns represent LabVIEW architectures and techniques that are solutions to specific problems in software design.

Creating Custom VI Templates

You can create custom template VIs to avoid adding the same components to the front panel or block diagram each time you want to perform a similar operation. Create a custom VI template by building a VI and saving it as a template.

You also can add templates as subVIs to a template VI. To add a template as a subVI, open the VI template and drag the VI icon in the upper right corner of the front panel or block diagram window to the block diagram of the top-level template VI. When you create a VI from a template that contains template subVIs and save the VI, LabVIEW prompts you to save each template subVI as a VI.

LabVIEW Project Templates and Sample Projects

LabVIEW includes several templates that provide starting points for useful design patterns, such as a state machine and a queued message handler. LabVIEW also includes sample projects that demonstrate working applications based on these templates. You can customize these templates and sample projects according to the needs of your application.

Select **File»Create Project** to display the Create Project dialog box, which lists these templates and sample projects.

Other Files

In the **Create New** list of the New dialog box, select an item from the **Other Files** category to create run-time menus, custom controls, global variables, and project libraries.

Installing LabVIEW

You can install LabVIEW on the Windows or Linux operating system.



RELATED INFORMATION

[Upgrading Your Version of LabVIEW](#) on page 87

If you have large LabVIEW applications that control or monitor critical operations; cannot afford extended downtime; use multiple modules, toolkits, or drivers; or are saved in an unsupported version of LabVIEW, use this rigorous upgrade process to detect and correct upgrade difficulties as efficiently as possible.

Installing LabVIEW on Windows

You can install LabVIEW with either NI Package Manager or the LabVIEW Platform media.



NOTE

If Windows Update is enabled, the installation process might hang if Windows Update interferes with the installation of Microsoft Visual C++ 2015 Run-Time. To resolve this issue, visit the NI Support article *NI Installer Hangs While Installing Visual C++ 2015 Run-Time* for more information and steps .



NOTE

To activate LabVIEW, use the serial number included with your software. For more information about activating your software, refer to *License Manager*.

Installing LabVIEW Using Package Manager

Refer to *Package Manager* for information about installing, removing, and upgrading NI software using Package Manager.

Installing LabVIEW Using the LabVIEW Platform Media



NOTE

If you purchased this product with an NI software suite or an NI product bundle, use the installation media that shipped with your purchase to install this product.

To install LabVIEW, modules, toolkits, and drivers, insert the LabVIEW Platform media and follow the onscreen instructions. When prompted, log in to your NI User Account to activate your NI products. If you are managing your LabVIEW license with a volume license server, you should receive your volume license file by email.



RELATED INFORMATION

[Package Manager](#)

[NI Installer Hangs While Installing Visual C++ 2015 Run-Time](#)

[License Manager](#)

Installing LabVIEW on Linux

NI provides a repository add-on for LabVIEW and supported drivers that you can install using the native package manager of your Linux distribution.

Visit the *Install LabVIEW on Linux Operating Systems* page for instructions on installing LabVIEW on Linux operating systems.



RELATED INFORMATION

[Install LabVIEW on Linux Operating Systems](#)

LabVIEW Linux Containers for CI/CD Workflows

LabVIEW supports Linux containers for streamlining CI/CD workflows and other development automation tasks.

NI provides the following from its official account on Docker Hub:

- A base Docker image with a ready-to-use LabVIEW 2025 Q3 installation named `labview:2025q3-linux`.
- A Dockerfile that can be used to build your own image for teams that want to use custom settings, tools, and scripts.

Consult the readme on the LabVIEW Linux Container page on the NI GitHub for installation guides and example use cases.



RELATED INFORMATION

[LabVIEW Linux Container - GitHub](#)

[Docker Hub](#)

LabVIEW Concepts

This section provides information about various features and functionalities in LabVIEW.

LabVIEW Functionalities

Boolean Controls and Indicators

Use Boolean controls and indicators located on the Boolean subpalettes to enter and display Boolean (TRUE/FALSE) values with objects such as buttons, switches, and LED lights.

The following list describes the common use cases of Boolean controls and indicators available on the Modern, Silver, System, or Classic subpalettes:



NOTE

The availability of Boolean controls and indicators may vary depending on the subpalette style you choose. Refer to the Modern, Silver, System, or Classic subpalette for Boolean control and indicator availability.

- **Buttons**— Use as a control to select a command by depressing the button.
- **Rockers, Toggles, and Switches**— Use as a control to select a command by rocking, toggling, or switching the object from one position to the other.
- **LEDs**— Use as an indicator to display that a condition is met. LabVIEW turns the light ON or OFF depending on whether the condition is met.
- **Radio Buttons**— Use as a control to select a command from a list of choices.
- **Checkboxes**— (System palette only) Use as a control to select a TRUE or FALSE state, such as when only one option is possible.
- **Mixed Checkboxes**— (System palette only) Use as a control to select a TRUE, FALSE, or MIXED state, such as when simultaneously setting a property of multiple objects where some are TRUE and FALSE.

Mechanical Actions of Boolean Controls

Use the configurable mechanical action of a Boolean control to create front panels that resemble the behavior of physical instruments such as oscilloscopes and multimeters. Switch and latch actions are similar in that they both change the value of the Boolean control. The actions differ in how the value of the control reverts to the previous value.

- **Switch action**— Use a switch action when you want only the user to revert the value of the control back to the previous value.
- **Latch action**— Use a latch action when you want the VI reading the control to revert the value of the control back to the previous value. The user can also return the

value of the control to the previous value, but only if the user does so before the VI reads the control.



Note Latch actions cannot be used with local variables, and latch actions used with event structures typically have their terminals placed inside the Value Change event frame.

Right-click a Boolean control and select the behavior from the options in the **Mechanical Action** menu. The following table provides more information about the three types of switch and latch behaviors:

Behavior	When to Use	Example
Switch When Pressed	Use as a control that changes the current value as soon as you click the control, and retains the new value until you click the control again.	Light switch—The light turns on as soon as you switch the control, and stays in that state until you switch the control again.
Switch When Released	Use as a control that changes the current value when you release the mouse button, and retains the new value until you click the control again.	Checkbox—The checkbox changes its value only after you release the mouse button, which gives you the ability to reconsider your choice after clicking by moving the cursor away from the checkbox before releasing the mouse button.
Switch Until Released	Use as a control that only changes the current value when you click and hold the mouse button down, and reverts to the previous value when you release the mouse button.	Doorbell—The bell rings as soon as you press the button, and continues to ring until you release the button.
Latch When Pressed	Use as a control that changes the current value as soon as you click the control, and reverts to the previous value after the VI reads the control.	Emergency stop button—The button stops the system as soon as you press the button, but resets as soon as the system reads the change in value, allowing for the button to be pressed again.
Latch When Released	Use as a control that changes the current value when you release the mouse button, and reverts to the previous value after the VI reads the control.	Close button—The button closes the program only after you release the mouse button and the application reads the change in value, which gives you the ability to reconsider your choice after clicking by moving the cursor away from the button before releasing the mouse button.

Behavior	When to Use	Example
Latch Until Released	Use as a control that only changes the current value when you click and hold the mouse button down. Once you release the mouse button, the value will revert to the previous value after the VI reads the control.	Robot movement control—The movement control informs the system of the robot to move the robot while the control is pressed down. Once the control is released, and the system of the robot has read the control, the robot goes back to its previous state of non-movement.



NOTE
(FPGA Module) On FPGA targets, use **Switch When Pressed**, **Switch When Released**, or **Switch Until Released** as a control that changes the current value once a new value is written from the host VI to the FPGA target, and retains the new value until another value is written from the host VI again. Use **Latch When Pressed**, **Latch When Released**, or **Latch Until Released** as a control that only changes the current value when a new value is written from the host VI to the FPGA target, and reverts to the previous value once the new value has been read by the FPGA target.

Container Controls and Indicators

Use the container controls located on the **Containers** and **Classic Containers** palettes to *group controls and indicators* or to *display the front panel of another VI* on the front panel of the current VI. (Windows) You also can use container controls to display *.NET Framework* and *ActiveX* objects on the front panel.

Tab Controls

Use tab controls to overlap front panel controls and indicators in a smaller area. A tab control consists of pages and tabs. Place front panel objects on each page of a tab control and use the tab as the selector for displaying different pages.

Tab controls are useful when you have several front panel objects that are used together or during a specific phase of operation. For example, you might have a VI that requires the user to first configure several settings before a test can start, then allows the user to modify aspects of the test as it progresses, and finally allows the user to display and store only pertinent data.

On the block diagram, the tab control is an *enumerated type control*. Terminals for controls and indicators placed on the tab control appear as any other block diagram terminal.

Subpanel Controls

Use the subpanel control to *display the front panel* of another VI on the front panel of the current VI. For example, you can use a subpanel control to design a user interface that behaves like a wizard. Place the **Back** and **Next** buttons on the front panel of the top-level VI and use a subpanel control to load different front panels for each step of the wizard.

When you place a subpanel control on the front panel, LabVIEW does not create a terminal for the control on the block diagram. Instead, LabVIEW creates an [Invoke Node](#) on the block diagram with the Insert VI method selected. To load a VI in the subpanel control, wire a reference to that VI to the Invoke Node.



NOTE

Because the subpanel control does not have a terminal, you cannot create an array of subpanel controls, and you cannot create a type definition of a subpanel control. You can place a subpanel control in a cluster to group the subpanel control with other controls, but the cluster cannot contain only a subpanel control or controls.

If the front panel of the VI you want to load is open or if you loaded the front panel in another subpanel control on the same front panel, LabVIEW returns an error, and you cannot load the front panel in the subpanel control unless the front panel is [reentrant](#). You also cannot load the front panel of a VI in a remote [application instance](#), and you cannot load front panels recursively.

If you load a VI that is not running, the VI in the subpanel control loads in edit mode.

LabVIEW displays only the visible area of the front panel of the VI you load in the subpanel control. After you stop running the VI that contains the subpanel control, LabVIEW clears the front panel in the subpanel control. You also can use the Remove VI method to unload the VI in the subpanel control.

Refer to the `labview\examples\Controls and Indicators\Containers\Containers.lvproj` for examples of using subpanel controls.

Dragging and Dropping in LabVIEW

To enable users of a VI to drag and drop data between controls while the VI is running, you must configure the controls to allow drag-and-drop behavior. You can enable dragging and dropping between controls in the following ways:

- Between text-based controls: Use the shortcut menu.
- From the operating system to path controls: Use the shortcut menu.
- Between any types of controls: Programmatically configure drag-and-drop events.



RELATED INFORMATION

[Configuring Run-Time Drag-and-Drop Behavior for Text-Based Controls](#) on page 1085

[Configuring Run-Time Drag-and-Drop Behavior for Text-Based Controls](#) on page 1085

[Implementing Custom Drag-and-Drop Behavior](#) on page 1087

Building the Block Diagram

After you build the front panel, you add code using graphical representations of functions to control the front panel objects. The block diagram contains this graphical source code, also known as G code or block diagram code.



RELATED INFORMATION

[Building the Front Panel](#) on page 590

[Block Diagram Objects](#) on page 126

[Functions Overview](#) on page 177

[Using Wires to Link Block Diagram Objects](#) on page 354

[Block Diagram Data Flow](#) on page 358

[Designing the Block Diagram](#) on page 353

Block Diagram Objects

Objects on the block diagram include terminals and nodes. You build block diagrams by connecting the objects with wires. The color and symbol of each terminal indicate the data type of the corresponding control or indicator. Constants are terminals on the block diagram that supply fixed data values to the block diagram.

Block Diagram Terminals

Front panel objects appear as terminals on the block diagram. Double-click a block diagram terminal to highlight the corresponding control or indicator on the front panel.

Terminals are entry and exit ports that exchange information between the front panel and block diagram. Data values you enter into the front panel controls enter the block diagram through the control terminals. During execution, the output data values flow to the indicator terminals, where they exit the block diagram, reenter the front panel, and appear in front panel indicators.

LabVIEW has control and indicator terminals, node terminals, constants, and specialized terminals on structures. You use wires to connect terminals and pass data to other terminals. Right-click a block diagram object and select **Visible Items»Terminals** from the shortcut menu to view the terminals. Right-click the object and select **Visible Items»Terminals** again to hide the terminals. This shortcut menu item is not available for expandable VIs and functions.

You can configure front panel controls or indicators to appear as icon or data type terminals on the block diagram. By default, front panel objects appear as icon terminals. For example, a knob icon terminal, shown as follows, represents a knob control on the front panel.



The DBL at the bottom of the terminal represents a data type of double-precision, floating-point numeric. A DBL terminal, shown as follows, represents a double-precision, floating-point numeric control.



Right-click a terminal and remove the checkmark next to the **View As Icon** shortcut menu item to display the data type for the terminal. Use icon terminals to display the types of front panel objects on the block diagram, in addition to the data types of the front panel objects. Use data type terminals to conserve space on the block diagram.



NOTE

Icon terminals are larger than data type terminals, so you might unintentionally obscure other block diagram objects when you convert a data type terminal to an icon terminal.

Control terminals have a thicker border than indicator terminals. Also, arrows indicate whether the terminal is a control or an indicator. An arrow appears on the right if the terminal is a control, and an arrow appears on the left if the terminal is an indicator.

Control and Indicator Data Types

The following table shows the symbols and uses for the different types of control and indicator terminals.

The color and symbol of each terminal indicate the data type of the corresponding control or indicator. Many data types have a corresponding set of functions that can manipulate the data, such as the String functions on the String palette that correspond to the string data type.

Refer to [Numeric Data Types](#) for more information about using numeric data types.

Control	Indicator	Data Type	Use	Default Values
		Single-precision, <i>floating-point</i> numeric	Saves memory and does not overflow the range of the numbers.	0.0
		Double-precision, floating-point numeric	Is the default format for numeric objects.	0.0
		Extended-precision, floating-point numeric	Performs differently depending on the platform. Use only when necessary.	0.0

Control	Indicator	Data Type	Use	Default Values
		<i>Complex</i> single-precision, floating-point numeric	Same as single-precision, floating-point, with a real and an imaginary part.	0.0 + 0.0i
		Complex double-precision, floating-point numeric	Same as double-precision, floating-point, with a real and an imaginary part.	0.0 + 0.0i
		Complex extended-precision, floating-point numeric	Same as extended-precision, floating-point, with a real and an imaginary part.	0.0 + 0.0i
		<i>Fixed-point</i> numeric	Stores values that fall within a user-defined range. This data type is useful when you do not need the dynamic range of floating-point representation or when you want to save resources when working with a target on which floating-point arithmetic is expensive in terms of FPGA resource usage, such as an FPGA target.	0.0
		8-bit <i>signed integer</i> numeric	Represents whole numbers and can be positive or negative.	0
		16-bit signed integer numeric	Same as above.	0
		32-bit signed integer numeric	Same as above.	0
		64-bit signed integer numeric	Same as above.	0
		8-bit <i>unsigned integer</i> numeric	Represents only non-negative integers and has a larger range of positive numbers than signed integers because the number of bits is the same for both representations.	0
		16-bit unsigned integer numeric	Same as above.	0
		32-bit unsigned integer numeric	Same as above.	0

Control	Indicator	Data Type	Use	Default Values
		64-bit unsigned integer numeric	Same as above.	0
		<64.64>-bit <i>time stamp</i>	Stores absolute time with high precision.	12:00:00.000 AM 1/1/1904 (Universal Time)
		<i>Enumerated type</i>	Gives users a list of items from which to select.	—
		<i>Boolean</i>	Stores Boolean (TRUE/FALSE) values.	FALSE
		<i>String</i>	Provides a platform-independent format for information and data, which you can use to create simple text messages, pass and store numeric data, and so on.	empty string
		<i>Array</i>	Encloses the data type of its elements in square brackets and takes the color of that data type. As you add dimensions to the array, the brackets become thicker.	—
		A <i>matrix</i> of complex elements.	The wire pattern differs from that of an array of the same data type.	—
		A <i>matrix</i> of real elements.	The wire pattern differs from that of an array of the same data type.	—
		<i>Cluster</i>	Encloses several data types. Cluster data types appear brown if all elements in the cluster are numeric or pink if any elements in the cluster are of non-numeric types. Error code clusters appear dark yellow, while <i>LabVIEW class</i> clusters are crimson by default or teal green for Report Generation VIs.	—
		<i>Path</i>	Stores the location of a file or directory using the standard syntax for the platform you are using.	empty path

Control	Indicator	Data Type	Use	Default Values
		<i>Dynamic</i>	(Express VIs) Includes data associated with a signal and the attributes that provide information about the signal, such as the name of the signal or the date and time the data was acquired.	—
		<i>Waveform</i>	Carries the data, start time, and Δt of a waveform.	—
		<i>Digital waveform</i>	Carries start time, Δx , the digital data, and any attributes of a digital waveform.	—
		<i>Digital</i>	Encloses data associated with <i>digital signals</i> .	—
		<i>Reference number</i> (refnum)	Acts as a unique identifier for an object, such as a file, device, or network connection.	—
		<i>Variant</i>	Includes the control or indicator name, the data type information, and the data itself.	—
		<i>I/O name</i>	Passes resources you configure to I/O VIs to communicate with an instrument or a measurement device.	—
		<i>Picture</i>	Includes a set of drawing instructions for displaying pictures that can contain lines, circles, text, and other types of graphic shapes.	—

Select consistent and simple data types to *optimize memory usage*.

Symbolic Numeric Values for Floating-Point Data

Undefined or unexpected data invalidate all subsequent operations. Floating-point operations return the following two symbolic values that indicate faulty computations or meaningless results:

- NaN (not a number) represents a floating-point value that invalid operations produce, such as taking the square root of a negative number.

- **Inf** (infinity) represents a floating-point value outside of the range for that data type. For example, dividing 1 by zero produces **Inf**. LabVIEW can return **+Inf** or **-Inf**.

LabVIEW does not check for overflow or underflow conditions on integer values.

Overflow and underflow for floating-point numbers is in accordance with IEEE 754, Standard for Floating-Point Arithmetic.

Floating-point operations propagate **NaN** and **Inf** reliably. However, integers and fixed-point numbers do not support symbolic numeric values. When you explicitly or implicitly **convert** **+Inf** to an integer or fixed-point number, the value becomes the largest value for that data type. For example, converting **+Inf** to a 16-bit signed integer produces the value 32,767, the largest possible value for a 16-bit signed integer. LabVIEW converts **-Inf** to the smallest value for that data type.

When you explicitly or implicitly convert **NaN** to an integer or a fixed-point value, the value becomes the largest value for that data type.

Before you convert data to integer or fixed-point data types, use the **Probe** tool to check intermediate floating-point values for validity. Check for **NaN** programmatically by wiring the Comparison function Not A Number/Path/Refnum? to the value you suspect is invalid. You also can use the In Range and Coerce function to keep values within a range you specify.



Note (FPGA Module) The bit pattern of **NaN** outputs on the FPGA may differ from outputs on the development computer.

Constants

Constants are terminals on the block diagram that supply fixed data values to the block diagram. Universal constants are constants with fixed values, such as pi (π) and infinity (∞). User-defined constants are constants you define and edit before you run a VI.

Label a constant by right-clicking the constant and selecting **Visible Items»Label** from the shortcut menu. Universal constants have predetermined values for their labels that you can edit by using the Operating tool or the Labeling tool.

Most constants are located at the bottom or top of their palettes.

Universal Constants

Use universal constants for mathematical computations and formatting strings or paths. LabVIEW includes the following types of universal constants:

- **Universal numeric constants** —A set of high-precision and commonly used mathematical and physical values, such as the natural logarithm base (e) and the

speed of light. The universal numeric constants are located on the Math & Scientific Constants palette.

- **Universal string constants** —A set of commonly used non-displayable string characters, such as line feed and carriage return. The universal string constants are located on the String palette.
- **Universal file constants** —A set of commonly used file path values, such as Not a Path, Not a Refnum, and Default Directory. The universal file constants are located on the File Constants palette.

User-Defined Constants

The Functions palette includes constants organized by type, such as Boolean, numeric, ring, enumerated type, color box, string, array, cluster, and path constants.

Create a user-defined constant by right-clicking an input terminal of a VI or function and selecting **Create»Constant** from the shortcut menu.

You cannot change the value of user-defined constants when the VI is running.

You also can create a constant by dragging a front panel control to the block diagram. LabVIEW creates a constant that contains the value of the front panel control at the time you dragged it to the block diagram. The front panel control remains on the front panel. Changing the value of the control does not affect the constant value and vice versa.

Use the Operating or Labeling tool to click the constant and edit its value. If *automatic tool selection is enabled*, double-click the constant to switch to the Labeling tool and edit the value.

Constant Folding and Loop Invariants

LabVIEW uses constant folding and loop invariants to optimize the performance of VIs.

- With constant folding, LabVIEW computes the result of operations on constant values on the first run of the VI instead of calculating the result during each run. When applying constant folding to structures such as For Loops for which all inputs are constants, LabVIEW executes the structure and computes the outputs of the structure on the first run of the VI instead of calculating the outputs during each run.

Note If you enable debugging, LabVIEW does not use constant folding for structures.

- Loop invariants are non-constant values that a For Loop or a While Loop computes that do not change as the loop iterates. For example, an Add function inside a For Loop wired to only non-indexing input tunnels is a loop invariant because the output of the Add function does not change as the loop iterates. LabVIEW computes the values of loop invariants before the loop runs instead of calculating them during each iteration of the loop.

You can configure LabVIEW to display constant folding and loop invariant hash marks on the block diagram by selecting **Tools»Options»Block Diagram** and configuring the **Hash Marks** options in the Block Diagram page.

Block Diagram Nodes

Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs. They are analogous to statements, operators, functions, and subroutines in text-based programming languages. LabVIEW includes the following types of nodes:

- **Functions** —Built-in execution elements, comparable to an operator, function, or statement.
- **SubVIs** —VIs used on the block diagram of another VI, comparable to subroutines.
- **Express VIs** —SubVIs designed to aid in common measurement tasks. You configure an Express VI using a configuration dialog box.
- **Structures** —Execution control elements, such as For Loops, While Loops, Case structures, Flat and Stacked Sequence structures, Timed structures, and **Event structures**.
- **Formula and Expression Nodes** —Formula Nodes are resizable structures for entering equations directly into a block diagram. Expression Nodes are structures for calculating expressions that contain a single variable.
- **Property and Invoke Nodes** —Property Nodes are structures for setting or finding properties of a class. Invoke Nodes are structures for executing methods of a class.
- **Call By Reference Nodes** —Structures for calling a dynamically loaded VI.
- **Call Library Function Nodes** —Structures for calling most standard shared libraries or DLLs.

Polymorphic VIs and Functions

Polymorphic VIs and functions can adjust to input data of different data types. Most LabVIEW structures are polymorphic, as are some VIs and functions.

Functions are **polymorphic** to varying degrees—none, some, or all of their inputs can be polymorphic. Some function inputs accept numeric values or Boolean values. Some accept numeric values or strings. Some accept not only scalar numeric values, but also arrays of numeric values, clusters of numeric values, arrays of clusters of numeric values, and so on. Some accept only one-dimensional arrays, although the array elements can be of any type. Some functions accept all types of data, including complex numeric values.

Numeric Data

LabVIEW represents numeric data as **floating-point numbers**, **fixed-point numbers**, **integers**, **unsigned integers**, and **complex numbers**.



NOTE

The difference among the numeric data types is the number of bits they use to **store data** and the data values they represent.

Certain data types also provide extended configuration options. For example, you can associate physical units of measure with floating-point data, including complex numbers, and you can [configure the encoding and range](#) for fixed-point data.

Associating Numeric Units with Floating-Point Numbers

You can [associate physical units of measure](#), such as meters or kilometers/second, with any numeric control or indicator that has floating-point representation.

Units for a control appear in a separate owned label, called the unit label. Display the unit label by right-clicking the control and selecting **Visible Items»Unit Label** from the shortcut menu. Right-click the unit label and select **Build Unit String** from the shortcut menu to edit the unit label.

When LabVIEW displays the unit label, you can enter a unit using [standard abbreviations](#) such as m for meters, ft for feet, s for seconds, and so on.

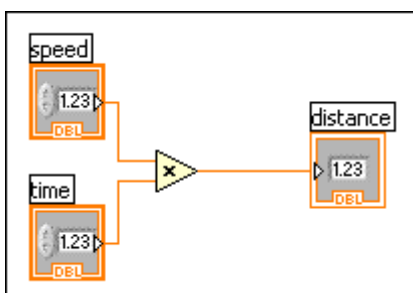
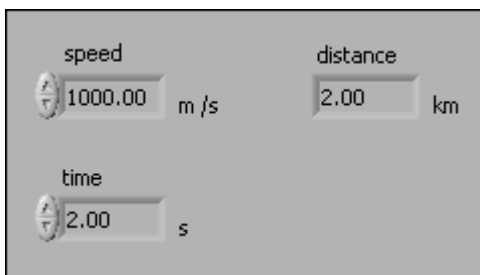


Note You cannot use units in Formula Nodes.

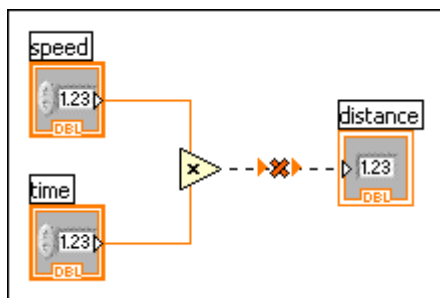
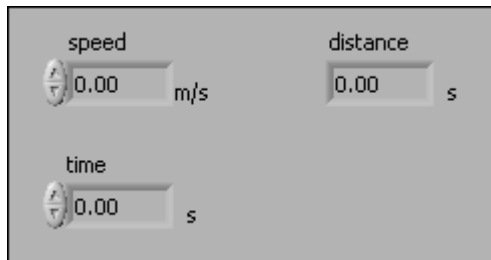
Units and Strict Type Checking

When you associate units with an object, you can wire only objects that have compatible units. LabVIEW uses strict type checking to verify that units are compatible. If you wire two objects with incompatible units, LabVIEW returns an error. For example, LabVIEW returns an error if you wire an object with mile as its unit type to an object with liter as its unit type, because a mile is a unit of distance and a liter is a unit of volume.

The following VI shows wiring objects with compatible units. In this VI, LabVIEW automatically scales the **distance** indicator to display kilometers instead of meters because kilometers is the unit for the indicator.

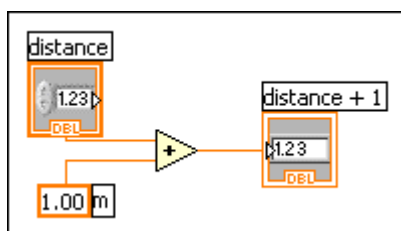


An error occurs in the following VI because **distance** has a unit type of seconds. To correct the error, change seconds to a unit of distance, such as kilometers, as shown in the previous VI.



Some VIs and functions are ambiguous with respect to units. You cannot use these VIs and functions with other terminals that have units. For example, the Increment function is ambiguous with respect to units. If you use distance units, the Increment function cannot tell whether to add one meter, one kilometer, or one foot. Because of this ambiguity, you cannot use the Increment function and other functions that increment or decrement values with data that have associated units.

To avoid ambiguity in this example, use a numeric constant with the proper unit and the Add function to create your own increment unit function, as shown in the following block diagram.



Fixed-Point Data Type

The fixed-point data type is a numeric data type that represents a set of rational numbers using binary digits, or bits. Unlike the *floating-point* data type, which allows the precision and range of the data to vary, you can configure fixed-point numbers to always use a specific number of integer and fractional bits.



Note To represent a rational number using the fixed-point data type, the denominator of the rational number must be reducible to a power of 2, because the binary number system is a base-2 number system.

Use the fixed-point data type when you do not need the dynamic range of floating-point representation or when you want to work with a target that does not support floating-point arithmetic.



Note (FPGA Module) Refer to Using the Fixed-Point Data Type for more information about using the fixed-point data type on an FPGA target. Evaluate your usage of numeric data types to determine which data type best suits your design.

Representing Rational Numbers with the Fixed-Point Data Type

A fixed-point number has a fixed number of integer and fractional bits. The integer bits appear to the left of the binary point, and the fractional bits appear to the right of the binary point. Because you can specify the exact number of bits you want to use, the binary point, which is the binary equivalent of the decimal point, has a fixed position.

For example, you can configure LabVIEW to represent the rational number 0.5 as a fixed-point number with 8 total bits and 4 integer bits.

Rational Number	Fixed-Point Equivalent
0.5	0000.1000

You also can configure LabVIEW to represent the same rational number as a fixed-point number with 16 total bits and 8 integer bits.

Rational Number	Fixed-Point Equivalent
0.5	00000000.10000000

Fixed-point numbers differ from floating-point numbers in that floating-point numbers allow the number of integer and fractional bits to vary when you perform an operation on the floating-point number. In other words, the binary point can move, or float, in a floating-point number.



Note If you do not specify the exact number of bits you want to use to represent the fixed-point equivalent of a rational number, LabVIEW adjusts the number of bits to avoid data loss, if possible. LabVIEW cannot handle numbers longer than 64 bits.

Configuring Fixed-Point Numbers

To set a number to fixed-point representation, right-click the numeric object and select **Representation** from the shortcut menu to change the data type of the object. You can *configure the encoding* for fixed-point numbers. You also can specify whether to *include an overflow status* with fixed-point numbers and how Numeric functions handle *overflow and rounding* conditions for fixed-point numbers. To configure a fixed-point number, right-click a constant, control, indicator, or Numeric function and select **Properties** from the shortcut

menu to display the **Numeric Properties**, **Numeric Constant Properties**, or **Numeric Node Properties** dialog box.

Encoding

Specify the binary encoding of a fixed-point number when you want the number to conform to a certain bit size.

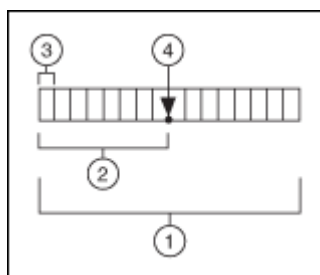
- **Signed** —The option that determines whether the fixed-point data is signed. If you select signed, the sign bit is always the first bit in the bit string that represents the data.
- **Unsigned** —The option that determines whether the fixed-point data is unsigned.
- **Word length** —The total number of bits in the bit string that LabVIEW uses to represent all possible values of the fixed-point data. LabVIEW accepts a maximum word length of 64 bits. Certain targets might *limit data to smaller word lengths*. If you open a VI on a target and the VI contains fixed-point data with larger word lengths than the target can accept, the VI contains broken wires. Refer to the documentation for a target to determine the maximum word length the target accepts.
- **Integer word length** —The number of integer bits in the bit string that LabVIEW uses to represent all possible values of the fixed-point data, or, given an initial position to the left or right of the most significant bit, the number of bits to shift the binary point to reach the most significant bit. The integer word length can be larger than the word length, and can be positive or negative.

LabVIEW displays this configuration in the following format in the Context Help window when you move the cursor over a wire that carries fixed-point data:

< encoding, word length, integer word length >

For example:

<s, 16, 8> is a signed, 16-bit number with eight integer bits. The following image represents a signed number with a word length of 16 and an integer word length of 8.

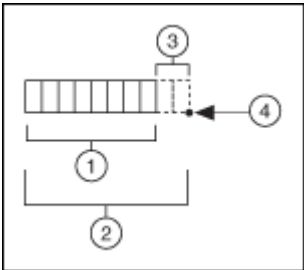


1	Word length (16 bits)
2	Integer word length (8 bits)
3	Sign bit

4	Binary point

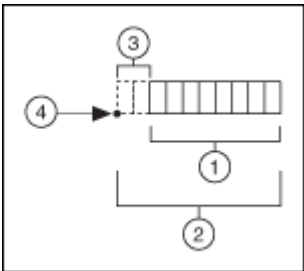
Long or Negative Integer Word Lengths

If the integer word length is larger than the word length, LabVIEW does not store the integer bits that exceed the word length. The following image represents a number with a word length of 8 and an integer word length of 10.



1	Word length (8 bits)
2	Integer word length (10 bits)
3	Integer bits not stored
4	Binary point

If the integer word length is negative, LabVIEW does not store any integer bits and also does not store the number of fractional bits equal to the negative number, starting from the binary point. The following image represents a number with a word length of 8 and an integer word length of -2.



1	Word length (8 bits)
2	Word length minus Integer word length (8 bits minus -2 bits)
3	Fractional bits not stored
4	Binary point

Range

LabVIEW calculates the range and delta for a fixed-point number according to the values you specify for the encoding of the fixed-point number.

- **Minimum** —The minimum value for the fixed-point data range.
- **Maximum** —The maximum value for the fixed-point data range.
- **Delta** —The increment between numbers within the specified range.

LabVIEW expresses this configuration in the following format in the Context Help window when you move the cursor over a wire that carries fixed-point data:

[Minimum, Maximum]: Delta

For example:

[-4, 4]:1 has a minimum value of -4, a maximum value of 4, and a delta of 1.

Fixed-Point Arithmetic

When you perform an arithmetic operation on fixed-point numbers for which you specified a word length, the position of the binary point does not change from the position you specify before you run the VI. This behavior contrasts with floating-point numbers, where the position of the binary point or decimal point can shift at run time due to an arithmetic operation. If the result of an arithmetic operation requires a larger word length than the fixed-point configuration allows, you might lose data from the result of the operation.

LabVIEW adjusts the word length of output values of Numeric functions to avoid data loss, if possible. However, LabVIEW cannot avoid data loss if the word length required to represent the output value exceeds the maximum you specify or the LabVIEW maximum of 64 bits. To [configure output settings](#) for a Numeric function, right-click the function and select **Properties** from the shortcut menu to display the **Numeric Node Properties** dialog box.

Overflow and Rounding

Overflow and rounding conditions can occur when you perform an arithmetic operation on fixed-point numbers or when you use the To Fixed-Point function to convert numeric data to fixed-point data. LabVIEW configures the outputs of Numeric functions to avoid overflow or rounding of fixed-point data, if possible. However, LabVIEW cannot avoid overflow or rounding under certain conditions.

For example, if you wire fixed-point values to Numeric functions such as Add, Subtract, Multiply, and Square, LabVIEW increases the word length of the output value to accommodate the result of the operation by default. This behavior is similar to the behavior of floating-point values. However, if avoiding overflow or rounding requires an output value that exceeds the maximum word length that LabVIEW accepts or the maximum word length that you specify, overflow or rounding conditions can occur. LabVIEW accepts a maximum word length of 64 bits.



Note If you wire fixed-point values to the Divide function, the Reciprocal function, or the Square Root function, rounding conditions always occur because the precision of the results can be infinite.

Other conditions where overflow or rounding can occur include [type cycles and values that you cannot represent](#) exactly in a base-2 number system.

Overflow

Overflow occurs when the result of an operation is outside the range of values that the output type can represent. You can select between the following overflow modes to handle overflow:

- **Saturate** —If the value is greater than the maximum value in the desired range of the output type, LabVIEW coerces, or rounds, the value to the specified maximum. If the value is less than the minimum value in the desired range of the output type, LabVIEW coerces the value to the specified minimum.
- **Wrap** —If the value is outside the desired range of the output type, LabVIEW discards significant bits of the value until the value is in the desired range of the output type. This option requires the output value to fit the encoding you specify. You cannot specify a desired range if you select this option, because LabVIEW automatically sets the range to fit the encoding.



Note Typically, the Saturate mode requires additional hardware resources on an FPGA.

To determine whether overflow occurs, you can [configure a fixed-point number](#) to include an overflow status. When you include an overflow status in a fixed-point number, LabVIEW allocates additional storage space to track whether the fixed-point number is the result of an operation that overflowed. After you configure a fixed-point number to include an overflow status, you can [display an overflow status LED](#) on fixed-point controls, constants, and indicators. This LED lights up when the overflow status of the fixed-point number is TRUE. You also can use the Fixed-Point Overflow? function to determine the overflow status of a fixed-point number. Use the Fixed-Point functions to manipulate the overflow status of a fixed-point number.



Note LabVIEW does not check for underflow conditions on fixed-point numbers.

When you use the Comparison functions to compare fixed-point numbers, LabVIEW ignores the overflow status and compares only the numbers. For example, if you use the Equal? function to compare a fixed-point value of 1.5 with an overflow status of TRUE to a fixed-point value of 1.5 with no overflow status, LabVIEW returns TRUE.

To [configure output settings](#) for a Numeric function, right-click the function and select **Properties** from the shortcut menu to display the **Numeric Node Properties** dialog box. You can use this dialog box to configure how LabVIEW handles overflow and rounding conditions.

Rounding

Rounding occurs when the precision of the input value or the result of an operation is greater than the precision of the output type. LabVIEW coerces the input value to a value with the same precision as the output type. For example, assume the output type can represent the successive values s and t . If the result of an operation is the value x , and x is between s and t such that $s < x < t$, rounding occurs because the output type cannot represent x exactly.

You can select from the following rounding modes to handle rounding:

- **Truncate** —Rounds the value down to the nearest value that the output type can represent. LabVIEW discards the least significant bits of the value. This rounding mode has the best performance but also produces the least accurate output values. For example, if the output value x falls between the successive values s and t such that $s < x < t$, LabVIEW sets x to equal s .
- **Round-Half-Up** —Rounds the value to the nearest value that the output type can represent. If the value to round is exactly between two valid values, this mode rounds the value up to the higher of the two valid values. LabVIEW adds half a least significant bit to the output value and then truncates the value. This rounding mode produces more accurate output values than truncate but also has a larger impact on performance. For example, if the output value x falls between the successive values s and t such that $s < x < t$, LabVIEW sets x to equal the closer of s or t . If x is exactly between s and t , LabVIEW selects t .
- **Round-Half-Even** —Rounds the value to the nearest value that the output type can represent. If the value to round is exactly between two valid values, LabVIEW checks the bit of the value that becomes the least significant bit after rounding. If the bit is 0, this mode rounds the value to the lower of the two values that the output type can represent. If the bit is not 0, this mode rounds the value to the higher of the two values. This rounding mode has the largest impact on performance but produces more accurate output values than truncate. This mode also neutralizes the bias towards higher values that can occur when you perform multiple rounding operations that use the round-half-up mode. This is the default rounding mode. For example, if the output value x falls between the successive values s and t such that $s < x < t$, LabVIEW sets x to equal the closer of s or t . If x is exactly between s and t , select s if the least significant bit after rounding is 0. Select t if the least significant bit after rounding is not 0.



Note The performance impact of any rounding mode is minimal. However, the performance impact can be significant if you run a VI that handles overflow on a target such as FPGA.

Refer to [Using the Fixed-Point Data Type](#) for more information about performance impact on an FPGA.

Using Polymorphic Units

Use units to gain an additional level of consistency checking when you evaluate expressions and formulas, also known as dimensional analysis. With LabVIEW, you do not

have to worry about conversions among systems of units because LabVIEW handles unit conversion when it displays data or when you enter data.

The LabVIEW built-in functions, such as Add and Multiply, are polymorphic with respect to units and automatically handle different units. However, to build a subVI with the same polymorphic unit capability, you must [use polymorphic units](#).

You can use polymorphic units for one VI to do the same calculation regardless of the units the inputs receive. For example, if you want to create a VI that computes the root-mean-square value of a waveform, you must define the unit associated with the waveform. A separate VI is necessary for voltage waveforms, current waveforms, temperature waveforms, and so on. However, instead of rewriting the same VI for each case, you can write a single subVI with polymorphic units and call it from the VIs with specific units.

LabVIEW treats a polymorphic unit as a unique unit. The polymorphic unit cannot convert to any other unit and propagates throughout the block diagram just as other units do. When you wire a control with the polymorphic unit $\$1$ to an indicator that also has the polymorphic unit $\$1$, the units match, and the VI can compile.

You can use $\$1$ in combinations like any other unit. For example, if you multiply a control by 3 seconds and wire it to an indicator, the indicator must be $\$1$ s units. Wiring an indicator that has different units from the control results in a broken wire.

A call to a subVI that contains polymorphic units computes output units based on the units its inputs receive. For example, suppose you create a subVI that has two inputs with the polymorphic units $\$1$ and $\$2$ that creates an output in the form $\$1\$2/s$. If you wire the subVI with inputs of m/s to the $\$1$ input and kg to the $\$2$ input, LabVIEW computes the output unit as kg m/s².

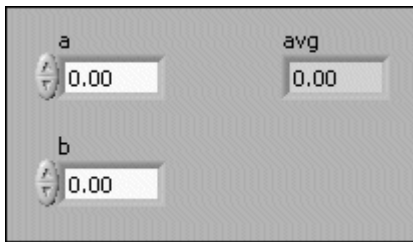
Suppose a different VI has two inputs of $\$1$ and $\$1/s$ and computes an output of $\$1^2$. If you wire this VI with inputs of m/s to the $\$1$ input and m/s² to the $\$1/s$ input, LabVIEW computes the output unit as m²/s². However, if you wire this VI with inputs of m to the $\$1$ input and kg to the $\$1/s$ input, the subVI call is broken. LabVIEW declares one of the inputs as a unit conflict and computes (if possible) the output from the other unit. A VI with polymorphic units can have a subVI with polymorphic units because LabVIEW keeps the respective units distinct.

When to Use Polymorphic Units

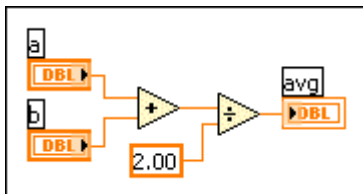
This section presents several examples of VIs with and without polymorphic units. It begins with a simple averaging example without any units. It then provides an example with a base unit. Next it turns this example into one with polymorphic units and uses it as a subVI for another example. The [Multiple Averages](#) example and the [Multiple Rates](#) example demonstrate when you might want to use a subVI with polymorphic unit capability.

Averaging without Units

The following front panel belongs to a VI that computes the average of two numbers. Notice that there are no units on the controls or indicator.

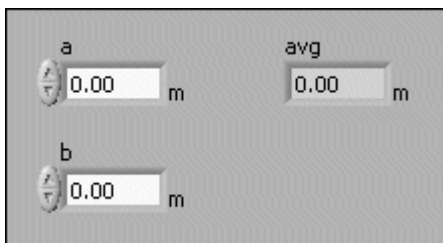


The following block diagram belongs to the VI.



Averaging with One Unit

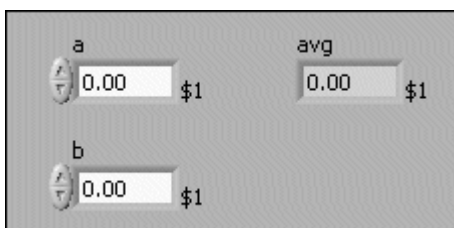
The following front panel belongs to a VI that computes the average of two numbers, each with a unit of meters.



The block diagram for this VI is identical to the previous block diagram. The units appear only on the front panel. If one control or indicator on the front panel has a unit, all controls and indicators on the front panel must have a unit or the block diagram displays broken wires, and the VI will not run.

Averaging with Polymorphic Units

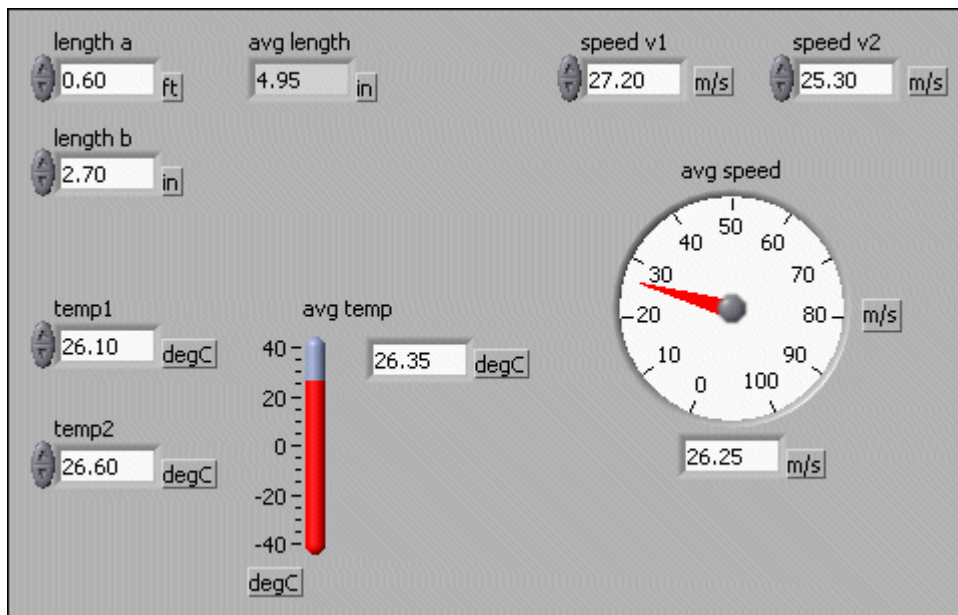
The following front panel belongs to a VI that computes the average of two numbers. Unlike the previous front panel, the following front panel has a polymorphic unit. The [Multiple Averages](#) example uses this VI as a subVI.



The block diagram for this VI is identical to the previous block diagram. As with the Averaging with Meters example, the polymorphic units appear only on the front panel. If one control or indicator on the front panel has a unit, all controls and indicators on the front panel must have a unit or the block diagram displays broken wires, and the VI will not run.

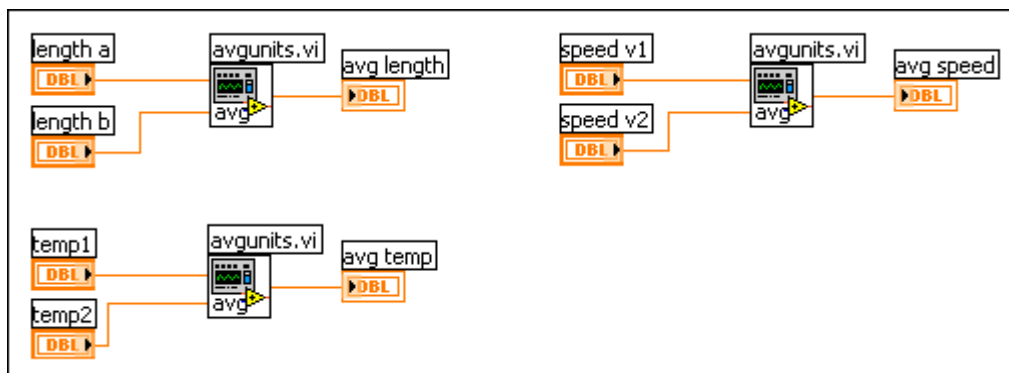
Multiple Averages

The following front panel belongs to a VI that calculates three different averages, each with different units, which is an example of when to use a subVI with polymorphic units.



Notice that **length a** has a unit of feet, **length b** has a unit of inches, and **avg length** has a unit of inches. LabVIEW automatically handles the conversion from feet to inches for you because this VI uses polymorphic units.

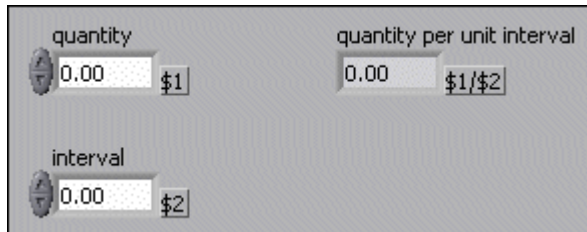
The following block diagram uses the [Averaging with Polymorphic Units VI](#) to average each example shown in the [Multiple Averages](#) example.



Notice that you can use the Averaging with Polymorphic Units subVI with all the units shown in the [Multiple Averages VI](#). Instead of creating a separate VI each time you want to average two numbers with units, you can use a subVI with polymorphic units.

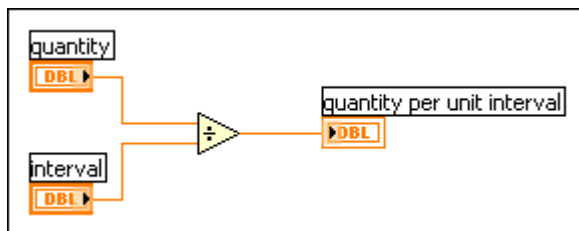
Rates with Polymorphic Units

You also can use polymorphic units when an indicator unit is a combination of the input units, such as calculating rates. The following front panel belongs to a VI that calculates a quantity per interval. You usually do not create such simple subVIs. This is just an example to illustrate two polymorphic units on a subVI. The [Multiple Rates](#) example uses this VI as a subVI.



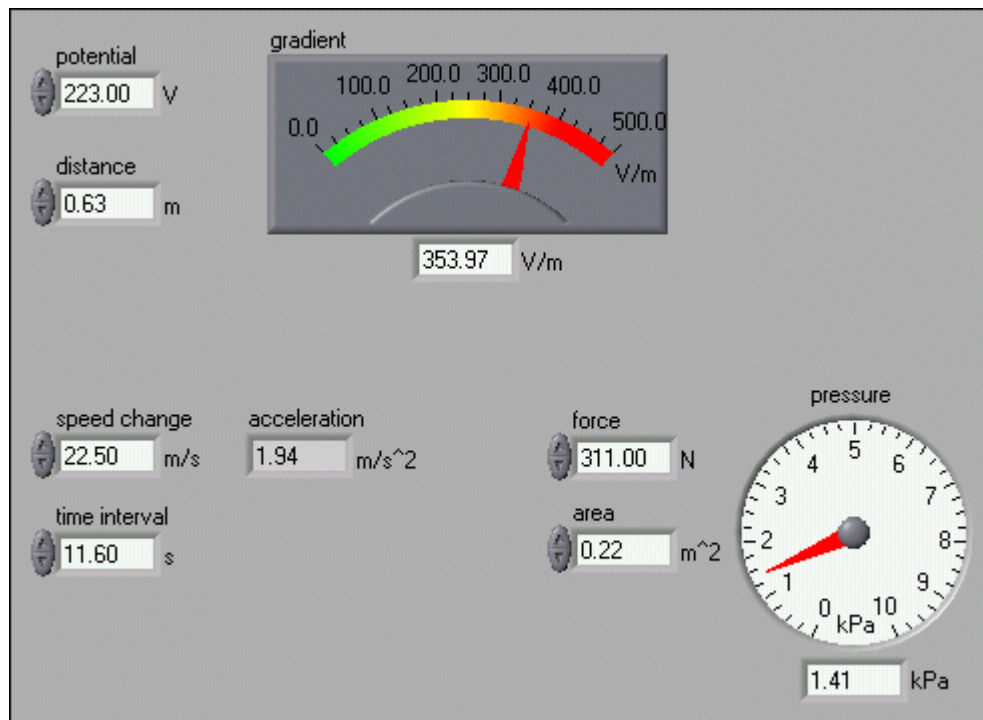
Notice that the quantity unit is \$1 and the interval unit is \$2. You can use any number, 1 through 9, for a polymorphic unit.

The following block diagram belongs to the VI.



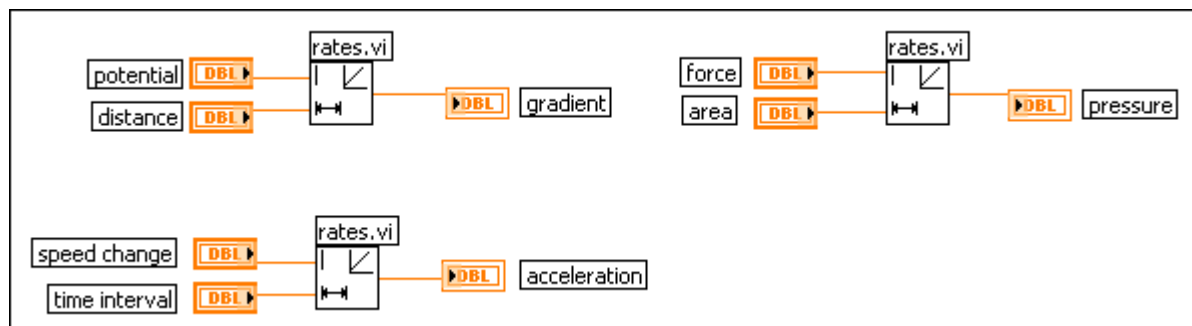
Multiple Rates

The following front panel belongs to a VI that calculates three different rates – gradient, acceleration, and pressure.



Notice that two of the units on the controls are compound units. The previous Rates with Polymorphic Units subVI accepts both simple and compound units.

The following block diagram belongs to the VI.



The block diagrams in [Multiple Averages](#) and [Multiple Rates](#) are almost the same. Each performs three different calculations using one subVI with polymorphic units.

Compiled Object Cache

A compiled object cache is the location where LabVIEW stores the compiled code for the following kinds of files:

- VIs and other file types that have [separate compiled code](#)
- VIs that you deploy to an RT target
- VIs that compile when you [build a build specification](#)

The object caches are located in the LabVIEW Data directory.

You do not need to manually interact with the object cache. However, you might want to clear the object cache if you delete a large number of VIs in the previous list. LabVIEW does not automatically delete the associated compiled code when you delete these kinds of VIs. To reclaim the disk space occupied by the compiled code of deleted VIs, you must clear the entire cache that contains the compiled code for the deleted VIs.



CAUTION

After you clear an object cache, LabVIEW requires time to recompile each file whose compiled code was stored in the object cache. This recompilation occurs the next time LabVIEW loads each file. Consider this performance tradeoff before clearing an object cache.

Types of Distributions

A distribution is a collection of software content wrapped up for consumption by clients of the software.

You can distribute your applications to clients into one of the following output types:

- **Package** —Contents of the distribution build into an NI Package Manager package file `.nipkg`. Your clients can install the distribution directly from Package Manager. You can also add the distribution to a feed so that clients can subscribe to the feed and install the distribution from Package Manager via network access.
- **Package installer** —Contents of the distribution build into an installer file `.exe`. By default, the package installer includes all the package dependencies in the output so that your clients can install the distribution without network access.

A feed is a collection of package files that includes a manifest containing information on the included packages. NI recommends using feeds when distributing multiple packages that have dependencies on one another. With feeds, you can create a single location where you host and maintain all packages intended for end user access.

Structure of LabVIEW File System

Organization of LabVIEW

Learn about the structure of the LabVIEW file system and the suggested locations for saving files.

Organization of the LabVIEW Directory Structure

Learn about the LabVIEW file system on Windows, macOS, and Linux. LabVIEW installs driver software for GPIB, DAQ, VISA, IVI, Motion Control, and IMAQ hardware depending on platform availability.

The LabVIEW directory installs in the following location depending on your operating system:

Windows (32-bit)—C:\Program Files (x86)\National Instruments\LabVIEW <VERSION>

Windows (64-bit)—C:\Program Files\National Instruments\LabVIEW <VERSION>

Linux—/usr/local/natinst/LabVIEW-<VERSION>-64

macOS—/Applications/National Instruments/LabVIEW <VERSION> 64-bit



NOTE

LabVIEW modules and toolkits you have installed might create additional directories within the LabVIEW directory.

After you complete installation, the LabVIEW directory contains the following groupings.

Libraries

user.lib—Directory in which you can [save controls and VIs](#) you create. LabVIEW displays controls on User Controls palettes and VIs on User Libraries palettes. This directory does not change if you upgrade or uninstall LabVIEW.

vi.lib—Contains libraries of built-in VIs, which LabVIEW displays in related groups on the Functions palette. Do not save files in the **vi.lib** directory because LabVIEW overwrites these files when you upgrade or reinstall.

instr.lib—Contains instrument drivers used to control PXI, VXI, GPIB, serial, and computer-based instruments. When you install NI instrument drivers and place them in this directory, LabVIEW adds them to the Instrument Drivers palette.

Structure and Support

menus—Contains files LabVIEW uses to configure the structure of the Controls and Functions palettes.

resource—Contains additional support files for the LabVIEW application. Do not save files into this directory because LabVIEW overwrites these files when you upgrade or reinstall.

project—Contains files that become items on the LabVIEW **Tools** menu.

ProjectTemplates—Contains starting points for common LabVIEW design patterns that you customize with the Create Project dialog box.

templates —Contains templates for common VIs.

wizard —Contains files that become items on the LabVIEW **File** menu.

www —Location of HTML files you can access through the Web Server.

Learning and Instruction

- **examples** —Contains example VIs. Select **Help»Find Examples** to browse or search the examples.
- **help** —contains files that become items on the LabVIEW Help menu.

Suggested Location for Saving Files

LabVIEW installs the `vi.lib` and the `resource` directories for LabVIEW system purposes only. Do not save your files in these directories.

You can save your files in the following directories:

user.lib —Any controls or VIs you want to display on the [User Controls or User Libraries palettes](#).



NOTE

Save subVIs in the `user.lib` directory only if they are usable, without modification, across projects. Paths to VIs in `user.lib` are relative to the `labview` directory. Paths to subVIs you save elsewhere are relative to the parent VI. Therefore, copying a VI from `user.lib` to modify it for a special case does not change the path to its subVIs located in `user.lib`.

If you want to copy a VI and its subVIs to another directory or computer, do not save the VIs in `user.lib` because when you run the top-level VI from the new location LabVIEW references the original subVIs in `user.lib`.

instr.lib —Any instrument driver VI you want to display on the Instrument Drivers palette.

project —VIs you use to extend LabVIEW capabilities. VIs you store in this directory appear on the **Tools** menu.

wizard —VIs you store in this directory appear on the **File** menu.

www —Location of HTML files you can access through the Web Server.

help —Any VIs, `.hlp`, and `.chm` files that you want to make available on the **Help** menu.

LabVIEW Data —Any data files LabVIEW generates, such as `.lvn` or `.txt` files.

You also can [create a directory](#) anywhere on your hard drive to store LabVIEW files that you create.

When you open a VI you place in the `labview\help`, `labview\project`, and `labview\wizard` directories from the **Help**, **Tools**, or **File** menu, LabVIEW opens the VI in a private [application instance](#) (NI.LV.Dialog).

Grouping Data Using Strings, Clusters, Arrays, and Collections

Use the following data types to group data in LabVIEW:

- **String** —Groups sequences of ASCII characters.
- **Cluster** —Groups data elements of mixed types.
- **Array** —Groups data elements of the same type in unsorted order. Includes the matrix data type, a 2D array you can use to make modeling of math problems easier:
 - **Matrix** —Groups rows or columns of real or complex scalar data for some math operations, such as linear algebra operations.
- **Collection** —Includes the following sub-types:
 - **Set** —Groups unique data elements of the same type in sorted order.
 - **Map** —Groups key-value pairs where all the keys are unique. The key and value can each be any data type.

Related Concepts

[Grouping Data with Strings](#)

[Grouping Data with Arrays and Clusters](#)

[Grouping Data with Collections](#)

[Using Matrices](#)

LabVIEW Subpalettes

Graph Subpalette

Use the **Graph** palette objects to plot numeric data on a graph, chart, or plot.



NOTE

The availability of graph and chart controls and indicators may vary depending on the subpalette style you choose. Refer to the Modern, Silver, System, or Classic subpalette for graph control and indicator availability.

Palette Object	Description
<i>Waveform Chart and Graph</i>	Displays data typically acquired at a constant rate.
<i>XY Graph</i>	Displays data acquired at a non-constant rate and data for multivalued functions.
<i>Express XY Graph</i>	Displays the same data as an XY Graph. You can configure the settings of an Express XY Graph interactively through a dialog box.
<i>Intensity Chart and Graph</i>	Displays 3D data on a 2D plot by using color to display the values of the third dimension.
<i>Digital Waveform Graph</i>	Displays data as pulses or groups of digital lines.
<i>Mixed Signal Graph</i>	Displays data types accepted by waveform graphs, XY graphs, digital waveform graphs, and clusters that contain any combination of those data types.
<i>Compass Plot</i>	Displays vectors that emanate from the center of a circular grid.
<i>Error Bar Plot</i>	Displays error bars at each point above and below a line graph.
<i>Feather Plot</i>	Displays vectors that emanate from equally spaced points along a horizontal axis.
<i>XY Plot Matrix</i>	Displays rows and columns of scatter graphs.
<i>3D Picture</i>	Displays graphical representations of 3D objects.

Subpalette	Description
<i>Controls</i>	Use the Controls palette objects to display advanced graphs and graphical representations of 2D objects.
<i>3D Graph</i>	Use the 3D Graph palette objects to plot data in 3D.

Controls Subpalette

Use the **Controls** palette objects to display advanced graphs and graphical representations of 2D objects.

Palette Object	Description
<i>ActiveX 3D Graph</i>	Displays 3D data on a 3D plot in an ActiveX object on the front panel.
<i>Polar Plot</i>	Displays data represented by polar coordinates.
<i>Smith Plot</i>	Displays complex reflection coefficients corresponding to complex impedances on a polar plot.
<i>Min-Max Plot</i>	Displays the minimum and maximum of a plot.
<i>Distribution Plot</i>	Displays a frequency histogram.
<i>Radar Plot</i>	Displays multivariate data in a circular plot with one axis for each variable.
<i>2D Picture</i>	Displays graphical representations of 2D objects.

Subpalette	Description
<i>Datatypes</i>	Use the Datatypes palette objects with Picture Functions VIs and the <i>2D Picture</i> control to create visual representations of data.

Datatypes Subpalette

Use the **Datatypes** palette objects with Picture Functions VIs and the *2D Picture* control to create visual representations of data.

Palette Object	Description
<i>Point</i>	Use to define x- and y-coordinates for a point.
<i>Rectangle</i>	Use to define the corners of a rectangle.
<i>Font Enum</i>	Use to select either a User-specified Font or one of three built-in fonts: Application Font , System Font , and Dialog Font .
<i>User Font</i>	Use when Font Enum is set to User-specified Font to specify font, text size, and text formatting.
<i>Text Alignment</i>	Use to set the horizontal and vertical alignment of text.

3D Graph Subpalette

Use the **3D Graph** palette objects to plot data in 3D.

Palette Object	Description
<i>Scatter</i>	Displays the relationship between two variables as a set of discrete points.
<i>Bar</i>	Displays a vertical bar graph.
<i>Pie</i>	Displays a pie chart.
<i>Ribbon</i>	Displays a parallel line plot.
<i>Contour</i>	Displays a curve along which a function with two variables has a constant value.
<i>Quiver</i>	Displays vector arrows representing the magnitude and direction of a set of points. Quiver plots are also known as vector fields.
<i>Comet</i>	Displays an animated graph with a circle that follows the data points.
<i>Surface</i>	Displays data with a connecting surface.
<i>Mesh</i>	Displays a mesh surface with open spaces.
<i>Waterfall</i>	Displays the surface of the data and the area on the y-axis below the data points.
<i>3D Surface Graph</i>	Displays 3D data with a connecting surface.
<i>3D Parametric Graph</i>	Displays a graph such that the x, y, and z variables are each functions of a common parameter.
<i>3D Line Graph</i>	Displays a line or curve in terms of x, y, and z points.

LabVIEW Controls and Indicators

I/O Name Controls and Indicators

Use the I/O name controls and indicators on the **I/O** and **Classic I/O** palettes to pass DAQ channel names, VISA resource names, and IVI logical names you configure to I/O VIs to communicate with an instrument or a DAQ device.

I/O name constants are on the Functions palette. A constant is a terminal on the block diagram that supplies fixed data values to the block diagram.

**NOTE**

All I/O name controls or constants are available on all platforms, which allows you to develop I/O VIs on any platform that can communicate with devices that are platform specific. However, if you try to run a VI with a platform-specific I/O control on a platform that does not support that device, you receive an error.

(Windows) Use Measurement & Automation Explorer (MAX), available from the **Tools** menu, to configure DAQ channel names, VISA resource names, and IVI logical names.

(macOS and Linux) Use the configuration utilities for the instrument to configure VISA resource names and IVI logical names. Refer to the documentation for the instrument for more information about the configuration utilities.

The IMAQ session control is a unique identifier that represents the connection to the hardware.

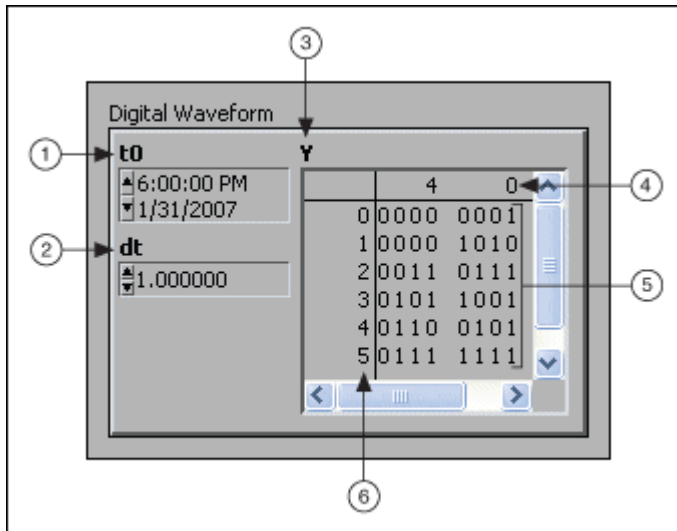
Waveform Control

Use the waveform control to manipulate individual data elements of a [waveform](#). The waveform control carries the data, start time, and delta t of a waveform.

Digital Waveform Control

Use the digital waveform control to manipulate the individual elements of a [digital waveform](#). Use the digital waveform control to specify start time, delta t, the data, and the attributes of a digital waveform.

The digital waveform control in the following illustration displays digital data with six samples that each contain eight signals, a start time of 6:00 PM on 1/31/2007, and delta t of 1.

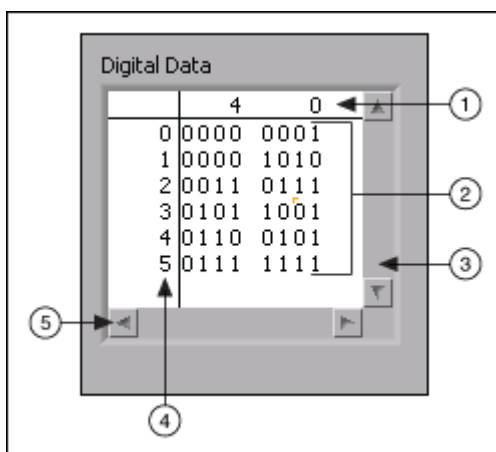


1 Start Time	3 Digital Data	5 Data
2 Delta t	4 Signals	6 Samples

Digital Data Control

The digital data control displays digital data arranged in rows and columns as a *digital table*. Use the digital data control to build digital waveforms or to display digital data extracted from a digital waveform. Wire the digital waveform data control to a digital data indicator to view the samples and signals of a digital waveform.

The digital data control in the following illustration displays six samples that each contain eight signals.



1 Signals	3 Vertical scroll bar	5 Horizontal scroll bar
2 Data	4 Samples	

You can *insert and delete* rows and columns in the digital data control. Use the shortcut menu to *cut, copy, and paste* digital data within the control.

You can display the data in the digital data control in binary, hexadecimal, octal, and decimal formats. The digital states L, H, Z, X, T, and V, which are states some measurement devices use, appear as question marks in the control if you choose to display the values in hexadecimal, octal, or decimal format. Right-click the control, select **Data Format** from the shortcut menu, and select a data format for the control.

Digital Tables

The digital data control and digital waveform control contain data in digital tables. A digital table can represent data in eight digital data states. The following table describes the digital data states.

Value	Digital Data State	Description
0	0 (Drive Low)	Forces logic low. Drives to the low voltage level (V_{OL} .
1	1 (Drive High)	Forces logic high. Drives to the high voltage level (V_{OH} .
2	Z (Force Off)	Forces logic high impedance. Turns the driver off.
3	L (Compare Low)	Compares logic low (edge). Compares a voltage level lower than the low voltage threshold.
4	H (Compare High)	Compares logic high (edge). Compares a voltage level higher than the high voltage threshold.
5	X (Compare Unknown)	Compares logic unknown. Does not compare.
6	T (Compare Off)	Compares logic high impedance (edge). Compares a voltage level between the low voltage threshold (V_{OL} and the high voltage threshold (V_{OH} .
7	V (Compare Valid)	Compares logic valid level (edge). Compares a voltage level either lower than the low voltage threshold (V_{OL} or higher than the high voltage threshold (V_{OH} .



NOTE
You can use the Build Digital Data function to *build a digital table*.

Compressing Digital Data

You can *compress the data* in a digital table. Compress digital data when you want to display two or more serial digital signals with the same bit sets on the same row to better visualize the data. For example, if you acquire 10 digital waveform samples and nine waveforms are identical but the tenth waveform differs, compressing the digital data helps you easily find which waveform is different. Compressing digital data also conserves memory resources.

The following illustration shows two digital data tables, each displaying the same data. The first table shows uncompressed digital data. The second table shows the same digital data after compression.

Digital Data	
	0
0	0101
1	0101
2	1111
3	1111
4	1111
5	0000
6	0000

Compressed Digital Data	
	0
0	0101
2	1111
5	0000
6	0000

You can use the Digital Waveform VIs and functions to compress, uncompress, and otherwise manipulate digital data and digital waveforms.

List, Table, and Tree Controls and Indicators

Use the listbox, tree control, and table controls and indicators located on the **List, Table & Tree** and **Classic List, Table & Tree** palettes to give users a list of items from which to select.

Listboxes

You can configure listboxes to accept single or multiple selections. Use the multicolumn listbox to display more information about each item, such as the size of the item and the date it was created.

When you type characters in a listbox at run time, LabVIEW selects the first item in the listbox that begins with the characters you type. Use the left and right arrow keys to go to the previous or next items that match the characters you type. Right-click a listbox and select **Selection Mode»Highlight Entire Row** from the shortcut menu to highlight the entire row when you select an item.

You can [add a symbol](#) next to a list item, such as in the LLB Manager window, where directories and files have different symbols. You also can insert separator lines between list items.

You can [use a Property Node](#) to modify list items and to gather information about list items, such as to detect the currently selected items or to set the handling of upper and lowercase characters when you type characters in a listbox at run time.

Refer to the following VIs for examples of using a listbox:

- Listbox VI: labview\examples\Controls and Indicators\Listbox

Tree Controls

Use the tree control to give users a hierarchical list of items from which to select. You organize the items you enter in the tree control into groups of items, or nodes. Click the expand symbol next to a node to expand it and display all the items in that node.

When you type characters in a tree control at run time, LabVIEW selects the first item in the tree control that begins with the characters you type. You can change the hierarchy of items in the tree control by selecting an item and pressing the period . key to indent the current item or the comma , key to move the current item to the left. Right-click a tree control and select **Selection Mode»Highlight Entire Row** from the shortcut menu to highlight the entire row when you select an item.

You configure items in a tree control the same way you configure them in a listbox. You also can change the type of symbol that appears next to each node, and you can configure whether the user can *drag and drop* items within the tree control.

You can use an *Invoke Node* to modify items in the tree control and to gather information about items, such as to detect which items, if any, the user double-clicked. When you add an item in the tree control, LabVIEW creates a unique tag for the item. You use this tag to modify items or to gather information about items programmatically. Right-click the tree control and select **Edit Items** from the shortcut menu to modify the tags that LabVIEW creates for each item.

Tables

Use the table control to create a *table* on the front panel.

.NET and ActiveX Controls and Indicators



NOTE

The following content applies to .NET Framework only. LabVIEW has been gradually adding support for .NET Core since LabVIEW 2024 Q3. Refer to *LabVIEW New Features and Changes* for specific .NET Core support added in different releases.

Use the *.NET* and *ActiveX* controls located on the **.NET & ActiveX** palette to manipulate common .NET or ActiveX controls, such as rich text boxes, picture boxes, and calendars. You can add additional .NET or ActiveX controls to this palette for later use. Select **Tools»Import» .NET Controls to Palette** or **ActiveX Controls to Palette** to convert a set of .NET or ActiveX controls, respectively, to custom controls and add them to the **.NET & ActiveX** palette.



NOTE

Creating and communicating with .NET Framework objects requires the .NET Framework 4.0. NI strongly recommends that you use .NET Framework objects only in *LabVIEW projects*.

Numeric Controls and Indicators

Use numeric controls and indicators on the front panel to enter and display numeric data in LabVIEW applications. The following list describes the common use cases of numeric controls and indicators available on the Modern, Silver, System, or Classic subpalettes:



NOTE

The availability of numeric controls and indicators may vary depending on the subpalette style you choose. Refer to the Modern, Silver, System, or Classic subpalette for numeric control and indicator availability.

Numeric Control or Indicator	Description
Numerics	Use to enter or display numeric data.
Time Stamps	Use to send or retrieve a time and date value.
Fill Slides	Use to display numeric data in a vertical or horizontal slide with a customizable scale.
Pointer Slides	Use to display numeric data in a vertical or horizontal slide with a customizable scale and a pointer that helps you see the exact value.
Progress Bars	Use to show progress in a vertical or horizontal bar.
Graduated Bars	Use to show progress in a vertical or horizontal bar with customizable dividers to help you break up the value into equal parts. For example, you can show that a value has reached 25% of the target value.
Knobs, Dials, Meters, and Gauges	Use to enter or display numeric data in a rotary action.
Tanks and Thermometers	Use to display numeric data in a vertical slide that resembles a real tank or thermometer instrument.
Scrollbars	Use to display numeric data in a vertical slide that scrolls through data by dragging the square scroll box, scrolling the mouse wheel, clicking the increment and decrement arrows, or clicking empty space in the scrollbar.
Color Boxes	Use to display a color that corresponds to a specified value. For example, you can use color boxes to indicate different conditions, such as out-of-range values.
Color Ramps	Use to display numeric data with colors. As the input value changes, the display changes to a previously chosen color for the value.
Waveforms	Use to enter or display numeric data in a waveform.

**TIP**

Replace an existing value in a numeric control or indicator with a new value by pressing the <Enter> key, clicking outside the digital display window, or clicking the **Enter** button. While the VI runs, LabVIEW waits until you perform one of these actions to interpret the new values. For example, while changing a value in the digital display to 135, you do not want the VI to receive the values 1 and 13 before 135.

**NOTE**

To retain the floating-point representation, LabVIEW adds any necessary prefixes to the unit string when you format a numeric control or indicator in an engineering or System International (SI) notation and do not have a complex representation.

Related Information

[Changing Numeric Representation](#)

[Changing Data Ranges of Numeric Controls and Indicators](#)

[Changing Format and Precision of Numeric Controls and Indicators](#)

[Configuring the Time and Date in a Time Stamp Control](#)

[Changing Values of Numeric Objects with a Scale](#)

Refnum Controls and Indicators

Use the reference number controls located on the **Refnum** and **Classic Refnum** palettes to work with files, directories, devices, and network connections. Use the [control refnum](#) to pass front panel object information to subVIs.

A reference number, or refnum, is a unique identifier for an object, such as a file, device, or network connection. When you open a file, device, or network connection, LabVIEW creates a refnum associated with that file, device, or network connection. All operations you perform on open files, devices, or network connections use the refnums to identify each object. Use a refnum control to pass a refnum into or out of a VI. For example, use a refnum control to modify the contents of the file that a refnum is referencing without closing and reopening the file.

Because a refnum is a temporary pointer to an open object, it is valid only for the period during which the object is open. If you close the object, LabVIEW disassociates the refnum from the object, and the refnum becomes obsolete. If you open the object again, LabVIEW creates a new refnum that is different from the first refnum. LabVIEW allocates memory for an object that is associated with a refnum. Close the refnum to release the object from memory.

LabVIEW remembers information associated with each refnum, such as the current location for reading from or writing to the object and the degree of user access, so you can perform concurrent but independent operations on a single object. If a VI opens an object

multiple times, each open operation returns a different refnum. LabVIEW automatically closes refnums for you when a VI finishes running, but it is a good programming practice to close refnums as soon as you are finished with them to most efficiently use memory and other resources. Close refnums in the opposite order that you opened them. For example, if you obtain a refnum to object A and invoke a method on object A to obtain a refnum to object B, close the refnum to object B first and then close the refnum to object A.

If you open a refnum inside a For Loop or While Loop, close that refnum for each iteration of the loop because LabVIEW repeatedly allocates memory for the refnum and does not free the memory until the VI finishes running.

Ring and Enumerated Type Controls and Indicators

Use the ring and enumerated type controls and indicators located on the **Ring & Enum** and **Classic Ring & Enum** palettes to create a list of strings you can cycle through.

Ring Controls

Ring controls are numeric objects that associate numeric values with strings or pictures. Ring controls appear as pull-down menus that users can cycle through to make selections.

Ring controls are useful for selecting mutually exclusive items, such as trigger modes. For example, use a ring control for users to select from continuous, single, and external triggering.

Right-click a ring control and select **Edit Items** from the shortcut menu to add items to the ring control list in the control. The order of the items in the **Edit Items** page of the **Ring Properties** dialog box determines the order of the items in the control. You also can [configure the ring control](#) so users can enter numeric values not already associated with any entries in the list of items defined for the control.

To enter an undefined value in the ring control at run time, click the control, select **<Other>** from the shortcut menu, enter a numeric value in the digital display that appears, and press the <Enter> key. The undefined value appears in the ring control in angle brackets. LabVIEW does not add the undefined value to the ring control list.

When you configure the list of items for a ring control, you can [assign a specific numeric value](#) to each item. If you do not assign specific numeric values to the items, LabVIEW assigns sequential values that correspond to the order of the items in the list, starting with a value of 0 for the first item.

Enumerated Type Controls

Use enumerated type controls to give users a list of items from which to select. An enumerated type control, or enum, is similar to a text or menu ring control in that both contain lists of strings through which you can cycle. However, enumerated type controls differ from ring controls in the following ways:

- The data type of an enumerated type control includes information about the numeric values in the control and their corresponding strings. The data type of a ring control is numeric only.
- The numeric representation of the enumerated type control is an 8-, 16-, or 32-bit unsigned integer, while ring controls can have additional representations. Right-click either type of control and select **Representation** from the shortcut menu to change the representation of the control.
- You cannot allow the user to enter undefined values in enumerated type controls, and you cannot assign specific numeric values to items in enumerated type controls. If you need this functionality, use a ring control.
- You can edit the items of an enumerated type control only at edit time, while you can edit ring control items at run time using a Property Node.
- When you wire an enumerated type control to the selector terminal of a Case structure, LabVIEW matches the cases to the string values of items in the control, not the numeric values. When you use a ring control with a Case structure, LabVIEW matches cases to the numeric values of items.
- When you wire an enumerated type control to the selector terminal of a Case structure, you can right-click the structure and select **Add Case for Every Value** to create a case for the string value of each item in the control. However, when you wire a ring control to the selector terminal of a Case structure, you must manually enter each case.

All arithmetic functions except Increment and Decrement treat the enumerated type control the same as an unsigned integer. Increment changes the last enumerated value to the first value, and Decrement changes the first enumerated value to the last value. When coercing a signed integer to an enumerated type, negative numbers change to equal the first enumerated value, and out-of-range positive numbers change to equal the last enumerated value. Out-of-range unsigned integers always change to equal the last enumerated value.

If you wire a floating-point value to an enumerated type indicator, LabVIEW coerces the floating-point value to the closest numeric value in the enumerated type indicator. LabVIEW handles out-of-range numbers as previously described. If you wire an enumerated control to any numeric value, LabVIEW coerces the enumerated type value to a numeric value. To wire an enumerated type control to an enumerated type indicator, the items in the indicator must match the items in the control. However, the indicator can have more items than the control.



TIP

When using enumerated type controls, always *make a type definition* of the control. Creating type definitions prevents you from needing to rewrite the code each time you add or remove an item from an enumerated type control.

String and Path Controls and Indicators

Use string controls and indicators to enter or display text. Use path controls and indicators to enter or display file or folder paths.

Pass new or changed values to the block diagram from a string or path control during run time by clicking outside the text field. By default, LabVIEW waits until you perform this action to interpret the new values. You can update the string value as you type by right-clicking a string control and selecting **Update Value while Typing** from the shortcut menu. You can also use the Update While Typing? property to update the value programmatically.



NOTE

The availability of string and path controls and indicators may vary depending on the subpalette style you choose. Refer to the Modern, Silver, System, or Classic subpalette for string and path control and indicator availability.

String Controls and Indicators

Right-click a string control or indicator to change the string display type and behavior, such as password display or hex display. You can also use VIs and functions from the **File I/O** palette to pass strings to an external file, such as a text file or spreadsheet.

Related Information

[Configuring String Controls and Indicators](#)

[Formatting Strings](#)

[Grouping Data with Strings](#)

Combo Box Controls

Use combo box controls to create a list of strings you can cycle through, such as a pull-down menu. Combo box controls are different from ring and enum controls because they output data as strings instead of numerics.



NOTE

As you type a string in a combo box control at run time, LabVIEW selects the first, shortest string in the control that begins with the letters you type. If no strings match the letters you type and the control does not allow undefined string values, LabVIEW does not accept or display the letters you type in the control.



TIP

Press the <Up> and <Down> arrow keys to cycle through combo box values.

Related Information

[Editing Items in Combo Box Controls](#)

[Specifying Custom Values for Combo Box Strings](#)

[Selecting and Entering Strings in Combo Box Controls at Run Time](#)

Path Controls and Indicators

You can drag a path, folder, or file from (Windows) Windows Explorer or (macOS) the Finder and place it in a path control if dropping is enabled.



NOTE

Dropping is enabled by default. To disable dropping, right-click the path control and select **Advanced»Allow Dropping** from the shortcut menu.

Path controls and indicators work similarly to string controls and indicators, but LabVIEW formats the path using the standard syntax for the platform you are using.

Invalid Paths

If a function that returns a path fails, the function returns <Not A Path> in the indicator. Use <Not A Path> as the default value for a path control to detect when the user fails to provide a path and display a file dialog box with options for selecting a path. Use the File Dialog Express VI to display a file dialog box.

Empty Paths

Use empty paths to prompt the user to specify a path. An empty path appears as an empty string on Windows and macOS 32-bit and as a slash (/) on Linux and macOS 64-bit. (Windows) When you wire an empty path to a file input/output (I/O) function, the empty path refers to the list of drives mapped to the computer. (macOS) The empty path refers to the mounted volumes. (Linux) The empty path refers to the root directory.

Absolute and Relative Paths

A relative path describes the location of a file or directory relative to an arbitrary location in the file system. An absolute path describes the location of a file or directory starting from the top level of the file system. Use relative paths in VIs to avoid having to rework the paths if you build an application or run the VI on a different computer.

Related Information

[File Paths](#)

[Configuring Browse Options of Path Controls](#)

[Changing the Value of a Path Control](#)

Variant and Class Controls and Indicators

Use the variant controls and indicators and class controls and indicators on the **Variant & Class** palette to interact with variant and class data.

LabVIEW Object

The LabVIEW Object is the common ancestor data type for all [LabVIEW classes](#). You can use the LabVIEW Object [to create generic methods](#) that can handle all LabVIEW class data types.

Creating Custom Controls, Indicators, and Type Definitions

Use custom controls and indicators to extend the available set of front panel objects. You can create custom user interface components for an application that vary cosmetically from built-in LabVIEW controls and indicators. You can save a custom control or indicator you created in a directory or LLB and use the custom control or indicator on other front panels. You also can create an icon for the custom control or indicator and add it to the Controls palette. Refer to the [caveats and recommendations](#) before you begin creating custom controls and indicators.

Use type definitions and strict type definitions to link all the instances of a custom control or indicator to a saved custom control or indicator file. You can make changes to all instances of the custom control or indicator by editing only the saved custom control or indicator file, which is useful if you use the same custom control or indicator in several VIs.



NOTE

You also can use [XControls](#) to design and create complex controls and indicators in LabVIEW. Unlike custom controls, XControls have dynamic run-time and edit-time behavior that is defined by VIs that run in the background. You can create and edit XControls only in the LabVIEW Professional Development System. If a VI contains an XControl, you can run the VI in all LabVIEW packages.

Using the Control Editor Window

Use the Control Editor window to customize controls and indicators and to [define the private data control](#) for a LabVIEW class. For example, you can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.

You can display the Control Editor window in the following ways:

- Right-click a control or indicator on the front panel and select **Advanced»Customize** from the shortcut menu.
- Use the Positioning tool to select a control or indicator on the front panel and select **Edit»Customize Control**.
- [Use](#) the New dialog box.
- Right-click an existing control or control template and select **Open**.
- Right-click **My Computer** in the Project Explorer window and select **New»Type Definition** from the shortcut menu.
- [Create a LabVIEW class](#) and double-click the private data control.

Note If you are defining the class private data control, the Control Editor window opens with the **Cluster of class private data** on the block diagram automatically. Also, LabVIEW defaults the ring control of the Control Editor window to **Class Private Data** and you cannot change it.

- Right-click the instance of a type definition or strict type definition from the front panel or block diagram and select **Open Type Def.** from the shortcut menu.

Even though the Control Editor window looks like a front panel, you use it only for customizing a single control or indicator. For a LabVIEW class this single control must be a cluster. The Control Editor window does not have a block diagram and cannot run.

The Control Editor window can contain only one control or indicator. You can customize an array or cluster control or indicator that contains other sub-controls or sub-indicators. The array or cluster itself is the single control. The **Invalid Control** button  appears on the Control Editor window toolbar while you move controls or indicators in and out of an array or cluster. Click the **Invalid Control** button to display an explanation of the error in the Error list window.

A mode indicator button on the Control Editor window toolbar shows whether you are in edit mode  or in customize mode . Click this button to toggle between the two modes. You also can switch between modes by selecting **Operate»Change to Customize Mode** or **Operate»Change to Edit Mode**. The Control Editor window opens in edit mode.

Use edit mode to change the size or color of a control or indicator and to select options from its shortcut menu, just as you do in edit mode on a front panel.

Use customize mode to make extensive changes to controls or indicators by changing the individual parts of a control or indicator.

Customize Mode Shortcut Menus

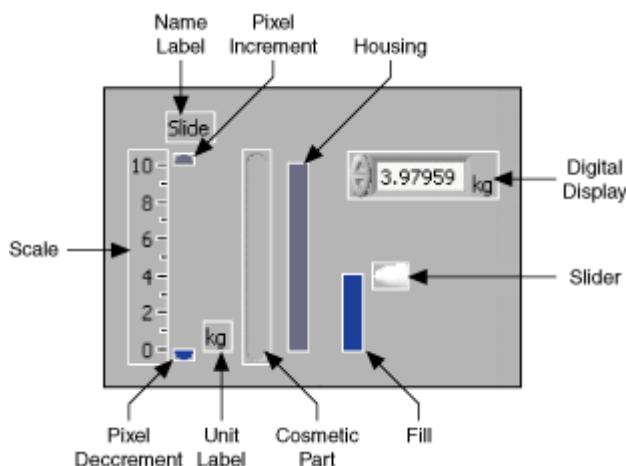
In customize mode, a shortcut menu for each part replaces the shortcut menu for the control or indicator as a whole. Different parts have different shortcut menus. Use the shortcut menus to perform the following operations:

- Customize *cosmetic parts*, such as objects from the **Decorations** palette added to a custom control or indicator.
- Customize text parts, such as the name label.
- Customize *controls or indicators as parts*, such as the numeric control used for the digital display of a slide control. Knobs, meters, and charts also use a numeric control for a digital display. Some controls are more complicated. For example, the graph uses a tree control for its cursor display part.

Customizing Individual Parts of Controls and Indicators

Independent Parts of Controls or Indicators

All controls and indicators are built from smaller parts. For example, the following illustration shows the individual parts of a slide control.



When you switch to customize mode in the Control Editor window, the parts of the control or indicator become independent. You can make changes to each part without affecting any other part. Customize mode shows all parts of the control or indicator, including any parts that are hidden in edit mode, such as the name label or the radix on a numeric control. Because the parts of the control or indicator are detached from each other, you cannot operate or change the value of the control or indicator while in customize mode.

In customize mode, you can use the Positioning tool to select two or more parts of the control or indicator and rearrange the parts by using the [Align Objects](#), [Distribute Objects](#), and [Reorder](#) pull-down menus on the toolbar.

Control Parts Window

Select **Window»Show Parts Window** to display the Control Parts window. Use the **Control Parts** window to size and position the individual parts of a control or indicator. The **Control Parts** window identifies the parts of the control or indicator and shows the exact position and size of each part in pixels.

The **Current Part** section of the **Control Parts** window contains an image and the name of the part you selected in the Control Editor window. Click the **Current Part** display to display and select among all the parts of the control or indicator. Click the **Current Part** increment or decrement arrow to scroll through the parts of the control or indicator.

When you move or resize a part in the Control Editor window, LabVIEW updates the position and size in the **Current Part** and **Part Position and Dimensions** sections of the **Control Parts** window. You also can enter the position and size values directly in the **Part Position and Dimensions** section to move or resize the part in the Control Editor window, which is useful when you must make two parts exactly the same size or align one part with another. After entering position and size values in the **Part Position and Dimensions** section, press the <Enter> key or click the left mouse button for the changes to take effect.

Adding Cosmetic Parts to Custom Controls and Indicators

You can use the Control Editor window to add graphics, text, and decorations to a custom control or indicator in either edit mode or customize mode. If you paste a graphic or text from the clipboard, create a label using the Labeling tool, or select a decoration from the Decorations palette, the object becomes a new decoration part of the control or indicator and appears with the control or indicator when you place it on a front panel. In the Control Editor window, you can move, resize, change the layering order, or replace the new decoration part. When you use a custom control or indicator on a front panel, you can change the size of any decoration parts you added. However, you cannot move the added decoration parts.

Customize Mode Shortcut Menu Items for Cosmetic Parts

Cosmetic parts can have a single graphic, multiple graphics that display at different times, and multiple graphics that are independent from each other. To customize cosmetic parts, place the Control Editor window in customize mode and right-click the cosmetic part you want to customize. The shortcut menu items available depend on the type of cosmetic part.

Cosmetic Parts with a Single Graphic

The shortcut menu for cosmetic parts with a single graphic contains the following items:

- **Copy to Clipboard** —Places a copy of the graphic of the part on the clipboard. After you copy a shape to the clipboard, right-click a part and select **Import Picture from Clipboard** from the shortcut menu to import the shape into the part. When you need simple shapes for customizing cosmetic parts, consider copying the shapes from built-in parts of other controls or indicators instead of creating them in a graphics application. For example, if you need a simple rectangular shape, right-click the housing and select **Copy to Clipboard** from the shortcut menu. Graphics taken from built-in parts or decorations resize better than graphics made in a graphics application. For example, when you resize a rectangle drawn in a graphics application, the rectangle grows uniformly, enlarging its area and making its border thicker. When you resize a rectangle copied from a part of a built-in control or indicator, the rectangle keeps the same thin border. Parts copied from a built-in control or indicator have a similar appearance on color monitors and on black-and-white monitors. In addition, you can use the Coloring tool to add color to graphics you copy from built-in parts and decorations. You cannot change the color of a graphic you import from another source because the color is part of the definition of the graphic.
- **Import Picture from Clipboard** —Replaces the current graphic of a cosmetic part with the graphic currently on the clipboard. For example, you can import graphics of an open and closed valve for a Boolean switch. If the clipboard does not contain a graphic, the **Import Picture from Clipboard** item is not available.
- **Import from Clipboard at Same Size** —Replaces the current graphic with the graphic currently on the clipboard and keeps the original size of the part, shrinking or enlarging the clipboard graphic to fit. If the clipboard does not contain a graphic, the **Import from Clipboard at Same Size** item is not available.
- **Import from File** —Replaces the current graphic of a cosmetic part with the graphic you select in the file dialog box.

- **Import from File at Same Size** —Replaces the current graphic with the graphic you select in the file dialog box and keeps the original size of the part, shrinking or enlarging the graphic from file to fit.
- **Revert** —Restores a part to its original appearance without changing the position of the part. If you display the Control Editor window for a front panel control or indicator and make changes to a part, selecting **Revert** from the shortcut menu changes the part back to the way it looks on the front panel. If you open a custom control or indicator file in the Control Editor window, the **Revert** item is not available.
- **Original Size** —Restores the graphic of a part to its original size, which is useful for graphics you import from other applications and then resize. Some graphics imported from other applications do not look as good as the original when you resize them. You might want to restore the imported graphic to its original size to improve the appearance of the graphic. If you do not import a graphic, the **Original Size** item is not available.



Note You also can replace a decoration with an image from the clipboard by selecting the decoration and selecting **Edit»Paste**. To replace a decoration with an image from a file, right-click the decoration and select **Import from File** from the shortcut menu.

Cosmetic Parts with Multiple Dependent Graphics

Some cosmetic parts have multiple dependent graphics that display at different times. The different graphics are the same size and use the same colors. When you move or resize the current graphic of the cosmetic part, its other graphics also move the same amount or change size proportionally. For example, a Boolean switch has four different graphics. The first graphic shows the FALSE state. The second graphic shows the TRUE state. The third graphic shows the *Switch When Released state*, which is the TRUE to FALSE transition state. The fourth graphic shows the Latch When Released state, which is the FALSE to TRUE transition state. When the Boolean control is in the Switch When Released state or the Latch When Released state, the value of the Boolean control does not change until you release the mouse button. Between the time you click the mouse button and the time you release the mouse button, the Boolean control shows the third or fourth graphic as a transition state.

The shortcut menu for cosmetic parts with multiple dependent graphics contains all the items available for cosmetic parts with a single graphic plus the **Picture Item** item. Selecting **Picture Item** from the shortcut menu displays all the graphics that belong to a cosmetic part. The current graphic item has a dark border around it. When you import a graphic, you change only the current graphic item. To import a graphic for one of the other graphic items, right-click the part, select **Picture Item** from the shortcut menu, select the graphic item, and import the new graphic.

Cosmetic Parts with Multiple Independent Graphics

A cosmetic part with more than one graphic can have graphics of different sizes that each use different colors. For example, the slide uses two graphics of different sizes to show which slider is active on a multivalued slide. The shortcut menu for cosmetic parts with multiple independent graphics contains all the items available for cosmetic parts with

single graphics and multiple dependent graphics plus the **Independent Sizes** item. Select **Independent Sizes** from the shortcut menu to place a checkmark next to it so you can move and resize each graphic individually without changing the other graphics of the cosmetic part.

Text Parts

A text part is a graphic with text, such as a label. The shortcut menu for a text part contains all the items available for cosmetic parts with a single graphic and the items available for text elements on a front panel in edit mode. The **Control Parts** window shows only the background graphic for the text part, not the text itself. You can customize the background graphic but not the text.

Edit Mode Option for Boolean Controls and Indicators

You can import graphics into Boolean controls and indicators when the Control Editor window is in edit mode, unlike other controls and indicators, into which you can import graphics only when the Control Editor window is in customize mode. When the Control Editor window is in edit mode and you right-click a Boolean control or indicator, the shortcut menu that appears contains all the items available for Boolean controls and indicators on a front panel in edit mode, as well as the **Import Picture from Clipboard** item.

Linking Custom Controls and Indicators to a Saved Custom Control or Indicator File

When you place a custom control or indicator in a VI, no connection exists between the custom control or indicator you saved and the instance of the custom control or indicator in the VI. Each instance of a custom control or indicator is a separate, independent copy. Therefore, changes you make to a custom control or indicator file do not affect VIs already using that custom control or indicator. If you want to link instances of a custom control or indicator to the custom control or indicator file, save the custom control or indicator as a [type definition or strict type definition](#). All instances of a type definition or a strict type definition link to the original file from which you created them.

When you save a custom control or indicator as a type definition or strict type definition, any data type changes you make to the type definition or strict type definition affect all instances of the type definition or strict type definition in all the VIs that use it. Also, cosmetic changes you make to a strict type definition affect all instances of the strict type definition on the front panel.

Type Definitions and Strict Type Definitions

Use type definitions and strict type definitions to link all the instances of a custom control or indicator to a saved custom control or indicator file. You can make changes to all instances of the custom control or indicator by editing only the saved custom control or indicator file, which is valuable if you use the same custom control or indicator in several VIs.

Type Definitions

Type definitions identify the correct data type for each instance of a custom control or indicator. When the data type of a type definition changes, all instances of the type definition automatically update. In other words, the data type of the instances of the type definition change in each VI where the type definition is used. However, because type definitions identify only the data type, only the values that are part of the data type update. For example, on numeric controls, the data range is not part of the data type. Therefore, type definitions for numeric controls do not define the data range for the instances of the type definitions. Also, because the item names in ring controls do not define the data type, changes to ring control item names in a type definition do not change the item names in instances of the type definition. However, if you change the item names in the type definition for an enumerated type control, the instances update because the item names are part of the data type. An instance of a type definition can have its own unique caption, label, description, tip strip, default value, size, color, or style of control or indicator, such as a knob instead of a slide.

If you change the data type in a type definition, LabVIEW converts the old default value in instances of the type definition to the new data type, if possible. LabVIEW cannot preserve the instance default value if the data type changes to an incompatible type, such as replacing a numeric control or indicator with a string control or indicator. When the data type of a type definition changes to a data type incompatible with the previous type definition, LabVIEW sets the default value of instances to the default value you specify in the `.ctl` file. If you do not specify a default value, LabVIEW uses the default value for the data type. For example, if you change a type definition from a numeric to a string type, LabVIEW replaces any default values associated with the old numeric data type with empty strings.

Strict Type Definitions

A strict type definition forces everything about an instance to be identical to the strict type definition, except the caption, label, description, tip strip, and default value. As with type definitions, the data type of a strict type definition remains the same everywhere you use the strict type definition. Strict type definitions also define other values, such as range checking on numeric controls and the item names in ring controls. The only VI Server properties available for strict type definitions are those that affect the appearance of the control or indicator, such as Visible, Disabled, Key Focus, Blinking, Position, and Bounds.



Note If you link a constant to a strict type definition, the constant will instead behave as though the link is not strict. Strict type definitions require that all instances of a type definition maintain identical cosmetic attributes. However, most cosmetic attributes are invalid on the block diagram. Because constants are only found on the block diagram, they do not maintain cosmetic attributes.

You cannot prevent an instance of a strict type definition from automatically updating unless you [remove the link](#) between the instance and the strict type definition.

Creating and Editing Type Definitions and Strict Type Definitions

You can [create and edit a type definition or strict type definition](#) from the front panel or block diagram. Use the Control Editor window and the same techniques that you use to

customize other controls and indicators to customize type definitions and strict type definitions.

Preventing an Instance of a Type Definition from Automatically Updating

You can prevent an individual instance of a type definition from automatically updating when the data type of the type definition changes by right-clicking the instance and removing the checkmark next to the **Auto-Update from Type Def.** shortcut menu item. Removing the checkmark from the **Auto-Update from Type Def.** item prevents the instance from automatically updating when the data type of the type definition changes but preserves the link between the instance and the type definition. You can remove the link between an instance and a type definition by right-clicking the instance and selecting **Disconnect from Type Def.** from the shortcut menu.



Note You cannot prevent an instance of a strict type definition from automatically updating unless you *remove the link* between the instance and the strict type definition.

If you disable automatic updating for an instance of a type definition and change the data type of the instance or change the data type of the type definition, the instance appears dimmed, and the **Run** button appears broken. To *update the instance manually*, right-click the instance and select **Review and Update from Type Def.** to display the Review and Update from Type Def. dialog box. To enable the application to run without updating the instance, right-click the instance and select **Disconnect from Type Def.** from the shortcut menu to permanently *remove the link* between the instance and the type definition.

Updating Type Definition Instances Manually

If an instance of a type definition appears dimmed, you must *update the instance manually* using the Review and Update from Type Def. dialog box.

When you change the data type of a type definition, LabVIEW automatically attempts to update all instances of the type definition. If LabVIEW is unable to map the previous default values to the updated controls or constants when updating instances, LabVIEW places those instances in an unresolved state. An unresolved instance appears dimmed, and the **Run** button of the VI in which the instance exists appears broken. To resolve the data mapping discrepancies, you must manually update the instances. Alternatively, you can remove the link between unresolved instances and the type definition to enable the application to run.

Removing the Link between an Instance and a Type Definition or Strict Type Definition

You can permanently remove the link between an instance of a type definition or a strict type definition and the type definition or strict type definition by right-clicking the instance and selecting **Disconnect from Type Def.** from the shortcut menu. Disconnecting from the type definition or strict type definition removes the restrictions on the data type of the instance, making the instance a control or indicator.

Unavailable Type Definitions and Strict Type Definitions

For each instance of a type definition or strict type definition, the VI containing the instance maintains a connection to the file where you saved the type definition or strict type definition. For a VI that contains an instance of a type definition or strict type definition to run, the VI must have the file that contains the type definition or strict type definition available. If you open a VI and LabVIEW cannot find a type definition or strict type definition the VI needs, LabVIEW disables the instances of that type definition or strict type definition in the VI, and the **Run** button appears broken. To allow the VI to run, find and open the correct type definition or strict type definition. You also can right-click the disabled instance and select **Disconnect From Type Def.** from the shortcut menu to permanently remove the link between the instance and the type definition or strict type definition.

Using Type Definitions and Strict Type Definitions

You place type definitions and strict type definitions on the front panel or block diagram the same way you place any custom control or indicator on the front panel or block diagram. When you right-click an instance of a type definition or a strict type definition, the shortcut menu includes type definition options. You can open the type definition or strict type definition from the front panel or from the block diagram by right-clicking the instance and selecting **Open Type Def.** from the shortcut menu.

You can identify the instance of a type definition or strict type definition on the block diagram with a glyph marking the upper left corner of the instance . If you hover the cursor over the glyph, a tip strip displays the name of the type definition or strict type definition. Changes you make to type definitions affect instances placed on block diagrams as constants in the same way the changes affect front panel instances.



Note If you change a strict type definition, instances placed on block diagrams as constants act as if the link is not strict, and LabVIEW updates the constant only with data type changes. For example, if you update the string values in a *ring control* saved as a strict type definition, LabVIEW does not update instances of the strict type definition placed on the block diagram because the string values are not part of the data type of the ring control.

Using Cluster Type Definitions and Cluster Strict Type Definitions

If you place an instance of a cluster type definition or cluster strict type definition on a front panel, use the Bundle By Name function and the Unbundle By Name function on the block diagram to access the elements of the cluster. Unlike the Bundle function and the Unbundle function, which reference cluster elements by cluster order, the Bundle By Name function and the Unbundle By Name function reference elements of the cluster by name. Therefore, reordering the elements or adding new elements to the cluster type definition or cluster strict type definition does not break the VI.

If you delete an element from the cluster type definition or cluster strict type definition wired to the Bundle by Name function or the Unbundle By Name function, you must change the block diagram because the reference to the missing element becomes invalid. Invalid names in the Bundle by Name function and the Unbundle by Name function appear

black. Click the Bundle by Name function or the Unbundle By Name function to display a list of valid names from which to select.

When you create references to controls in a cluster type definition or cluster strict type definition, LabVIEW uses the cluster order to determine which controls the references point to in the cluster. Because of this behavior, if you delete a control from the cluster type definition or cluster strict type definition, the remaining reference that points to the deleted control may now point to a different control in the cluster. If you delete a control in a cluster type definition or cluster strict type definition, make sure that each reference still points to the correct control in the cluster.

XControls in LabVIEW

Using XControls

Use XControls to design and create complex controls and indicators in LabVIEW.



NOTE

You can create and edit XControls only in the LabVIEW Professional Development System. If a VI contains an XControl, you can run the VI in all LabVIEW packages.

When you create an XControl, you combine built-in LabVIEW controls and indicators to take advantage of the combined functionality of the built-in LabVIEW controls and indicators. Unlike [custom controls](#), which you use to create custom user interface components that vary cosmetically from built-in LabVIEW controls and indicators, XControls have dynamic run-time and edit-time behavior that is defined by VIs that run in the background. When you use an XControl in a VI, the block diagram for that VI is simplified because the XControl includes the behavior of the control.

When you [create an XControl](#), you use the [Project Explorer](#) window to edit the XControl library. The XControl library contains all of the components of an XControl. A typical XControl consists of abilities, properties, and methods. XControl [abilities](#) are components of the XControl that the XControl requires to function properly. Required abilities define the appearance, data type, display state, and initialization of the XControl. An XControl can have additional, optional abilities. The Select Ability dialog box lists and describes all required and optional abilities that are applicable to the XControl. The XControl [properties](#) and [methods](#) allow the user of an XControl to configure the control programmatically. Creating properties and methods for an XControl is optional.

Refer to the Simple Dual Mode Thermometer XControl project in the `labview\examples\Controls and Indicators\XControls\Dual Mode Thermometer` directory for an example of using XControls. Find related examples

Using XControl Abilities

[XControls](#) abilities are components of the XControl that the XControl requires to function properly. Abilities are represented by VIs or controls `.ctl`s that LabVIEW calls at the

appropriate time. Required abilities define the appearance, data type, display state, and initialization of the XControl. XControls include the following required abilities:

- **Data** —Data is a *type definition* VI that specifies the data type of the XControl. Double-click the Data ability in the XControl library to open and edit the Data type definition. By default, the Data type definition includes a numeric control. Replace the existing control in the VI with a new control that represents the data type of the XControl. Save the Data type definition after you make the changes.
- **State** —State, also known as display state, is a *type definition* VI that specifies any information other than the data of an XControl that affects the appearance of the control. At any point in time, the XControl uses the Data and State abilities to update its appearance. Double-click the State ability in the XControl library to open and edit the State type definition. By default, the State type definition includes a cluster with a numeric control. Replace the existing control in the VI with a new control that represents the display state of the XControl. Save the State type definition after you make the changes.



NOTE

When the type of display state changes, use the XControl Properties dialog box to update the version of the XControl. If the version has changed, you can use Init to update the State ability.

- **Facade** —Facade defines the appearance of the XControl. The Facade ability is represented by the Facade VI. The front panel of the Facade VI defines the appearance of the XControl. The block diagram of the Facade VI describes the behavior of the XControl. Double-click the Facade ability in the XControl library to open and *edit the Facade VI*. Save the Facade VI after you make the changes.
- **Init** —LabVIEW calls the Init ability when the XControl is first placed on a front panel or when a VI that contains the XControl is loaded into memory. Use this ability to initialize the display state before the XControl is displayed. The Init ability also plays an important role in updating previous versions of the XControl to the latest version. When you load a VI that contains an XControl, LabVIEW calls Init and checks if the version of the XControl has changed since the last time the VI was saved. If the version has changed, you can use Init to update the State ability. Init includes **Previous Version** and **Previous State** controls and a **Current State** indicator. When the version changes, Init converts the value of **Previous State**, which is passed in as a variant, to the new state format. The new state is passed to the **Current State** indicator.

An XControl can have additional, optional abilities. The Select Ability dialog box lists and describes all required and optional abilities that are applicable to the XControl.

Using the XControl Facade Ability

The Facade *ability* defines the appearance of an *XControl*. The Facade ability is represented by the Facade VI. Because the XControl requires the Facade ability, LabVIEW automatically adds the Facade VI to the XControl library when you create an XControl. The Facade VI must be a *reentrant* VI.

Editing the Facade VI

You add LabVIEW controls and indicators to the Facade VI to create the front panel of the XControl. By default, the Facade VI includes data, display state, and container state controls and data, display state, and action indicators. These controls and indicators are hidden and are not part of the XControl appearance. Resize the front panel window to fit only the controls and indicators that make up the appearance of the XControl. Set the [window appearance properties](#) of the Facade VI to adjust the [size](#) and [appearance](#) of the XControl.

The block diagram of the Facade VI handles events for the front panel controls and the Facade VI itself. It contains a While Loop with an Event structure in it. The Facade VI is not a continuously running VI. LabVIEW calls the Facade VI only when there is an event pending for the VI to handle. LabVIEW calls the Facade VI with the current data, display state, and container state. The VI handles the event and returns the updated data, display state, and type of action performed.

Use the **Container State** cluster to provide information about the instance of the XControl. The **Container State** cluster includes three elements— **Indicator?**, **Run Mode?**, and **Refnum**. Use **Indicator?** to determine if the XControl instance is a control or indicator. Use **Run Mode?** to determine if the VI you are using the XControl in is running or idle. **Run Mode?** is set to TRUE when the owning VI is running. **Refnum** is a reference to the XControl instance that you use only to query information about the XControl. Do not use **Refnum** to modify the XControl instance from the Facade VI.

When the Facade VI changes the data or display state, it should update the **Action** cluster and inform LabVIEW of the changes. The **Action** cluster contains three elements— **Data changed**, **State changed**, and **Action name**. Set **Data changed** to TRUE when the Facade VI changes the **Data Out** indicator. Set **State changed** to TRUE when the Facade VI changes the **Display State Out** indicator. Optionally, use **Action name** to name the action being performed. This name appears when you select the **Edit»Undo** option. Typically, the Facade VI changes the data or display state in response to events generated on the front panel controls.

Facade VI Events

The Facade VI also responds to events generated in response to certain user interactions. These events include the Data Change, Display State Change, Direction Change, and Exec State Change events. The Facade VI must handle the Data Change and Display State Change events in order for the XControl to function properly. Handling the Direction Change and Exec State Change events is optional.

LabVIEW calls the Facade VI to handle these events. For example, when you copy an XControl, LabVIEW generates all four events and calls the Facade VI on the new instance of the XControl. LabVIEW also generates all four events and calls the Facade VI when you load the front panel of a VI that contains the XControl. When the VI that contains the XControl switches from edit Mode to run Mode, and vice versa, LabVIEW generates Exec State Change and calls the Facade VI to respond to the VI execution state change.

When you select **Edit»Undo** or **Edit»Redo**, LabVIEW also calls the Facade VI. LabVIEW reverts the display state of the XControl before any action by the Facade VI changes the state. The current display state takes the place of the previous state. LabVIEW generates a Display State Change event and calls the Facade VI to update the XControl after the Undo or Redo action.

Edit each event case to configure how the XControl responds to each event. To create XControls that respond quickly, configure the Facade VI to handle events and finish as quickly as possible. Do not use a Wait function to delay execution in the While Loop within the Event structure.



NOTE

You also can use the Facade VI to *programmatically customize the shortcut menu* for an XControl. You cannot use the Shortcut Menu Editor to customize the shortcut menu for an XControl.

Using XControls in VIs

After you *create an XControl*, you can use it in a VI just as you use a built-in LabVIEW control or indicator. When you use an XControl in a VI, the block diagram for that VI is simplified because the XControl includes the behavior of the control. Place an XControl on the front panel of a VI, click the **Select a Control** icon on the Controls palette, and navigate to the XControl .Xctl you want to place on the front panel.



NOTE

When you place an XControl on the front panel, LabVIEW loads the XControl library. LabVIEW locks the XControl library while the XControl is in use by a VI. You must *unlock the XControl library* if you need to edit it while the XControl is in use by a VI.

After you add the XControl to the VI, use it just as you would a built-in LabVIEW control or indicator. You can *resize* the XControl as a whole to fit the front panel. You can access custom and built-in menu items for the control. You also can programmatically configure the XControl through the custom *properties* and *methods* implemented in the XControl.

Refer to the Run Dual Mode Thermometer XControl VI in the labview\examples\Controls and Indicators\XControls\Dual Mode Thermometer directory for an example of using XControls. Find related examples

Functions in LabVIEW

Functions Overview

Functions are the essential operating elements of LabVIEW. Function icons on the Functions palette have pale yellow backgrounds and black foregrounds. Functions do not have front panels or block diagrams but do have connector panes. You cannot open or edit a function.

The Functions palette also includes the VIs that ship with LabVIEW. *Use built-in VIs* as subVIs in an application to reduce development time.

Adding Terminals to Functions

You can change the number of terminals for some functions. For example, to build an array with 10 elements, you must add 10 terminals to the Build Array function.

You can add terminals to functions by using the Positioning tool to drag the top or bottom borders of the function up or down, respectively. You also can use the Positioning tool to remove terminals from functions, but you cannot remove a terminal that is already wired. You must first delete the existing wire to remove the terminal.

You also can add or remove terminals by right-clicking one of the terminals of the function and selecting **Add Input**, **Add Output**, **Remove Input**, or **Remove Output** from the shortcut menu. Depending on the function, you can add terminals for inputs, outputs, or refnum controls. The **Add Input** and **Add Output** shortcut menu items add a terminal immediately after the terminal you right-clicked. The **Remove Input** and **Remove Output** shortcut menu items remove the terminal you right-clicked. If you use the shortcut menu items to remove a wired terminal, LabVIEW removes the terminal and disconnects the wire.

Built-in Categories

LabVIEW organizes the VIs and functions on the Functions palette using categories. When you install LabVIEW, built-in categories appear on the Functions palette. Some of the built-in categories do not contain objects by default. After you install certain modules, toolkits, and drivers, objects appear in the built-in categories that are empty by default.

LabVIEW also includes categories that are not built in. Categories that are not built in appear only after you install certain modules, toolkits, and drivers.

Click the **Customize** button on the Functions palette and select **Change Visible Palettes** from the shortcut menu to display all built-in categories on the Functions palette. You also can click the **Change Visible Palettes** link at the bottom of a pinned Functions palette to display the built-in categories. The following built-in categories are available on the Functions palette.

- **Programming** —Use the Programming VIs and functions as the basic building blocks for a VI.
- **Measurement I/O** —Use the Measurement I/O VIs and functions to interface with data acquisition devices. This palette displays the VIs and functions for the hardware drivers you install. Most common National Instruments hardware drivers ship with LabVIEW.
- **Instrument I/O** —Use the Instrument I/O VIs and functions to interface with GPIB, serial, modular, PXI, and other types of instruments. This palette displays installed instrument drivers such as GPIB, Serial, and so on.
- **Vision and Motion** —Use the Vision and Motion VIs and functions to interface with any of the NI Vision and Motion products. This palette displays the VIs and functions for the NI products you install, such as NI Vision, NI-Motion, and so on.
- **Mathematics** —Use the Mathematics VIs to perform many different kinds of mathematical analysis. You also can interface real-world measurements to the mathematical algorithms in order to obtain practical solutions.

- **Signal Processing** —Use the Signal Processing VIs to perform signal generation, digital filtering, data windowing, and spectrum analysis.
- **Data Communication** —Use the Data Communication VIs and functions to exchange data between applications.
- **Connectivity** —Use the Connectivity VIs and functions to work with .NET objects, ActiveX-enabled applications, input devices, register addresses, source control, and Windows registry keys.
- **Control & Simulation** —Use the Control & Simulation VIs and functions to design, analyze, simulate, and deploy dynamic system models. This palette contains VIs and functions for NI Control Design and Simulation products such as the Control Design and Simulation Module.
- **Express** —Use the Express VIs and functions to build common measurement tasks.
- **Addons** —Use the Addons category to locate palettes of some modules or toolkits that you have installed in LabVIEW.
- **Favorites** —Use the Favorites category to group together items on the Functions palette that you access frequently. You can [add items](#) to the Favorites category.
- **User Libraries** —Use the User Libraries category to [add VIs](#) to the Functions palette. By default, the User Libraries category does not contain any objects.

Polymorphic Functions

Polymorphism is the ability of VIs and functions to automatically adapt to accept input data of different data types. Functions are polymorphic to varying degrees—none, some, or all of their inputs can be polymorphic. Some function inputs accept numeric values or Boolean values. Some accept numeric values or strings. Some accept not only scalar numeric values but also arrays of numeric values, clusters of numeric values, arrays of clusters of numeric values, and so on. Some accept only one-dimensional arrays although the array elements can be of any type. Some functions accept all types of data, including complex numeric values. You also can create and use [polymorphic units](#).

Numeric Conversion

You can convert any numeric representation to any other numeric representation. When you wire two or more numeric inputs of different representations to a function, the function usually returns the data in the larger, or wider, representation. The functions coerce the smaller representations to the widest representation before execution and LabVIEW places a [coercion dot](#) on the terminal where the conversion takes place.

Some functions, such as Divide, Sine, and Cosine, produce floating-point output by default. If you wire integers to their inputs, these functions convert the integers to double-precision, floating-point numbers before performing the calculation. However, you can [configure most Numeric functions](#) to return a data type you specify.

For floating-point, scalar quantities, it is usually best to use double-precision, floating-point numbers. Single-precision, floating-point numbers save little or no run time and [overflow](#) much more easily. The analysis libraries, for example, use double-precision, floating-point numbers. You should only use extended-precision, floating-point numbers when necessary. The performance and precision of extended-precision arithmetic varies among the platforms.

For integers, it is usually best to use a 32-bit signed integer.

If you wire an output to a destination that has a different numeric representation, LabVIEW converts the data according to the following rules:

- **Signed or unsigned integer to floating-point number** —LabVIEW converts the data to the nearest value possible. This conversion is exact when the floating-point number is more precise than the integer. If the integer is more precise than the floating-point number, LabVIEW maintains as much precision as possible in the conversion.
- **Floating-point number to signed or unsigned integer** —LabVIEW moves out-of-range values to the integer's minimum or maximum value. Most integer objects, such as the iteration terminal of a For Loop, round floating-point numbers. LabVIEW rounds a fractional part of 0.5 to the nearest even integer. For example, LabVIEW rounds 6.5 to 6 rather than 7.
- **LabVIEW treats enums as unsigned integers** —For example, if you convert -1, a floating-point number, to an enum, LabVIEW coerces the value into the range of the enum. If the range of the enum is 0 to 25, LabVIEW converts the value to 0.
- **Integer to integer** —LabVIEW does not move out-of-range values to the integer's minimum or maximum value. If the source is smaller than the destination, LabVIEW extends the sign of a signed source and places zeros in the extra bits of an unsigned source. If the source is larger than the destination, LabVIEW copies only the least significant bits of the value.
- **Integer, floating-point, or fixed-point number to fixed-point number** —LabVIEW coerces values outside the desired range for the fixed-point number to within the fixed-point number's maximum and minimum values.

Polymorphism for Numeric Functions

The arithmetic functions take numeric input data. With some exceptions noted in the function descriptions, the default output has the same numeric representation as the input or, if the inputs have different representations, the default output is the larger of the inputs. For example, if you add an 8-bit integer and a 16-bit integer, the default output is a 16-bit integer. If you configure the output of a Numeric function, the settings you specify override the default behavior.

The arithmetic functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on. A formal and recursive definition of the allowable input type is as follows:

Numeric type = numeric scalar OR array [numeric type] OR cluster [numeric types]

The numeric scalars can be floating-point numbers, integers, or complex floating-point numbers. LabVIEW does not allow you to use arrays of arrays.

Arrays can have any number of dimensions of any size. Clusters can have any number of elements. The output type of functions is of the same numeric representation as the input type. For functions with one input, the functions operate on each element of the array or cluster.

For functions with two inputs, you can use the following input combinations:

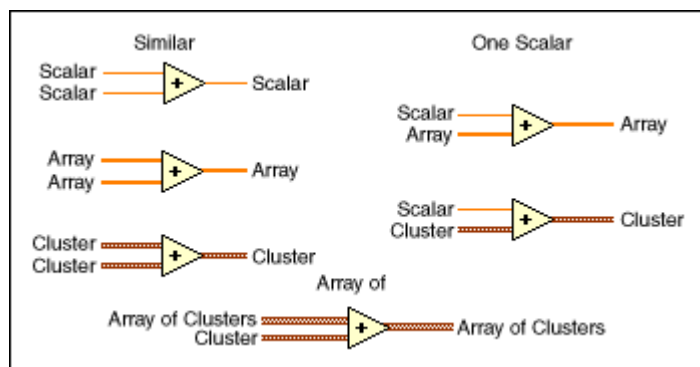
- **Similar** —Both inputs have the same structure, and the output has the same structure as the inputs.
- **One scalar** —One input is a numeric scalar, the other is an array or cluster, and the output is an array or cluster.
- **Array of** —One input is a numeric array, the other is the numeric type itself, and the output is an array.

For similar inputs, LabVIEW performs the function on the respective elements of the structures. For example, LabVIEW can add two arrays element by element. Both arrays must have the same dimensionality. You can add arrays with differing numbers of elements; the output of such an addition has the same number of elements as the smallest input. Clusters must have the same number of elements, and the respective elements must be of the same type.

For operations involving a scalar and an array or cluster, LabVIEW performs the function on the scalar and the respective elements of the structure. For example, LabVIEW can subtract a number from all elements of an array, regardless of the dimensionality of the array.

For operations that involve a numeric type and an array of that type, LabVIEW performs the function on each array element. For example, a graph is an array of points, and a point is a cluster of two numeric types, x and y. To offset a graph by 5 units in the x direction and 8 units in the y direction, you can add a point, (5, 8), to the graph.

The following illustration shows the possible polymorphic combinations of the Add function.



Polymorphism for Boolean Functions

The logical functions accept Boolean input data, numeric input data, and error clusters. If the input is numeric, LabVIEW performs a bitwise operation. If the input is an integer, the output has the same representation. If the input is a floating-point number, LabVIEW rounds it to a 32-bit integer, and the output is a 32-bit integer. If the input is an error cluster, LabVIEW passes only the TRUE or FALSE value of the **status** parameter of the error cluster to the input terminal.

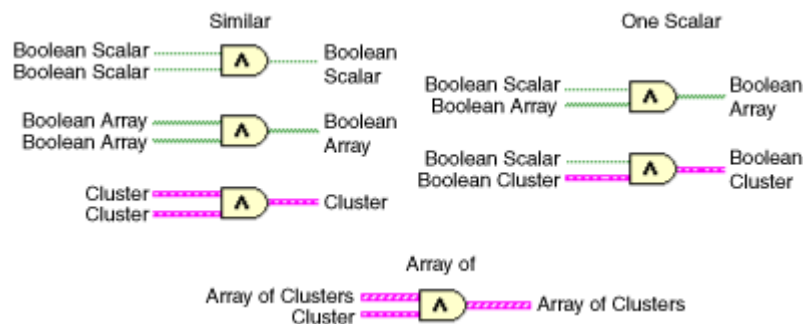
The logical functions work on arrays of numbers or Boolean values, clusters of numbers or Boolean values, arrays of clusters of numbers or Boolean values, and so on.

A formal and recursive definition of the allowable input type is as follows:

Logical type = Boolean scalar OR numeric scalar OR array [logical type] OR cluster [logical types]

except that complex numbers and arrays of arrays are not allowed.

Logical functions with two inputs can have the same input combinations as the arithmetic functions. However, the logical functions have the further restriction that the base operations can only be between two Boolean values or two numbers. For example, you cannot have an AND between a Boolean value and a number. The following illustration shows some combinations of Boolean values for the AND function.



Polymorphism for Array Functions

Most of the array functions accept n -dimensional arrays of any type. However, the wiring diagrams in the function descriptions show numeric arrays as the default data type.

Polymorphism for String Functions

String Length, To Upper Case, To Lower Case, Reverse String, and Rotate String accept strings, clusters and arrays of strings, and arrays of clusters. To Upper Case and To Lower Case also accept numbers, clusters of numbers, and arrays of numbers, interpreting them as ASCII codes for characters. Width and precision inputs must be scalar.

Polymorphism for String Conversion Functions

The Path To String and String To Path functions are polymorphic. That is, they work on scalar values, arrays of scalars, clusters of scalars, arrays of clusters of scalars, and so on. The output has the same composition as the input but with the new type.

Polymorphism for Additional String to Number Functions

Number To Decimal String, Number To Hex String, Number To Octal String, Number To Engineering String, Number To Fractional String, and Number To Exponential String accept clusters and arrays of numbers and produce clusters and arrays of strings. Decimal String To Number, Hexadecimal String To Number, Octal String To Number, and Fract/Exp String

To Number accept clusters and arrays of strings and produce clusters and arrays of numbers. Width and precision inputs must be scalar.

Polymorphism for Cluster Functions

The Bundle and Unbundle functions do not show the data type for their individual input or output terminals until you wire objects to these terminals. When you wire them, these terminals look similar to the data types of the corresponding front panel control or indicator terminals.

Polymorphism for Comparison Functions

The Comparison functions Equal?, Not Equal?, and Select take inputs of any type, as long as the inputs are the same type.

The functions Greater Or Equal?, Less Or Equal?, Less?, Greater?, Max & Min, and In Range and Coerce take inputs of any type except complex, path, or refnum, as long as the inputs are the same type. You can compare numbers, strings, Boolean values, arrays of strings, clusters of numbers, clusters of strings, and so on. However, you cannot compare a number to a string or a string to a Boolean value, and so on.

The functions that compare values to zero accept numeric scalars, clusters, and arrays of numbers. These functions output Boolean values in the same data structure as the input.

The Not A Number/Path/Refnum? function accepts the same input types as functions that compare values to zero. This function also accepts paths and refnums. Not A Number/Path/Refnum? outputs Boolean values in the same data structure as the input.

The functions Decimal Digit?, Hex Digit?, Octal Digit?, Printable?, and White Space? accept a scalar string or number input, clusters of strings or non-complex numbers, arrays of strings or non-complex numbers, and so on. The output consists of Boolean values in the same data structure as the input.

The function Empty String/Path? accepts a path, a scalar string, clusters of strings, arrays of strings, and so on. The output consists of Boolean values in the same data structure as the input.

You can use the Equal?, Not Equal?, Not A Number/Path/Refnum?, Empty String/Path?, and Select functions with paths and refnums, but no other Comparison functions accept paths or refnums as inputs.

Comparison functions that accept arrays and clusters normally return Boolean arrays and clusters of the same structure. If you want the function to return a single Boolean value, right-click the function and select **Comparison Mode»Compare Aggregates** from the shortcut menu to *compare aggregates*.

Polymorphism for Logarithmic Functions

The Logarithmic functions take numeric input data. If the input is an integer, the output is a double-precision, floating-point number. Otherwise, the output has the same numeric representation as the input.

These functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on. A formal and recursive definition of the allowable input type is as follows:

Numeric type = numeric scalar OR array [numeric type] OR cluster [numeric types]

except that arrays of arrays are not allowed.

Arrays can be any size and can have any number of dimensions. Clusters can have any number of elements. The output type is of the same numeric representation as the input, and the functions operate on each element of the cluster or array. Allowable input type combinations for the *two-input Logarithmic functions* include the following:

- **Similar** —Both inputs have the same structure, and the output has the same structure as the inputs.
- **One scalar** —One input is a numeric scalar, the other is a numeric array or cluster, and the output is an array or cluster.

Using Comparison Functions

Use the Comparison functions to compare *numeric values*, *strings*, *Boolean values*, *arrays*, and *clusters*. Most Comparison functions test one input or compare two inputs and return a Boolean value.

Comparing Numeric Values

Most of the Comparison functions test one input or compare two inputs and return a Boolean value. The functions convert numbers to the same representation before comparing them. The functions convert each input to the largest representation needed to make an accurate comparison. Comparisons with one or two inputs that have the value *NaN* return a result that indicates that the comparison was not equal.

Not all real numbers can be represented in the ANSI/IEEE standard *floating-point numbers*. Because of this, comparisons using floating-point numbers may yield results you do not expect because of rounding errors.

Comparing Strings

Comparison functions compare strings according to the values of the ASCII characters codes. Comparison functions compare strings one element at a time starting with the 0th element until they encounter an inequality or until one string ends. The functions treat the longer string as greater than the shorter string.

For example, α (with a decimal value of 97) is greater than A (65), which is greater than the numeral 0 (48), which is greater than the space character (32). LabVIEW compares characters one by one from the beginning of the string until an inequality occurs, at which time the comparison ends. For example, LabVIEW evaluates the strings abcd and abef until it finds c, which is less than the value of e. The presence of a character is greater than the absence of one. Thus, the string abcd is greater than abc because the first string is longer.

The functions that test the category of a string character, such as the Decimal Digit? and Printable? functions, evaluate only the first character of the string.

Comparing Boolean Values

The Comparison functions treat the Boolean value TRUE as greater than the Boolean value FALSE.

Comparing Arrays and Clusters

Some Comparison functions have two modes for comparing arrays or clusters of data. In Compare Aggregates mode, if you compare two arrays or clusters, the function returns a single Boolean value. In Compare Elements mode, the function compares the elements individually and returns an array or cluster of Boolean values.

In Compare Aggregates mode, the string comparison and array comparison operations follow exactly the same process—the Comparison function treats the string as an array of ASCII characters.



NOTE

The Max & Min function behaves differently than other comparison functions. The Max & Min function returns either the maximum element or array or the minimum element or array, depending on which mode you specify and which output you specify.

Right-click a Comparison function and select **Comparison Mode»Compare Elements** or **Comparison Mode»Compare Aggregates** from the shortcut menu to change the mode of the function. Some Comparison functions operate only in Compare Aggregates mode, so the shortcut menu items do not appear.

Arrays

When comparing multidimensional arrays, each array wired to the function must have the same number of dimensions. The Comparison functions that operate only in Compare Aggregates mode compare arrays the same way they compare strings—one element at a time starting with the first element until they encounter an inequality.

Compare Elements Mode

In Compare Elements mode, Comparison functions return an array of Boolean values of the same dimensions as the input arrays. Each dimension of the output array is the size of the smaller of the two input arrays in that dimension. Along each dimension such as a row, column, or page, the functions compare corresponding element values in each input array to produce the corresponding Boolean value in the output array.

Compare Aggregates Mode

In Compare Aggregates mode, comparison functions return a single Boolean result after comparing the elements in an array. Comparison functions compare elements sequentially until a decision can be made, much like the alphabetical ordering of English words, where letters of each word are compared until they are not equal. Therefore, the function performs the following steps to determine the result of the comparison:

- The function compares corresponding elements in each input array, starting at the beginning of the array.
- If the corresponding elements are not equal, the function stops—it returns the result of this comparison.
- If the corresponding elements are equal, the Comparison function processes the next pair of values, until it finds an inequality or reaches the end of one of the input arrays.
- If all values in the input arrays are equal but one array has extra elements at the end, the longer array is considered greater than the shorter one. For example, a Comparison function considers the array `[1, 2, 3, 2]` to be greater than the array `[1, 2, 3]`.



Note The In Range and Coerce function behaves differently than other Comparison functions in Compare Aggregates mode because it requires three inputs. After comparing the first set of elements, the function processes the next set of values only if all three inputs, **upper limit**, **lower limit**, and **x**, are equal. Otherwise, the function returns the result of comparing the first set of elements.

Clusters

Clusters you compare must include the same number of elements, each element in the clusters must be of compatible types, and the elements must be in the same cluster order. For example, you can compare a cluster of a DBL and a string to a cluster of an I32 and a string.

Compare Elements Mode

In Compare Elements mode, Comparison functions return a cluster of Boolean elements, one for each corresponding element in the input clusters.

Compare Aggregates Mode

In Compare Aggregates mode, comparison functions return a single Boolean result after comparing the elements in a cluster. Comparison functions compare elements sequentially until a decision can be made, much like the alphabetical ordering of English words, where letters of each word are compared until they are not equal. Therefore, the function performs the following steps to determine the result of the comparison:

- The function compares corresponding elements in each input cluster, starting at the beginning of the cluster.
- If the corresponding elements are not equal, the function stops—it returns the result of this comparison.
- If the corresponding elements are equal, the comparison function processes the next pair of values, until it finds an inequality or reaches the end of the input clusters.
- If all values in the input clusters are equal but one cluster has extra elements at the end, the larger cluster is considered greater than the smaller one. For example, a Comparison function considers the cluster [1, 2, 3, 2] to be greater than the cluster [1, 2, 3].



Note The In Range and Coerce function behaves differently than other Comparison functions in Compare Aggregates mode because it requires three inputs. After comparing the first set of elements, the function processes the next set of values only if all three inputs, **upper limit**, **lower limit**, and **x**, are equal. Otherwise, the function returns the result of comparing the first set of elements.

Use Compare Aggregates mode on clusters if you are comparing two records containing sorted data, where elements later in the cluster are considered secondary to elements earlier in the cluster. For example, if you compare a cluster containing two strings, last name followed by first name, the function would compare the first name fields only if the last name fields matched.

Comparing References

LabVIEW compares VI Server references using the Equal? and Not Equal? comparison functions. You can use these comparison functions to compare an array or cluster of a data type to a scalar of the same data type and produce an array or cluster of Boolean values. When comparing VI Server references, these functions return TRUE or FALSE based on whether the two references refer to the same object, regardless of whether the references have the same value or the same capability. Because you can have multiple references to the same object, the integer value of the references can be different but still refer to the same object. The capability of two references also can differ even though they both might refer to the same object. For example, if one of the references is authenticated while the other is not, the references would have different capabilities but could still refer to the same object.

For example, one VI Server reference might be a strict reference to a digital numeric control and the other might be a generic reference, but if they refer to the same object, they are equal. Similarly, when two references refer to the same object but one of the references is authenticated and the other is not, the function returns TRUE. However, if you have a local

reference and a remote reference to the same LabVIEW object, the Equal? function returns FALSE.

Comparing Variants

When comparing variant data, the data type, value, attribute name(s), and attribute value(s) must match to be considered equal.

VIs in LabVIEW

Express VIs

An Express VI is a VI whose settings you can configure interactively through a dialog box. Express VIs appear on the block diagram as *expandable nodes* with icons surrounded by a blue field.

You can configure an Express VI by setting options in the configuration dialog box that appears when you place the Express VI on the block diagram. You can double-click the Express VI or right-click the Express VI and select **Properties** from the shortcut menu to reopen the configuration dialog box. You also can configure the Express VI by wiring values to terminals of the Express VI on the block diagram.

Parameters that you can configure for an Express VI from the configuration dialog box are called *configurable parameters*. Parameters that you can configure for an Express VI from the block diagram are called *expandable parameters* or expandable terminals. A parameter can be both configurable and expandable, which means you can configure it from both locations. The configuration you specify for an Express VI with the configurable parameters determines the expandable terminals that appear on the block diagram.

You can use Express VIs to perform common measurement tasks. You also can *create* and edit Express VIs to distribute to users for building applications easily. Use the Create or Edit Express VI dialog box, available by selecting **Tools»Advanced»Create or Edit Express VI**, to create an Express VI *from an existing VI*, *from another Express VI*, or *from a blank VI*.

Benefits of Using Express VIs

The primary benefit of Express VIs is their interactive configurability. Express VIs are useful when you want to give users a VI or library of VIs for building their own applications easily with minimal programming expertise.

The configurability of Express VIs provides an interactive way to determine settings for operations that the user might not fully understand. For example, you might need a digital filter for a signal, but a library containing dozens of filter VIs does not help you choose the correct filter response or help you decide which parameters most directly affect the specific signal. The Filter Express VI allows you to interactively select the type of filter, manipulate filter parameters, and view the filter response in different ways. You can preview the results of different filters and filter parameters directly on the signal in the configuration dialog box of the Express VI. After you run the Filter Express VI, the Express VI displays the new signal as default data in the configuration dialog box.

Another advantage of Express VIs is that separate instances of an Express VI function independently. When you place an Express VI on the block diagram, an instance of the Express VI is embedded in that particular block diagram. The settings that you select in the configuration dialog box affect only that instance of the Express VI. If you place a standard VI in five different areas on the same block diagram, the result is five exact copies of the VI. The source code, default values, and front panel remain the same for all five copies. However, if you place an Express VI in five areas on a block diagram, the result is five separate Express VIs with **different names**, all independently configurable. Each instance of an Express VI can include different numbers and types of expandable terminals. The Context Help window also **updates** with the different settings for each instance.

Caveats When Using Express VIs

Express VIs do not provide run-time interactive configuration for VIs. If you need run-time reconfiguration, build an application with a user interface that contains features similar to a configuration dialog box.

Express VIs are designed for ease of use. If you need an application to run with strict memory restrictions or high execution speeds, use standard VIs.

Using the Dynamic Data Type

Most **Express VIs** accept and/or return the dynamic data type. The dynamic data type appears as a dark blue terminal, shown as follows.



The dynamic data type accepts data from and sends data to the following data types, where the scalar data type is a floating-point number or a Boolean value:

- 1D array of waveforms
- 1D array of scalars
- 1D array of scalars—most recent value
- 1D array of scalars—single channel
- 2D array of scalars—columns are channels
- 2D array of scalars—rows are channels
- Single scalar
- Single waveform

To view dynamic data, you can wire the data directly to an indicator, such as a graph, chart, or numeric indicator. However, because LabVIEW must convert the data to the data type of the indicator, displaying dynamic data can slow the execution speed of the VI.



NOTE

Most VIs and functions besides Express VIs do not automatically accept the dynamic data type. Therefore, to use a built-in VI or function to analyze or process dynamic data, you must manually **convert** the dynamic data to a type that the VI or function accepts.

The dynamic data type includes the data associated with a signal, as well as attributes that provide information about that signal, such as the name of the signal or the date and time the data was acquired. Attributes specify how the signal appears on a graph or chart. For example, if you use the DAQ Assistant Express VI to acquire a signal and plot that signal on a graph, the name of the signal appears in the plot legend of the graph, and the x-scale adjusts to display timing information associated with the signal in relative or absolute time based on the attributes of the signal. If you use the Spectral Measurements Express VI to perform an FFT analysis on the signal and plot the resulting value on a graph, the x-scale automatically adjusts to plot the signal in the frequency domain based on the attributes of the signal. Right-click a dynamic data type output terminal of a VI or function on the block diagram and select **Create»Graph Indicator** from the shortcut menu to display the data in a graph or select **Create»Numeric Indicator** from the shortcut menu to display the data in a numeric indicator.

The following table lists indicators that accept the dynamic data type and the type of data the dynamic data type can contain. The table also describes how indicators handle the data.

Data in Dynamic Data Type	Indicator	Result
Single numeric value	Graph	Plots the single value, includes timestamp and attributes
Single channel	Graph	Plots the whole waveform, includes timestamp and attributes
Multiple channels	Graph	Plots all the data, includes timestamps and attributes
Single numeric value	Numeric indicator	Displays the single value
Single channel	Numeric indicator	Displays the last value of the data from the channel
Multiple channels	Numeric indicator	Displays the last value of the data from the first channel
Single numeric value	Boolean indicator	Displays a TRUE value if numeric value is greater than or equal to 0.5
Single channel	Boolean indicator	Displays a TRUE value if the last value of the data from the channel is greater than or equal to 0.5

Data in Dynamic Data Type	Indicator	Result
Multiple channels	Boolean indicator	Displays a TRUE value if the last value of the data from the first channel is greater than or equal to 0.5

Retrieving and Setting Dynamic Data Attributes

Use the Get Dynamic Data Attributes Express VI to retrieve dynamic data attributes. When you add the Get Dynamic Data Attributes Express VI to the block diagram, a configuration dialog box appears. Use this dialog box to retrieve the attributes of a signal in the dynamic data you wire to this Express VI.

Use the Set Dynamic Data Attributes Express VI to set dynamic data attributes, such as the signal name, timestamp, time mode, and so on. When you add the Set Dynamic Data Attributes Express VI to the block diagram, a configuration dialog box appears. Use this dialog box to modify or set the attributes of a signal in the dynamic data you wire to this Express VI.



Note The Get Dynamic Data Attributes Express VI and Set Dynamic Data Attributes Express VI refer to both channels and signals as signals.

Converting Dynamic Data

Use the Convert to Dynamic Data Express VI to convert numeric, Boolean, waveform, and array data to dynamic data for use with [Express VIs](#). When you place the Convert to Dynamic Data Express VI on the block diagram, a configuration dialog box appears. Use this dialog box to select the kind of data to convert to the dynamic data type.

Use the Convert from Dynamic Data Express VI to convert dynamic data to numeric, waveform, and array data types for use with other VIs and functions. When you place the Convert from Dynamic Data Express VI on the block diagram, a configuration dialog box appears. The configuration dialog box displays options that let you specify how you want to format the data that the Convert from Dynamic Data Express VI returns.

When you wire dynamic data to an array indicator, LabVIEW places the Convert from Dynamic Data Express VI on the block diagram. Double-click the Convert from Dynamic Data Express VI to open the configuration dialog box and control how the data appears in the array.

Creating SubVIs from Express VIs

You can create a subVI from an [Express VI](#) configuration. For example, you might want to save the configuration of the Write To Measurement File Express VI for use as a subVI in other VIs you build rather than reconfiguring the Express VI every time. Right-click the Express VI and select **Open Front Panel** from the shortcut menu to create a subVI from an Express VI.

When you create a subVI from an Express VI, the front panel of the subVI appears. You then can edit the VI and save it. To save the values entered in controls, select **Edit»Make Current Values Default** or right-click each control and select **Data Operations»Make Current Value Default** from the shortcut menu. A new subVI displayed as an *expandable node* replaces the Express VI on the block diagram.

After you create a VI from an Express VI, you cannot convert the subVI back into an Express VI.

Creating Express VIs

An *Express VI* consists of the following components:

- *Configuration dialog box VI* —Allows users to configure settings for the run-time behavior of the Express VI.
- *Source VI* —Contains the code for the Express VI. The source VI also contains a link to the configuration dialog box VI.

When you place an Express VI on the block diagram, a configuration dialog box appears. The configuration dialog box VI determines the appearance and functionality of this configuration dialog box. After you configure the Express VI and close the configuration dialog box, the Express VI appears as an *expandable node* on the block diagram. You can set the inputs and outputs of the expandable node. However, you cannot double-click the expandable node to open it and see the underlying front panel and block diagram as you can do with a subVI. You must *convert the Express VI to a subVI* in order to view the underlying code. The underlying code of the Express VI is determined by the source VI. After you convert an Express VI to a subVI, you no longer can access the configuration dialog box or convert the subVI back to an Express VI.

You must design both the configuration dialog box VI and the source VI for any Express VI you create.

To create and distribute an Express VI, you must complete the following stages of development.

- *Design the source VI* for the Express VI.
- *Design the configuration dialog box*, including the user interface.
- *Associate help files* with the Express VI.
- Place the Express VI in the LabVIEW *directory structure*.
- Add the Express VI to a *new or existing palette*.

When you add an Express VI to the `labview\user.lib\_express` directory, the Express VI appears on the Express User Libraries palette. A **paletteMenu** subpalette appears for each LLB or directory in the `labview\user.lib\_express` directory that

does not start with an underscore. As a result, both the Express VI and the subVI of the source VI appear on the palette.



NOTE

If the **paletteMenu** subpalette does not appear on the Express User Libraries palette, change the palette view to refresh the palettes. You might need to restart LabVIEW to view new Express VIs on a palette.

You can [edit the palettes](#) to display only the Express VIs and to remove additional layers of subpalettes so the Express VIs are easier to access. You can edit custom palette views with Express VIs in the same way you edit palettes with other objects.



Note When you place an Express VI on a palette, the blue box around the icon might appear to be missing. However, after you save changes to the palette, the Express VI looks correct.

Creating Documentation for Express VIs

You can configure an [Express VI](#) such that a [help file launches](#) when you click the **Help** button in the configuration dialog box. You also can configure an Express VI to [display the current configuration](#) in the Context Help window.

Associating Help Files with Express VIs

Use the Control Online Help function to associate help files with Express VIs so that a help file launches when you click the **Help** button in the configuration dialog box. This function appears on the block diagram of each [configuration dialog box VI](#) you build.

The Control Online Help function opens a .chm or .html file located in a directory that you specify in the **Path to the help file** input. The function searches the file for text that you specify in the **String to search for** input and opens the file at that point in the text.

Displaying Context Help for Express VIs

Use the genHelp VI to [display the current configuration](#) of an Express VI in the Context Help window. The genHelp VI specifies the controls whose values you want to display in the **Context Help** window when you hover over the Express VI on the block diagram. You also can specify text, such as the control names, to precede the control values in the **Context Help** window. The genHelp VI appears as a subVI in the configuration dialog box VI for all Express VIs you create using the Express VI Creator wizard. The name of this subVI is genHelp ExpressVIName.

The ex_PPGetProp subVI of the genHelp VI appends control values to help strings. Use a separate copy of the ex_PPGetProp VI for each control that contains a value you want to display in the Context Help window. The ex_PPGetProp VI concatenates the help for each control, so you must wire the output from one copy of the VI to the input of the subsequent VI.

After you configure the genHelp VI for the Express VI, the Context Help window updates each time you close the configuration dialog box of the Express VI. If you do not modify the genHelp VI appropriately, an error message related to context help appears when you close the configuration dialog box. Additionally, the corresponding *validation test* for the Express VI fails.

Locating Express VI Components

An *Express VI* consists of a *configuration dialog box VI*, a *source VI*, and subVIs. LabVIEW saves source VIs and configuration dialog box VIs in different LLBs that are stored together in the file system. If the source VI and configuration dialog box VI share subVIs, you can save the subVIs to a third LLB or place them in the most logical location.

LabVIEW saves Express VI components using the following naming conventions:

Component	Source VI Equivalent	Configuration Dialog Box VI Equivalent
File Library	ExpressVIName Source.llb	_ ExpressVIName Config.llb
VI Filename	ExpressVIName Source.vi	Configure ExpressVIName.vi
SubVI Filename	sub ExpressVIName.vi	genHelp ExpressVIName.vi

You can find Express VIs and their components in the following directories:

- labview\vi.lib\express —Contains the Express VIs that LabVIEW provides. This directory contains subdirectories of Express VIs that correspond to the subpalettes on the Functions palette.
- labview\vi.lib\express\express shared —Contains the support VIs that Express VIs and configuration dialog box VIs call. Use the VIs in this directory to perform operations such as handling radio buttons and *generating context help*.
- labview\user.lib_express —Contains Express VIs you create and save in the default location. (macOS and Linux) The default location for Express VIs you create is the LabVIEW default data directory because not all users can write to the user.lib directory.

The source VI for an Express VI contains a link to its associated configuration dialog box VI. If you move the configuration dialog box VI to a different location relative to vi.lib or user.lib, the link breaks, and the Express VI prompts users to find the configuration dialog box VI.

When you save a VI that contains an Express VI, LabVIEW saves the Express VI inside the VI. When you open the VI, the Express VIs within it load from the same file as the VI and not from the source VI file of the Express VI. Therefore, you must place all the functionality of the Express VI in a subVI that you can update as necessary.

Deploying Express VIs to Other Computers

If you want to move Express VIs to other computers, move the two VI libraries for the Express VI and any related .mnu files. When you create Express VIs that you want to deploy to other computers, you must use the same directory structure that the target computers use. The components of an Express VI link to each other through both relative and specific paths. If the directory structure is different on each computer to which you deploy, the links break, and the Express VI prompts users to find the configuration dialog box VI.

Designing Configuration Dialog Box VIs

The configuration dialog box VI of an *Express VI* allows users to configure settings for the run-time behavior of the Express VI. The configuration dialog box VI also contains the user interface for the Express VI. After you *create* a new Express VI with the Express VI Creator wizard and *design the source VI*, you can *design the front panel* of the configuration dialog box VI and then *modify the provided block diagram template* to configure the run-time behavior of the Express VI.

Designing the Configuration Dialog Box VI Front Panel

Open the front panel of a configuration dialog box VI to complete the user interface. By default, LabVIEW defines only the basic **OK**, **Cancel**, and **Help** buttons. The **Main Selection** frame contains the user interface objects for the configuration dialog box. Place any user interface objects for the Express VI configuration dialog box in the **Main Selection** frame.

The following list describes some recommendations to consider when *building the user interface*.

- Use controls and indicators from the **System** palette as much as possible or copy and paste objects from other configuration dialog boxes. System controls use platform-specific drawing functions, so they are useful in making dialog boxes look and behave like standard dialog boxes for each platform.
- Use system colors and fonts.
- Minimize the number of hidden selections, such as hidden controls, tab controls, or multiple pages. If you cannot create a configuration dialog box without hidden objects, use a second page or a tab control.
- Keep the user interface style consistent. For example, the **OK**, **Cancel**, and **Help** buttons should appear in the lower right corner of a configuration dialog box.

Also use the *LabVIEW style checklist* and other development guidelines when building the user interface.

After creating the user interface, resize the front panel to show only the necessary items for the configuration of the Express VI. In general, resize the front panel to display only the **Main Selection** frame and the **OK**, **Cancel**, and **Help** buttons.

Designing the Configuration Dialog Box VI Block Diagram

After creating the user interface for the configuration dialog box, modify the provided block diagram template. The block diagram of the configuration dialog box VI is divided into three sections: Initialize, Main While Loop, and Reconfigure.

Initialize Section

The Initialize section of the block diagram includes the **Control Data** array and an empty Stacked Sequence structure. The **Control Data** array lists the [configurable parameters](#) for the source VI. The Stacked Sequence structure allows you to set default values for configurable controls.

The control names in the **Control Data** array match the control names in the source VI. The array must include all controls that transfer values between the configuration dialog box and the Express VI. The Express VI reads the control values, [generates context help information](#) for them, and writes them back to the Express VI.

Each element in the **Control Data** array is a cluster that contains the following objects:

- **Ctl Type** —Specifies the type of control in the source VI, such as a front panel control, a constant, the Express VI title, or a dynamic data type control.
- **SourceVI Ctl Label** —Specifies the item label as it appears on the front panel of the source VI.
- **ConfigDiag Ctl Label** —Specifies the item label as it appears on the front panel of the configuration dialog box VI. An empty string indicates the label is the same as the source VI label.



TIP

Right-click the **Control Data** array and select **Visible Items»Label** from the shortcut menu to view the labels for each array element.

Main While Loop Section

The Main While Loop section of the block diagram handles user interface events. This section corresponds to the area of the front panel that displays default values and values from the most recent iteration of the Express VI.

Because the configuration dialog box VI waits for user events, the main object in the While Loop is an Event structure. The Event structure waits for the user to click one of the three buttons that all dialog boxes include: **OK**, **Cancel**, and **Help**. Clicking either the **OK** or **Cancel** button ends the loop. Clicking the **OK** button causes the configuration dialog box VI to save the new configuration. Clicking the **Cancel** button closes the configuration dialog box without saving any changes. Clicking the **Help** button keeps the configuration dialog box open but also opens a [referenced help file](#).

The Main While Loop section of the configuration dialog box VI allows users to enter data that the VI writes in the Reconfigure section. When users select the settings they want from the Express VI configuration dialog box, the Main While Loop section processes the

selections. The Main While Loop section disables controls if necessary, updates radio buttons, and updates feedback indicators, such as graphs of sample or preview data.

Reconfigure Section

The Reconfigure section of the block diagram includes the **Terminals to Grow** array and the genHelp VI. The **Terminals to Grow** array lists the terminals of the Express VI that appear expanded by default. Edits to the array cannot cause expanded terminals to disappear or change order. As a result, terminals that users expand and wire override values that users select in the configuration dialog box. The genHelp VI updates the Context Help window with the current configuration of the Express VI.

You also can use the Reconfigure section to rename inputs and outputs and to wire controls and indicators to the terminal pane.

Designing Source VIs

The source VI of an *Express VI* contains the underlying code for the Express VI. The source VI serves as a wrapper around a subVI that determines how the Express VI behaves. The source VI also determines the connector pane and icon of the Express VI. After you *create* a new Express VI with the Express VI Creator wizard, build the source VI to specify how the Express VI behaves and to design the connector pane and icon.

Building the Source VI Front Panel

Build the front panel of the source VI to include all objects needed to pass information to the subVI. This front panel is not exposed to the user. If you created the Express VI *from an existing VI*, the front panel of the source VI contains all the controls and indicators from the original VI that you set as *expandable*. If you created the Express VI *from another Express VI*, the front panel of the source VI is identical to that of the original Express VI. If you created the Express VI *from a blank VI*, the front panel of the source VI is blank, and you must build it from scratch.

The following list describes some recommendations to consider when building the front panel of the source VI for an Express VI.

- Use only controls and indicators that exist on the Controls palette. If a user right-clicks an Express VI terminal and selects **Create»Control/Indicator** from the shortcut menu, LabVIEW creates a copy of the control/indicator on the front panel of the Express source VI.
- Use a 1D array of waveforms for multivalue inputs. For multivalue outputs, use a waveform graph. Otherwise, use a numeric control for scalar values, an LED control for Boolean values, and a string control for strings. Do not modify the controls and indicators.

Building the Source VI Block Diagram

After creating the front panel for the source VI, you can modify the block diagram to specify how the Express VI behaves. If you created the Express VI from an existing VI, the block diagram of the source VI contains a subVI identical to the original VI. If you created the Express VI from another Express VI, the block diagram of the source VI is identical to that of the original Express VI. If you created the Express VI from a blank VI, the block diagram of the source VI contains a template subVI. Build the subVI to perform all the intended functionality of the Express VI, and design the **connector pane** and a **unique icon** for the subVI. The connector pane must include all the inputs and outputs you want to configure programmatically through the Express VI.

The following list describes some recommendations to consider when building the subVI for an Express VI.

- If you are creating an Express VI that contains several memory-intensive subVIs that do not enable simultaneously, call them by reference. However, if you include the Express VI in an executable application created with the LabVIEW Application Builder, you must manually add the subVIs called by reference to the build.
- You can include objects that are not part of the core LabVIEW development environment, such as LabVIEW module or toolkit VIs, drivers, or other software. However, Express VIs that include these objects can run only on computers on which those objects are present.

The block diagram of the source VI must include a control or constant for each configurable or expandable parameter in the Express VI. If a parameter is both configurable and expandable, or if a parameter is expandable but not configurable, the block diagram of the source VI must contain a corresponding control or indicator for that parameter. If a parameter is configurable but not expandable, the block diagram of the source VI must contain a corresponding constant for that parameter.

Designing the Express VI Connector Pane and Icon

After building the front panel and block diagram of the source VI, design the connector pane and a unique icon for the source VI. You must include all expandable terminals on the connector pane of the source VI. You do not need to include configurable parameters on the connector pane. The connector pane and icon of the source VI correspond to the connector pane and icon of the Express VI.

Selecting Express VI Parameter Options

Parameters that you can configure for an **Express VI** from the configuration dialog box are called configurable parameters. Parameters that you can configure for an Express VI from the block diagram are called expandable parameters or expandable terminals. A parameter can be both configurable and expandable, which means you can configure it from both locations. The configuration you specify for an Express VI with the configurable parameters determines the expandable terminals that appear on the block diagram.

Parameters that you set as configurable or expandable must exist both in the **source VI** and the **configuration dialog box VI**. In the source VI, edit the block diagram to include a control

or constant for each configurable or expandable parameter. If a parameter is both configurable and expandable, or if a parameter is expandable but not configurable, the source VI must contain a corresponding control or indicator for that parameter. If a parameter is configurable but not expandable, the source VI must contain a corresponding constant for that parameter. You must include all expandable terminals on the connector pane of the source VI. You do not need to include configurable parameters on the connector pane.

On the [block diagram of the configuration dialog box VI](#), the **Control Data** array lists the configurable parameters for the Express VI, and the **Terminals to Grow** array lists the expandable terminals for the Express VI. Add, remove, or edit elements in these arrays to set the number and type of configurable and expandable parameters you want.

You also can set parameters as configurable or expandable when you use the Express VI Creator wizard to convert a standard VI to an Express VI. The **Configurable** column on the Convert VI page corresponds to the **Control Data** array on the block diagram of the configuration dialog box VI. The **Expandable** column on the **Convert VI** page corresponds to the **Terminals to Grow** array.

Before you create an Express VI, use the following table to determine which options work best for each parameter.

Parameter Options Selected	Result	Effect on Users
Configurable and Expandable (default)	Adds an element to the Terminals to Grow array. If the parameter is a control, an element is added to the Control Data array and a control to the source VI.	Users need either to configure the control in the configuration dialog box or to wire a value to the Express VI.
Configurable	Adds an element to the Control Data array and a constant to the source VI.	Users need to configure the control in the configuration dialog box.
Expandable	Adds an element to the Terminals to Grow array and a control/indicator to the source VI.	Users need to wire a value to the Express VI.
None	Does not modify either Express VI component.	Users can keep the default value.

When you run an Express VI, values you specify using expandable terminals take precedence over values you specify using configurable parameters.

Handling Variant Data

Sometimes you may need a VI to handle data of many different types in a generic way. You could write a different VI for each specific data type, but having copies of your VI is hard to maintain, since if you change one you have to remember to change all of them. LabVIEW provides the variant data type as one solution for these situations. The variant data type is a generic container for all other types of data in LabVIEW. When you convert other data to a

variant, the variant stores the data and the original data type of the data, which allows LabVIEW to correctly convert the variant data back to the original data at a later time. For example, if you convert string data to a variant, the variant stores the text of the string and an indicator that says that this data was originally a string (as opposed to a path or an array of bytes, or other possible LabVIEW types).

Use the Variant functions to create and manipulate variant data. You can convert any LabVIEW data type to the variant data type to use variant data in other VIs and functions. Several properties and methods, such as the Get VI:ConPane DataType method and the Connector Pane:DataType property, return the data type as a variant. Use the Data Type Parsing VIs to retrieve the data type of a variant and type information about the data type. You also can check whether the data type of a variant matches a specific data type.

Use the variant data type when it is important to manipulate data independently of data type, such as when you transmit or store data; read and/or write to unknown devices; or perform operations on a heterogeneous set of controls.

For example, use the variant data type when working with [ActiveX objects](#) that require a variant data type. You also can use the variant data type with control references to read or modify values of a [programmatically selected control](#).

You also can use the Flatten To String function to convert a data type to a string data type to represent data independently of type. [Flattening data](#) to strings is useful when you use TCP/IP to transmit data because the protocol understands only strings. However, using flattened data has limitations because LabVIEW cannot coerce flattened data when the original data type does not match the data type to which you want to convert. You cannot flatten ActiveX variant data. Also, attempting to unflatten a flattened integer as an extended-precision floating-point number fails.

Another advantage of using the variant data type is the ability to store attributes of the data. An attribute is information about the data that you define and the variant data type stores. For example, if you want to know the time when a piece of data was created, you can store the data as variant data and add an attribute called **Time** to store the time string. The attribute data can be of any type. Use variant attributes when you want to sort the data by a particular attribute, identify the device or application that generated the data, or filter the data for only those variants with a particular attribute.

**NOTE**

If you use the Flatten To String function on variant data, LabVIEW will flatten the variant and all its contents, including attributes. However, if you use the Variant To Flattened String function, LabVIEW flattens only the variant and discards any attributes.

Creating VIs and SubVIs

A VI can serve as a user interface or as an operation you use frequently. After you learn how to build a front panel and block diagram, you can create your own VIs and subVIs, distribute VIs, and build stand-alone applications and shared libraries.



RELATED INFORMATION

[Examples, VI Templates, Project Templates, and Sample Projects](#) on page 118

[Creating SubVIs from Sections of a VI](#) on page 201

[Using Icons](#) on page 216

[Saving VIs](#) on page 220

[Customizing VIs](#) on page 223

[Placing SubVIs on Block Diagrams](#) on page 1394

[Opening and Editing SubVIs](#) on page 1396

[Saving VIs](#) on page 929

[Customizing VIs](#) on page 942

Using Built-In VIs and Functions

LabVIEW includes built-in VIs and functions to help you build specific applications, such as data acquisition VIs and functions, VIs that access other VIs, VIs that communicate with other applications, and so on. You can use these VIs as subVIs in an application to reduce development time. Before you build a new VI, consider searching the Functions palette for similar VIs and functions and using an existing VI as the starting point for the new VI.



RELATED INFORMATION

[Creating Modular Code](#) on page 203

SubVIs in LabVIEW

Creating SubVIs from Sections of a VI

Convert a section of a VI into a subVI by using the Positioning tool to select the section of the block diagram you want to reuse and selecting **Edit»Create SubVI**. An icon for the new subVI replaces the selected section of the block diagram. LabVIEW creates controls and indicators for the new subVI, automatically configures the connector pane based on the number of control and indicator terminals you selected, and wires the subVI to the existing wires.

Creating a subVI from a selection is convenient but still requires careful planning to create a logical hierarchy of VIs. Consider which objects to include in the selection and avoid changing the functionality of the resulting VI.

Displaying SubVIs and Express VIs as Icons or Expandable Nodes

You can display VIs and Express VIs as icons or as expandable nodes. Expandable nodes appear as icons surrounded by a colored field. When displayed as expandable nodes, subVIs appear as icons surrounded by a yellow field and Express VIs appear as icons surrounded by a blue field. Use icons if you want to conserve space on the block diagram. Use expandable nodes to make wiring easier and to aid in documenting block diagrams. By default, subVIs appear as icons on the block diagram, and Express VIs appear as expandable nodes. To display a subVI or Express VI as an expandable node, right-click the subVI or Express VI and remove the checkmark next to the **View As Icon** shortcut menu item.



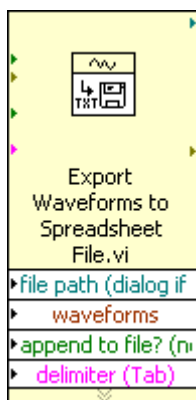
NOTE

If you display a subVI or Express VI as an expandable node, you cannot enable database access for that node.

When you resize an expandable subVI or Express VI, the input and output terminals of the subVI or Express VI appear below the icon. Optional terminals appear with gray backgrounds. Recommended or required input or output terminals you do not display appear as input or output arrows in the colored field that surrounds the subVI or Express VI icon. If you wire to an optional terminal when the subVI or Express VI is expanded, then resize the subVI or Express VI so the optional terminal no longer appears in the expanded field, the optional terminal appears as an input or output arrow in the colored field. However, if you unwire the optional terminal, the input or output arrow does not appear.

By default, inputs appear above outputs when you expand the subVI or Express VI. Right-click a terminal in the expandable field and select **Select Input/Output** from the shortcut menu to select an input or output to display. A line divides inputs from outputs in the shortcut menu. Labels for expandable subVIs and Express VIs appear in the colored field that surrounds the icon. To resize the expandable node so it accommodates the name of each terminal on a single line in the expandable field, right-click the subVI or Express VI and select **Size to Text** from the shortcut menu.

The following illustration shows an expandable subVI that displays four of 10 input and output terminals.



Creating Modular Code

You use subVIs in LabVIEW to create code that you can use in other VIs. Large block diagrams are hard to read and maintain. Identifying commonly reusable segments of code and replacing these segments with a subVI saves space and simplifies future updates to the code because when you edit the subVI, the changes affect all calls to the subVI, not just the current instance. A block diagram that contains several identical subVI nodes calls the same subVI each time.



NOTE

A subVI corresponds to a subroutine call in text-based programming languages.

After you build a VI, you must build a connector pane and create an icon to use it as a subVI. You then can place the subVI on the block diagram of another VI. The subVI controls and indicators receive data from and return data to the block diagram of the calling VI.



RELATED INFORMATION

[Creating SubVIs from Selections](#) on page 1396

[Opening and Editing SubVIs](#) on page 1396

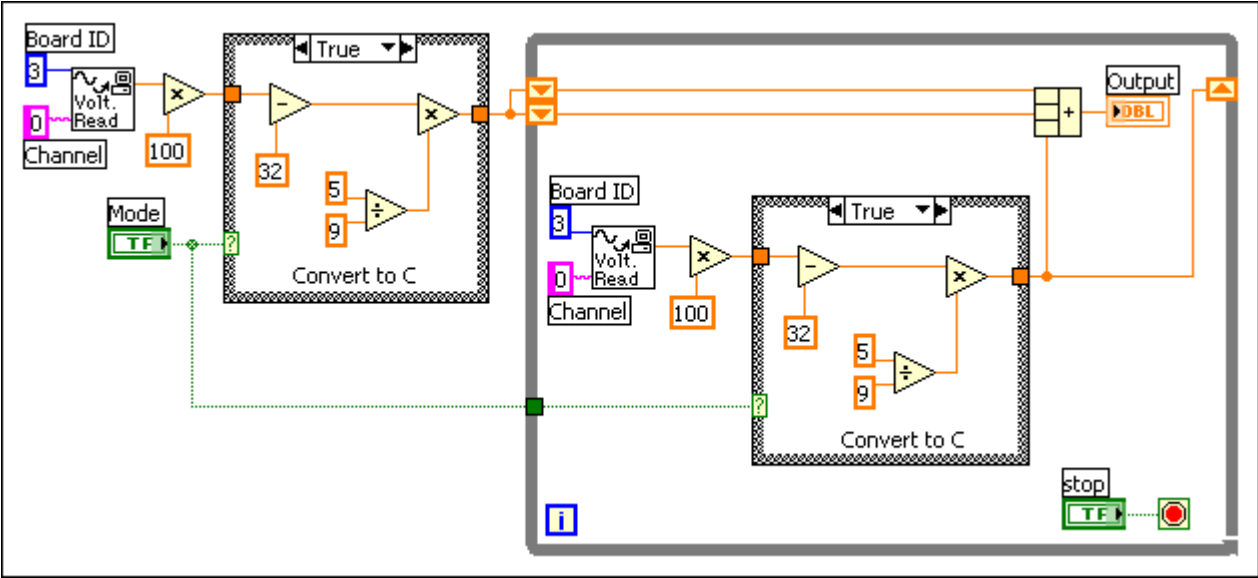
[Building the Connector Pane](#) on page 204

[Using Icons](#) on page 216

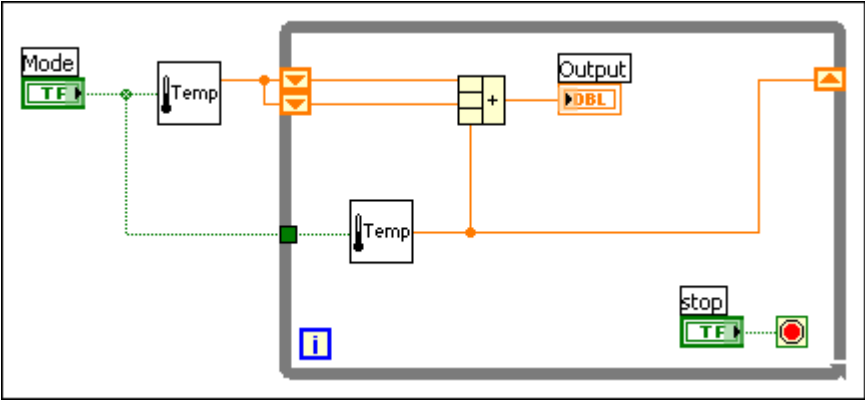
[Placing SubVIs on Block Diagrams](#) on page 1394

Watching for Common Operations

As you create VIs, you might find that you perform a certain operation frequently. Consider using subVIs or loops to perform that operation repetitively. For example, the following block diagram contains two identical operations.



You can create a subVI that performs that operation and call the subVI twice, as shown in the following block diagram.



You also can reuse the subVI in other VIs. You can *use a loop* to combine common operations.

Building the Connector Pane

To use a VI as a subVI, you need to build a connector pane. Every VI displays a connector pane, shown as follows, next to the VI icon in the upper right corner of the front panel window.



The connector pane is a set of terminals that corresponds to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages. The connector pane defines the inputs and outputs you can wire to the VI so you can use it as a subVI. A connector pane receives data at its input terminals, passes the

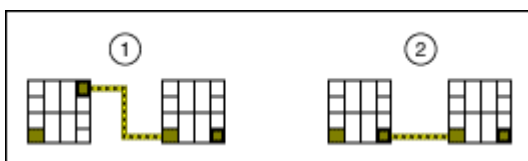
data to the block diagram code through the front panel controls, and receives the results at its output terminals from the front panel indicators.

Each rectangle on the connector pane represents a terminal. After you select a connector pane pattern, use the rectangles to assign front panel controls and indicators as inputs and outputs.

Recommendations for Building the Connector Pane

The following list describes some recommendations to consider when you build a connector pane. Refer to the *LabVIEW Style Guide* for more tips on creating connector panes.

- National Instruments recommends you avoid using more than 16 terminals. A connector pane with more than 16 terminals can be difficult to wire. If you need to pass more data, use [clusters](#).
- If you anticipate changes to the VI that would require a new input or output, leave extra terminals unassigned.
- When assigning terminals as inputs and outputs, make sure to split the terminals of the connector pane consistently. If you need to use the middle four terminals of the 4 x 2 x 2 x 4 pattern, divide them either horizontally or vertically. For example, assign the inputs to the top two terminals and the outputs to the bottom two terminals or assign the inputs to the left two terminals and the outputs to the right two terminals.
- If you create a group of subVIs that you use together often, give the subVIs a consistent connector pane with common inputs in the same location to help you remember where to locate each input. If you create a subVI that produces an output that another subVI uses as the input, such as references, task IDs, and error clusters, align the input and output connections to simplify the wiring patterns.
- Place the **error in** clusters on the lower left corner of the front panel and the **error out** clusters on the lower right corner. The following illustration shows examples of improperly and properly aligned error clusters.



1	Improper
2	Proper

Setting Required, Recommended, and Optional Inputs and Outputs

You can designate which inputs and outputs are required, recommended, and optional to prevent users from forgetting to wire subVI terminals.

For terminal inputs, required means that the block diagram on which you placed the subVI will be broken if you do not wire the required inputs. Required is not available for terminal outputs. For terminal inputs and outputs, recommended and optional mean that wiring the terminals is optional. If you do not wire the terminals, the block diagram on which you placed the subVI does not generate any warnings.

Inputs and outputs of VIs in `vi.lib` are already marked as **Required**, **Recommended**, or **Optional**. LabVIEW sets inputs and outputs of VIs you create to **Recommended** by default. Set a terminal setting to required only if the VI must have the input or output to run properly.

To designate which inputs and outputs are required, recommended, and optional, right-click a terminal on the connector pane and select **This Connection Is** from the shortcut menu. You can configure LabVIEW to set all new terminals you configure on the connector pane to required instead of recommended. Select **Tools»Options»Front Panel** and place a checkmark in the **Connector pane terminals default to required** checkbox. This option also applies to connections made using the **Edit»Create SubVI** menu item.

In the Context Help window, the labels of required terminals appear bold, recommended terminals appear as plain text, and optional terminals appear dimmed. The labels of optional terminals do not appear if you click the **Hide Optional Terminals and Full Path** button, shown as follows, in the **Context Help** window.



RELATED INFORMATION

[LabVIEW Wiki Style Guide](#)

[Selecting a Connector Pane Pattern](#) on page 1401

[Assigning Terminals to Controls and Indicators](#) on page 1398

Designing SubVI Front Panels

If users do not need to view the front panel of a subVI, you can spend less time on its appearance, including colors and fonts. However, front panel organization is still important because you might need to view the front panel while you debug the VI.

Add the controls and indicators to the front panel as they appear on the [connector pane](#). Add the controls on the left of the front panel and the indicators on the right. Place the **error in** clusters on the lower left of the front panel and the **error out** clusters on the lower right.



TIP

Include **error in** and **error out** clusters in all subVIs, even if the subVI does not process errors. **error in** and **error out** clusters are helpful for controlling execution flow.

Give controls [meaningful labels](#), including default values and measurement units.

Displaying the SubVI Front Panel

When LabVIEW calls a subVI, the subVI runs without displaying its front panel. To display the subVI front panel when LabVIEW calls the subVI, use the following dialog boxes:

- SubVI Node Setup dialog box—Sets options for only the current instance of the subVI.
- Customize Window Appearance dialog box—Sets options for every instance of the subVI.

Viewing the Hierarchy of VIs

The VI Hierarchy window displays a graphical representation of all open LabVIEW projects and targets, as well as the calling hierarchy for all VIs in memory. The window displays the following items:

- Projects and targets
- Static and dynamic subVI calls
- Type definitions
- Global variables
- Shared variables
- LabVIEW classes
- XControls
- Statecharts
- Project libraries .lvlib
- Express VIs
- Static VI references

Select **View»VI Hierarchy** to display the **VI Hierarchy** window. Use this window to [view the subVIs and other nodes](#) that make up the VIs in memory and to [search the VI hierarchy](#).

The **VI Hierarchy** window displays a top-level icon to represent the main LabVIEW [application instance](#), under which appear all the objects that are not part of a project or are not part of the application instance for a project. If you add a project, the **VI Hierarchy** window displays another top-level icon to represent the project. Each target you add appears under the project.

As you move the cursor over objects in the **VI Hierarchy** window, LabVIEW displays the name of each object in a tip strip. You can use the Positioning tool to drag an object from the **VI Hierarchy** window to the front panel or block diagram of another VI to use the object in that VI. You also can select and copy a node or several nodes to the clipboard and paste them to other front panels or block diagrams. Double-click a VI in the **VI Hierarchy** window to display the front panel of that VI.

The **VI Hierarchy** window is a dynamic view of what is in memory. If you rearrange items in the hierarchy, LabVIEW does not save the new position of items. When you close and then reopen the **VI Hierarchy** window, the positions regenerate.

A VI that contains subVIs has an arrow button on its bottom border. Click this arrow button to show or hide subVIs. A red arrow button appears when at least one subVI is hidden. A black arrow button appears when all subVIs are displayed. If a VI contains *recursive calls*, the **VI Hierarchy** window shows this relationship by drawing a dotted line between the recursive VIs.

Recursive VIs

Recursive VIs can call themselves from their own block diagram or from the block diagrams of subVIs. Recursive VIs are useful if you want to operate multiple times on the output of the same process. You can *make any VI recursive*, and you can use multiple recursive VIs in one VI hierarchy.

To use recursive VIs throughout a VI hierarchy, configure all VIs in the hierarchy in one of the following ways.

- Configure all VIs in the VI hierarchy to be *reentrant* and at least one VI to share clones of itself between instances of calls.
- Configure all VIs in the VI hierarchy to be *dynamic dispatch* member VIs.
- Configure at least one VI in the VI hierarchy to be a dynamic dispatch member VI or to be reentrant and share clones of itself between instances of calls. Configure the remaining VIs to be reentrant but with pre-allocated clones for each instance of a call.

(32-bit) LabVIEW allows up to 15,000 recursive calls on 32-bit platforms. (64-bit) LabVIEW allows up to 35,000 recursive calls on 64-bit platforms.

(Real-Time Module) The LabVIEW Real-Time Module supports recursion. However, recursion is not real-time deterministic and you cannot debug recursive VIs in LabVIEW Real-Time.

(FPGA Module) The LabVIEW FPGA Module currently does not support recursion.

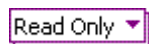
Polymorphic VIs

Polymorphic VIs adapt to different data types. A polymorphic VI is a collection of VIs with the same connector pane patterns. Each VI in the collection is an instance of the polymorphic VI, and each instance has at least one input or output terminal that accepts a different data type than in the other instances.

For example, the Read Key VI is polymorphic. Its **default value** terminal accepts Boolean; double-precision, floating-point numeric; 32-bit signed integer numeric; path; string; or 32-bit unsigned integer numeric data.

For most polymorphic VIs, the data types you wire to the inputs of the polymorphic VI determine the instance to use. If the polymorphic VI does not contain an instance compatible with that data type, a broken wire appears. If the data types you wire to the polymorphic VI inputs do not determine the instance to use, you must select the instance manually. If you manually select an instance of a polymorphic VI, the VI no longer behaves as a polymorphic VI because it accepts and returns only the data types of the instance you select.

To select the instance manually, right-click the polymorphic VI, select **Select Type** from the shortcut menu, and select the instance to use. You also can use the Operating tool to click the polymorphic VI selector, shown as follows, and select an instance from the shortcut menu.



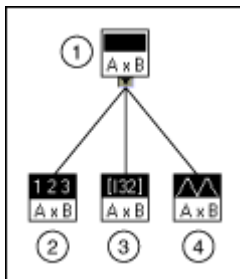
Right-click the polymorphic VI on the block diagram and select **Visible Items»Polymorphic VI Selector** from the shortcut menu to display the selector. To change the polymorphic VI to accept all the handled data types again, right-click the polymorphic VI and select **Select Type»Automatic** from the shortcut menu or use the Operating tool to click the polymorphic VI selector and select **Automatic** from the shortcut menu.

Build polymorphic VIs when you perform the same operation on different data types and the operation must be implemented differently across data types. If the operation can be implemented in similar ways for different data types without significant customization for a specific data type, *build malleable VIs* instead.

For example, if you want to perform the same mathematical operation on a single-precision floating-point numeric, an array of numeric values, or a waveform, you could create three separate VIs—Compute Number, Compute Array, and Compute Waveform. When you need to perform the operation, you place one of these VIs on the block diagram, depending on the data type you use as an input.

Instead of manually placing a version of the VI on the block diagram, you can create and use a single polymorphic VI.

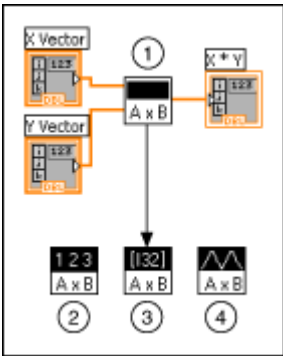
The polymorphic Compute VI contains three instances of the VI, as shown in the following illustration.



1	Compute
2	Compute Number
3	Compute Array

4	Compute Waveform

The Compute VI statically links the correct instance of the VI based on the data type you wire to the Compute subVI on the block diagram, as shown in the following illustration.



1	Compute
2	Compute Number
3	Compute Array
4	Compute Waveform

Polymorphic VIs differ from most VIs in that they do not have a block diagram or a front panel.

Consider the following issues when you build polymorphic VIs:

- All VIs you include in the polymorphic VI must have the same connector pane pattern because the connector pane of the polymorphic VI matches the connector panes of the VIs you use to create the polymorphic VI.
- The connector panes of the VIs must have corresponding input and output terminals. For example, if a terminal on the connector pane of one VI is an input, the corresponding terminal on the connector panes of the other VIs also must be an input or it must be unused.
- Each of the connector panes of the VIs can have different numbers of terminal connections.
- The VIs you include in the polymorphic VI do not have to consist of the same subVIs and functions.
- You cannot use polymorphic VIs in other polymorphic VIs.
- You cannot include a *malleable VI* .vim in a polymorphic VI.

If you want the data type a user wires to an input of the polymorphic VI to determine the instance to use and you do not want the user to know that the VI is polymorphic, consider the following guidelines when you build a polymorphic VI:

- Hide the polymorphic VI selector because the user does not need to select the instance manually.
- Create an icon for the polymorphic VI so that the icon for each instance does not appear in the Context Help window when a user moves the cursor over the polymorphic VI.
- Create a description of the polymorphic VI so that the description for each instance does not appear in the Context Help window when a user moves the cursor over the polymorphic VI. Include a description that describes the polymorphic VI regardless of the instance that is used.

When you place a polymorphic VI on the block diagram, LabVIEW loads the polymorphic VI into memory. If you make changes to the polymorphic VI in the **Polymorphic VI** window and close the window without saving the changes, the changes still exist in memory. If you close the VI and then open it again, the changes to the polymorphic VI no longer appear.

When you generate complete documentation for a VI that includes a polymorphic subVI, the polymorphic VI and its instances appear in the list of subVIs section of the documentation.

Related Information

[Building Polymorphic VIs](#)

[Choosing between Malleable VIs and Polymorphic VIs](#)

Malleable VIs

A malleable VI .vim is a VI that is inlined into its calling VI and can adapt each terminal to its corresponding input data type. With malleable VIs, you build a VI to perform the same operation on any acceptable data type instead of saving a separate copy of the VI for each data type. The malleable VIs that LabVIEW provides have orange backgrounds.

Refer to the `labview\examples\Malleable VIs\Basics\Malleable VIs Basics.lvproj` for an example of using malleable VIs.



RELATED INFORMATION

[VI Execution Speed](#) on page 265

[Using a Malleable VI for Different Data Types](#) on page 212

[Using a Malleable VI for Different Classes](#) on page 212

[Choosing between Malleable VIs and Polymorphic VIs](#) on page 213

[Building Malleable VIs](#) on page 928

[Debugging Malleable VIs](#) on page 929

Using a Malleable VI for Different Data Types

Malleable VIs adapt to input data types. All terminals of malleable VIs are polymorphic. Each time you wire an input data type to a terminal, type propagation runs to check if the wired type is acceptable. If type propagation errors are not detected, the malleable VI adapts the terminal to the input data type. For example, the Sort 2D Array VI is a malleable VI. The **2D array** input accepts an array of any data type except refnums. The **index** input accepts a signed 32-bit integer or an array of signed 32-bit integers.

Refer to the `labview\examples\Malleable VIs\Basics\Malleable VIs Basics.lvproj` for an example of using a malleable VI for different data types.



RELATED INFORMATION

[Malleable VIs](#) on page 211

[Polymorphic Functions](#) on page 179

[Type Propagation](#) on page 677

[Using a Malleable VI for Different Classes](#) on page 212

[Choosing between Malleable VIs and Polymorphic VIs](#) on page 213

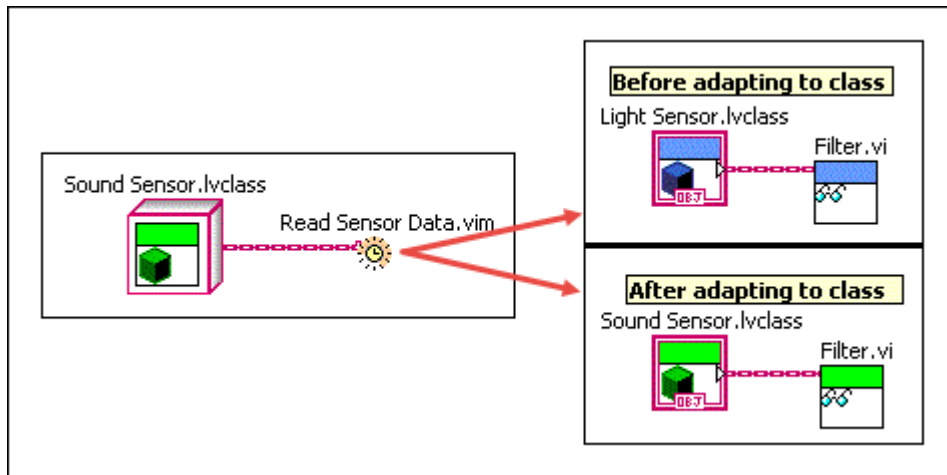
Using a Malleable VI for Different Classes

[Malleable VIs](#) adapt to input classes. A malleable VI that calls a method of class A accepts data of unrelated classes as long as the unrelated class has a public method with the same name, connector pane pattern, and terminal direction as the method of class A. The malleable VI adapts to the input class by replacing the call to the method of class A with a call to the public method in the input class, if such a method exists.

Compared with [dynamic dispatch VIs](#), malleable VIs are more flexible when adapting to input classes. A dynamic dispatch VI accepts only data of its own class or data of descendant classes. A malleable VI also accepts data of unrelated classes when possible.

The adaptability of a malleable VI defines an interface of method calls with method name, connector pane pattern, and terminal direction. The malleable VI accepts data of any class that fulfills the interface of method calls, which allows you to use one malleable VI to perform operations on data of unrelated classes or implement multiple interfaces defined by different malleable VIs on one class.

In the following figure, `Read Sensor Data.vi` calls the `Filter.vi` method of `Light Sensor.lvclass`, which has no common ancestor with `Sound Sensor.lvclass`. `Read Sensor Data.vi` adapts to `Sound Sensor.lvclass`, assuming that `Sound Sensor.lvclass` has a public method with the same name, connector pane pattern, and terminal direction as `Filter.vi`.



Refer to the `labview\examples\Malleable VIs\Class Adaptation\Malleable VIs - Class Adaptation.lvproj` for an example of using malleable VIs to define interfaces of method calls.

Related Information

[Using a Malleable VI for Different Data Types](#)

[Choosing between Malleable VIs and Polymorphic VIs](#)

Choosing between Malleable VIs and Polymorphic VIs

You can use a [malleable VI](#) or a [polymorphic VI](#) for different input data types. The following table compares the two types of VIs.

Aspect	Malleable VIs	Polymorphic VIs
Implementation	A single VI that computes whether a data type is acceptable by implementation.	A collection of instance VIs with a predefined set of acceptable data types.
Connector Pane	The same connector pane for all data types.	Different data types can have different connector panes.
Building	Create a VI with the <code>.vim</code> file extension or convert a standard VI into a malleable VI by changing the file extension to <code>.vim</code> .	Create an instance VI for each data type and include the instance VIs in a polymorphic VI.
Customizing	Customize sections of code in the malleable VI for specific data types using the Type Specialization structure.	Define different behaviors for specific data types by implementing different code in specific instance VIs.

Aspect	Malleable VIs	Polymorphic VIs
Adapt to classes?	Yes—A malleable VI that calls a method of class A accepts any class that has a public method with the same name, connector pane pattern, and terminal direction as the method of class A.	No—A polymorphic VI that calls a method of class A accepts only class A or descendants of class A.

Use the following table to help you decide which type of VI to choose.

Type	When to Use
Malleable VIs	- The operation can be implemented in similar ways for different data types without significant customization for a specific data type. - You want the VI to adapt to input classes.
Polymorphic VIs	- The operation must be implemented differently across data types. - The connector panes differ across data types.

VI Icons

Creating Icons

The primary way to create *icons* is to use the Icon Editor dialog box. You also can *drag and drop a graphic from the file system* or use the VI Icon methods in the VI class to create icons.

Creating Icons with the Icon Editor Dialog Box

You can use the Icon Editor dialog box to *create or edit icons*. The **Icon Editor** dialog box provides *icon templates*, *glyphs* from the Icon Library at ni.com, options for adding and formatting icon text, and support for editing with *layers*.

If you want to edit the icon for a VI or custom control, double-click the icon in the upper right corner of the front panel window, block diagram window, or *Control Editor* window to display the Icon Editor dialog box. You also can right-click the icon and select **Edit Icon** from the shortcut menu to display the **Icon Editor** dialog box. If you want to edit the icon for a *project library*, statechart, *class*, or *XControl*, click the **Edit** button on the **General**

Settings page of the corresponding Properties dialog box to display the **Icon Editor** dialog box.



NOTE

The Icon Editor dialog box displays icons in 24-bit color. However, LabVIEW displays icons only in 8-bit color on the block diagram. Therefore, the icon you create in this dialog box might appear slightly different from the corresponding icon on the block diagram.

Creating Icons by Dragging and Dropping Graphics

You also can *drag a graphic from anywhere in the file system* and drop it in the upper right corner of the front panel window to use the graphic as a VI icon. You can drag and drop .png, .bmp, or .jpg files.

LabVIEW converts the graphic you drag and drop to a 32 × 32 pixels icon. Graphics that are much larger than 32 × 32 pixels, particularly those that have fine gradations of color, might not convert well.



Note VIs with smaller icons can be awkward to select and wire and might look strange when wired. Refer to the *LabVIEW Style Checklist* for more tips on creating icons.

Guidelines for Designing Icons

You can use both pictures and text in an icon. A good size and font choice for text icons is 9-point Small Fonts in all caps.

If you localize an application, make sure you also localize the text on the icon. Do not use colloquialisms when making an icon because colloquialisms are difficult to translate. Also avoid using images in place of words if the images do not correspond to words of similar meaning in other languages.

You can use banners to identify related VIs. National Instruments recommends *creating and saving a banner as a template*. You then can use this template for a related VI icon and *modify the body of the VI icon* to provide information about the specific VI.

LabVIEW applies a library icon to the icon of all objects in the library. National Instruments recommends *creating only a banner for the library icon*. The icon for each object in the library then includes the banner, and you can *modify the body of the icon* to provide information about the specific object.

For example, you can create the following banner as a project library icon.



After you create the project library icon, you can create an icon for a VI within the project library. In the Icon Editor dialog box, select **Edit>Import Icon from Owning Library** to

import the icon of the owning project library. This displays the project library icon as a user layer of the VI icon called **NI_Library**. If a user layer with that name already exists, the new user layer overwrites the existing user layer.

Use icon text, glyphs, and the editing tools on the right side of the Icon Editor dialog box to modify the VI icon. For example, you can create a VI icon similar to the following icon.



Notice that the green **MY LIB** banner helps identify the VI as part of the project library.

Using Icons

Icons are graphical representations of VIs or project libraries.

Every VI displays an icon, shown as follows, in the upper right corner of the front panel and block diagram windows.



The default VI icon contains a number that indicates how many new VIs, up to nine VIs, you have opened since launching LabVIEW. To disable this numbering, select **Tools»Options»Front Panel** and remove the checkmark from the **Use numbers in icons of new VIs (1 through 9)** checkbox.

An icon can contain text or images. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI. If you add the VI to a palette, the VI icon also appears on the Functions palette.

Use the Icon Editor dialog box to edit a VI icon. Double-click the icon in the upper right corner of the front panel or block diagram window to display the **Icon Editor** dialog box.

You can use banners to identify related VIs. National Instruments recommends creating and saving a banner as a template. You then can use this template for a related VI icon and modify the body of the VI icon to provide information about the specific VI.

Icons for custom controls are similar to those of VIs. Every custom control displays an icon, shown as follows, in the upper right corner of the Control Editor window.



Double-click the icon in the upper right corner of the **Control Editor** window to display the Icon Editor dialog box, which you can use to edit the icon.

Icons for project libraries are different from those of VIs or custom controls. LabVIEW applies a library icon to the icon of all objects in the library. National Instruments

recommends creating only a banner for the library icon. For example, you can create the following banner as a project library icon.



The icon for each object in the library then includes the banner, and you can modify the body of the icon to provide information about the specific object.

Use the **General Settings** page of the Project Library Properties dialog box to view or edit the icon for the corresponding project library.

Icons for statecharts, classes, and XControls are similar to those of project libraries. Use the **General Settings** page of the Project Library Properties dialog box, Class Properties dialog box, or XControl Properties dialog box to view or edit the icon for the corresponding statechart, class, or XControl, respectively.



RELATED INFORMATION

[Using Libraries in LabVIEW Projects](#) on page 370

[Adding VIs or Objects from the Functions or Controls Palette](#) on page 835

[Creating a VI Icon](#) on page 1404

[Creating a VI Icon](#) on page 1404

[Creating a VI Icon](#) on page 1404

[Creating Custom Controls, Indicators, and Type Definitions](#) on page 165

[Creating Custom Controls, Indicators, and Type Definitions](#) on page 165

[Using Libraries in LabVIEW Projects](#) on page 370

[Creating a Library Icon](#) on page 1402

[Creating a Library Icon](#) on page 1402

[Creating a Library Icon](#) on page 1402

[Creating LabVIEW Classes](#) on page 565

[Using XControls](#) on page 174

Saving Icons

Save icons you create in the **Icon Editor** as templates or glyphs for later use.

About this task

You can save an *icon*, including *layer information*, with the corresponding VI or library. The Icon Editor can save icons in both 256-color (8-bit) and monochrome (1-bit) format.

If you open a third-party VI, you can choose whether to save the corresponding icon template as a template for future use. Use the **Templates** page of the Icon Editor Properties dialog box to configure settings for saving third-party icon templates.

Procedure

- 1. Complete the step that aligns with your goal.

Goal	Step
Save my icon as a template.	Select File»Save As»Template .
Save my icon as a glyph.	Select File»Save As»Glyph .

LabVIEW opens the Save Icon As dialog box.

- 2. Specify the name of your icon.
- 3. Specify the folder within your LabVIEW Data folder where you want to save the icon. To locate your LabVIEW Data folder, navigate to **Tools»Options»Paths** and select **Default Data Directory** from the drop-down menu.

Option	Description
LabVIEW Data\Icon Templates	The directory for icon templates.
LabVIEW Data\Glyphs	The directory for glyphs.

- 4. Click **OK**.

Results

By default, the **Icon Editor** saves your icon as 256-color .png file.

Using Glyphs

You can use the Icon Editor dialog box to create glyphs or add glyphs to an *icon*. The **Glyphs** page of the **Icon Editor** dialog box displays all .png, .bmp, and .jpg files in the LabVIEW Data\Glyphs directory. You can save glyphs you create to this directory so that they appear in the **Glyphs** page of the **Icon Editor** dialog box. By default, the **Glyphs** page includes all glyphs from the Icon Library at ni.com. Select **Tools»Synchronize with ni.com Icon Library** to display the Synchronize with Icon Library dialog box and synchronize the LabVIEW Data\Glyphs directory with the most recent glyphs in the Icon Library.

Select **Tools»List Glyphs and Icon Templates** to display all available icon templates and glyphs as well as the locations of those files on disk.

Select **Edit»Import Glyph from File** to display a file dialog box where you can locate and import a specific glyph on file.

Using Icon Templates

You can use the Icon Editor dialog box to create or modify *icon* templates. The **Templates** page of the **Icon Editor** dialog box displays all .png, .bmp, and .jpg files in the LabVIEW Data\Icon Templates directory. You can save icon templates you create to this directory so that they appear in the **Templates** page of the **Icon Editor** dialog box. Use the templates in the LabVIEW Data\Icon Templates\VI directory for VIs and custom controls. Use the templates in the LabVIEW Data\Icon Templates\Library directory for project libraries, classes, and XControls.



NOTE

If you do not use a template, consider adding a border to the icon. Double-click the Rectangle tool on the right side of the Icon Editor dialog box to add a one-pixel border to the icon in the **Line Color**.

If an icon template changes on disk, an icon referencing the template updates automatically when you next display the icon in the Icon Editor dialog box.

If you open a third-party VI, you can choose whether to save the corresponding icon template as a template for future use. Use the **Templates** page of the Icon Editor Properties dialog box to configure settings for saving third-party icon templates.

Select **Tools»List Glyphs and Icon Templates** to display all available icon templates and glyphs as well as the locations of those files on disk.

Select **Edit»Import Glyph from File** to display a file dialog box where you can locate and import a glyph to use as a template or an icon.

Using Layers

Use layers to modify overlapping parts of an *icon* separately. LabVIEW saves layer information about an icon with the corresponding VI or library.

You can use the **Layers** page of the Icon Editor dialog box to configure the name, opacity, visibility, and order of the layers of an icon. Select **Layers»Show Layers Page** in the **Icon Editor** dialog box to display the **Layers** page.

The **Layers** page of the Icon Editor dialog box is separated into three sections. The first section includes the **Icon Text** layer. This layer represents all text you enter on the **Icon Text** page of the **Icon Editor** dialog box. The **Icon Text** layer is always the uppermost layer of the icon.

The second section of the **Layers** page includes all user layers. The Icon Editor dialog box creates a new user layer for each glyph you add to the icon and for each non-consecutive use of the editing tools on the right side of the dialog box. Select **Layers»Create New Layer** to create a new user layer during consecutive uses of the editing tools.

You can select a user layer by clicking the corresponding preview on the **Layers** page or by using the Move tool to select a layer in the **Preview** area of the Icon Editor dialog box.

When you select a layer, the rest of the icon becomes dimmed so you can identify which layer you selected. After you select a layer, you can use the editing tools on the right side of the **Icon Editor** dialog box to modify that layer separately from all other layers.

You also can change the order of user layers. Select one or more user layers and click the **Move Up** or **Move Down** buttons to change the order of the layers. Use options in the **Layers** menu to combine user layers you select or to combine all user layers of the icon. Select **Edit»Clear User Layers** to delete all user layers of the icon. Place a checkmark in the **Merge all layers on commit** checkbox on the **Layers** page of the Icon Editor Properties dialog box to merge all layers when you close the Icon Editor dialog box. Because LabVIEW saves layer information about an icon with the corresponding VI or library, merging icon layers reduces the size of the VI or library on disk.

The third section of the **Layers** page includes the **Icon Template** layer. This layer represents the icon template you select on the **Templates** page of the Icon Editor dialog box. The **Icon Template** layer is always the bottommost layer of the icon.

The **Layers** page displays the preview, name, opacity, and visibility of each layer. You cannot change the name of the **Icon Text** or **Icon Template** layers. However, you can change the name of all user layers. The default name for a glyph layer is the filename of the glyph without the file extension. The default name for a layer corresponding to a use of the editing tools is **Tool**.

Click and drag the blue **Opacity** slider for a layer to make the layer either more transparent or more opaque. Click the **Visible?** icon  to hide or display the layer.

Saving VIs

Select **File»Save** to save a VI. You can save VIs as individual files or you can group several VIs together and save them in an LLB. LLB files end with the extension `.llb`. NI recommends that you save VIs as individual files, organized in directories, especially if multiple developers are working on the same project.

NI recommends that you save your VIs in *certain directories*. To make VIs easily accessible, limit the number and levels of directories you use in a project.

Advantages of Saving VIs as Individual Files

The following list describes reasons to save VIs as individual files:

- You can use the file system to manage the individual files.
- You can use subdirectories.
- You can store VIs and controls in individual files more robustly than you can store your entire project in the same file.
- You can use third-party source control providers.

Create a directory for all the VIs for one application and save the main VIs in this directory and the subVIs in a subdirectory. If the subVIs have subVIs, continue the directory hierarchy

downward. When you create the directory, organize the VIs and subVIs modularly according to the functionality of the subVIs.

Advantages of Saving VIs as LLBs

The following list describes reasons to save VIs as LLBs:

- You can transfer an LLB to other platforms more easily than you can transfer multiple individual VIs. You also can ensure that the user receives all the files needed.
- You can slightly reduce the file size of your project because LLBs are compressed to reduce disk space requirements.
- You can mark VIs in an LLB as top-level so that when you open the LLB, LabVIEW automatically opens all the top-level VIs in that LLB.

If you use LLBs, consider dividing your application into multiple LLBs. Put the high-level VIs in one LLB and set up other LLBs to contain VIs separated by function. Saving changes to a VI in an LLB takes longer than saving changes to an individual VI because the operating system must write the changes to a larger file. Saving changes to a large LLB also can increase memory requirements and decrease performance. Try to limit the size of each LLB to approximately 1 MB.

Managing VIs in LLBs

Use the LLB Manager window, available by selecting **Tools»LLB Manager**, to simplify copying, renaming, and deleting files within LLBs. You also can use this tool to create new LLBs and directories and convert LLBs to and from directories. Creating new LLBs and directories and converting LLBs to and from directories is important if you need to manage your VIs with source control tools.

Before you use the LLB Manager window, close all VIs that might be affected to avoid performing a file operation on a VI already in memory.



NOTE

You cannot cancel changes you make in the LLB Manager window.

(Windows) You also can double-click a `.llb` file in Windows Explorer to display the LLB Manager window so you can open, move, copy, rename, and delete files in the LLB.

Naming VIs

When you save VIs, use descriptive names. Descriptive names, such as `Temperature Monitor.vi` and `Serial Write & Read.vi`, make it easy to identify a VI and know how you use it. If you use ambiguous names, such as `VI_1.vi`, you might find it difficult to identify VIs, especially if you have saved several VIs.

Consider whether your users will run the VIs on another platform. Avoid using characters that some operating systems reserve for special purposes, such as \ : / ? * < > and #.



NOTE

If you have several VIs of the same name saved on your computer, carefully organize the VIs in different [project libraries](#) or LLBs to avoid LabVIEW referencing the wrong subVI when running the top-level VI. Refer to the [LabVIEW Style Checklist](#) for more tips on how to name VIs, LLBs, and directories.

Saving for a Previous Version

You can save VIs, LabVIEW projects, and project libraries for a previous version of LabVIEW. Refer to *Saving for a Previous Version* for details.

Automatic Saving for Recovery

In the event of an irregular shutdown or system failure, LabVIEW backs up any modified VI (.vi), VI template (.vit), control (.ctl), control template (.ctt), project (.lvproj), project library (.lvlib), XControl (.xctl), or LabVIEW class (.lvclass) files open at the time of the shutdown or failure to a temporary location.



NOTE

If you are entering text anywhere in a file at the time of the automatic save, LabVIEW does not save that file. This behavior prevents automatic saving from disrupting text entry.

Select **Tools»Options** and select **Environment** from the **Category** list to enable or disable saving for recovery and to specify how often LabVIEW should back up files. If you enable automatic saving, LabVIEW backs up files before you run a VI. You also can configure LabVIEW to back up files at periodic intervals.

LabVIEW stores backed up files in the LVAutoSave subdirectory of the [default data directory](#). Do not access or modify the files in the LVAutoSave directory manually.

If LabVIEW automatically saves files before an irregular shutdown or system failure, the Select Files to Recover window appears the next time you launch LabVIEW. Select the files you want to recover and click the **Recover** button. If you do not want to recover any files, deselect all files and click the **Discard** button. Click the **Cancel** button to move all selected files to the LVAutoSave\archives subdirectory of the default data directory. You can delete the contents of this subdirectory when you no longer need the backup files.



NOTE

Selecting a project or library for recovery does not automatically recover changes made to files in that project or library. To recover a modified project or library and all of its modified contents, make sure there is a checkmark beside the owning .lvproj or .lvlib file and beside each file the project or library contains.

Automatically saved files do not overwrite existing manually saved files. LabVIEW retains automatically saved files only until you manually save, discard, or archive the files.



RELATED INFORMATION

[Saving for a Previous Version](#) on page 938

You can save VIs, LabVIEW projects and project libraries for a previous version of LabVIEW.

Customizing VIs

You can configure VIs and subVIs to work according to your application needs. For example, if you plan to use a VI as a subVI that requires user input, configure the VI so that its front panel appears each time you call it.

You can configure a VI in many ways, either within the VI itself or programmatically by using the [VI Server](#).

Configuring the Appearance and Behavior of VIs

Select **File»VI Properties** to configure the appearance and behavior of a VI. Use the **Category** pull-down menu at the top of the VI Properties dialog box to select from several different option categories.

Customizing Menus

You can create custom menus for every VI you build, and you can configure VIs to show or hide menu bars. Show and hide menu bars by selecting **File»VI Properties**, selecting **Windows Appearance** from the **Category** pull-down menu, clicking the **Customize** button, and placing or removing a checkmark from the **Show Menu Bar** checkbox.

Configuring menus includes creating the menu and providing the block diagram code that executes when the user selects the various menu items.



NOTE

Custom menus appear only while the VI runs.

Creating Menus

You can build custom menus or modify the default LabVIEW menus statically when you edit the VI or programmatically when you run the VI.

To add a custom menu bar to a VI rather than the default menu bar, select **Edit»Run-Time Menu** and create a menu in the Menu Editor dialog box. LabVIEW creates a run-time menu .rtm file. After you create and save the .rtm file, you must maintain the same relative path between the VI and the .rtm file. You also can create a custom run-time

shortcut menu by right-clicking a control and selecting **Advanced»Run-Time Shortcut Menu»Edit**. This option opens the Shortcut Menu Editor.

You can use the Menu Editor and Shortcut Menu Editor dialog boxes to build custom menus either with application items, which are menu items LabVIEW provides in the default menu, or with user items, which are menu items you add. LabVIEW defines the behavior of application items, but you control the behavior of user items with the block diagram. You also can use the **Menu Editor** and **Shortcut Menu Editor** dialog boxes to associate a custom .rtm file with a VI or control. When the VI runs, it loads the menu from the .rtm file.

Use the Menu Editor and Shortcut Menu Editor dialog boxes to customize menus when editing a VI. Use the Menu functions to customize menus programmatically at run time. The functions allow you to insert, delete, and modify the attributes of user items. You can only add or delete application items because LabVIEW defines the behavior and state of application items.

Shortcut Menus in Run Mode

When a VI is running, or is in run mode, all front panel objects have an abridged set of shortcut menu items by default. Use the abridged shortcut menu items to cut, copy, or paste the contents of the object, to set the object to its default value, or to read the description of the object.



Note You cannot reinitialize, cut, or paste the data of an indicator in run mode. The **Reinitialize to Default Value**, **Cut Data**, and **Paste Data** options are available only for controls in run mode.

Some of the more complex controls have additional options. For example, the knob shortcut menu includes items to add a needle and to change the display of scale markers.

You can [customize](#) the run-time shortcut menu for each control you include in a VI. To customize a shortcut menu, right-click a control and select **Advanced»Run-Time Shortcut Menu»Edit** from the shortcut menu to display the Shortcut Menu Editor dialog box. Use the **Shortcut Menu Editor** dialog box to associate the default shortcut menu or a [customized shortcut menu file](#) .rtm with the control. You can customize shortcut menus [programmatically](#).

You also can [add shortcut menus](#) to front panels. To add a shortcut menu to the front panel, use the Shortcut Menu Activation and Shortcut Menu Selection pane events.

You also can [disable](#) the run-time shortcut menu on a control.



Note Custom run-time shortcut menus appear only while the VI runs.

You also can use the Mouse Down? and Shortcut Menu Activation? filter events to display a shortcut menu when you left-click a control. To perform this action, modify the data returned by the **Button** event data field of the Mouse Down? filter event. The value of the

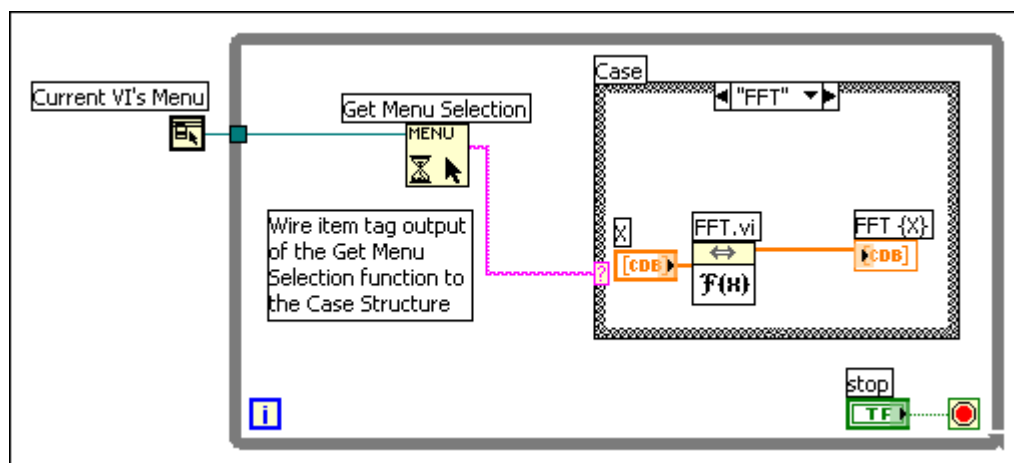
left mouse button is 1, and the value of the right mouse button is 2. In order to display the shortcut menu when you left-click a control, change the **Button** event data field to 2 so that LabVIEW treats a left-click like a right-click.

Menu Selection Handling

When you create a custom menu, you assign each menu item a unique, case-insensitive string identifier called a tag. When the user selects a menu item, you retrieve its tag programmatically using the Get Menu Selection function. LabVIEW provides a handler on the block diagram for each menu item based on the tag value of each menu item. The handler is a While Loop and Case structure combination that allows you to determine which, if any, menu is selected and to execute the appropriate code.

After you build a custom menu, build a Case structure on the block diagram that executes, or handles, each item in the custom menu. This process is called menu selection handling. LabVIEW handles all application items implicitly.

In the following block diagram, the Get Menu Selection function reads the menu item the user selects and passes the menu item into the Case structure, where the menu item executes.



If you know that a certain menu item takes a long time to process, wire a Boolean control to the **block menu** input of the Get Menu Selection function and set the Boolean control to TRUE to disable the menu bar so the user cannot select anything else on the menu while LabVIEW processes the menu item. Wire a TRUE value to the Enable Menu Tracking function to enable the menu bar after LabVIEW processes the menu item.

You also can [use the Event structure](#) to handle menu events.

Running and Debugging VIs

To run a VI, you must wire all the subVIs, functions, and structures with the correct data types for the terminals. Sometimes a VI produces data or runs in a way you do not expect.

You can use LabVIEW to configure how a VI runs and to identify problems with block diagram organization or with the data passing through the block diagram.



RELATED INFORMATION

[Running VIs](#) on page 226

[Correcting Broken VIs](#) on page 337

[Setting up the VI Revision History](#) on page 227

[Debugging Techniques](#) on page 329

[Handling Errors](#) on page 338

[Running VIs](#) on page 947

Running VIs

Running a VI executes the operation for which you designed the VI. You can run a VI if the **Run** button on the toolbar appears as a solid white arrow, shown as follows.



The solid white arrow also indicates you can use the VI as a *subVI* if you [create a connector pane](#) for the VI.

A VI runs when you click the **Run** or **Run Continuously** buttons or the single-stepping buttons on the block diagram toolbar. While the VI runs, the **Run** button changes to a darkened arrow, shown as follows, to indicate that the VI is running.



You cannot edit a VI while the VI runs.

Clicking the **Run** button runs the VI once. The VI stops when the VI completes its data flow. Clicking the **Run Continuously** button, shown as follows, runs the VI continuously until you stop it manually.



Clicking the [single-stepping](#) buttons runs the VI in incremental steps.



NOTE

The **Abort Execution** button  stops the VI immediately, before the VI finishes the current iteration. Aborting a VI that uses external resources, such as external hardware, might leave the resources in an unknown state by not resetting or releasing them properly. Design VIs with a stop button to avoid this problem.

Configuring How a VI Runs

Select **File»VI Properties** and select **Execution** from the **Category** pull-down menu to [configure how a VI runs](#). For example, you can configure a VI to run immediately when it opens or to pause when called as a subVI. You also can [configure the VI to run at different priorities](#). For example, if it is crucial that a VI runs without waiting for another operation to complete, configure the VI to run at time-critical (highest) priority.



Note Incorrectly configuring a VI to run at different priorities can cause unexpected execution. You do not need to configure most VIs to run at different priorities.

If you do not want a VI to run immediately, but you also do not want users to be able to edit the VI, you can change the VI to run mode by selecting **Operate»Change to Run Mode**. In run mode, all front panel objects have an abridged set of shortcut menu items. Users can access only front panel controls, toolbar buttons that run and stop the VI, and the menu bar. To allow editing again, select **Operate»Change to Edit Mode**.

Setting up the VI Revision History

Use the History window to display the development history of a VI, including revision numbers. As you make changes to the VI, record and track them in the **History** window. Select **Edit»VI Revision History** to display the **History** window. You also can [print](#) the revision history or save it to an HTML, RTF, or text file.

You can [set options](#) for, [add comments](#) to, [view](#), and [reset](#) the revision history.

Revision Numbers

The revision number is an easy way to track changes to a VI. The revision number starts at one and increases incrementally every time you save the VI. To display the current revision number in the title bar of the VI and the title bar of the History window, select **Tools»Options**, select **Revision History** from the **Category** list, and place a checkmark in the **Show revision number in titlebar** checkbox.

Revision numbers are independent of comments in the History window. Gaps in revision numbers between comments indicate that you saved the VI without a comment.

Because the history is strictly a development tool, LabVIEW automatically removes the history when you [remove the block diagram](#) of a VI. The History window is not available in the run-time version of a VI. The General page of the VI Properties dialog box displays the revision number, even for VIs without block diagrams. Click the **Reset** button in the **History** window to erase the revision history and reset the revision number.

Preventing Undefined Data

Do not rely on special values such as NaN, Inf, or empty arrays to determine if a VI produces undefined data. Instead, confirm that the VI produces defined data by making the VI report an error if it encounters a situation that is likely to produce undefined data.

For example, if you create a VI that uses an incoming array to auto-index a For Loop, determine what you want the VI to do when the input array is empty. Either produce an output error code, substitute defined data for the value that the loop creates, or use a Case structure that does not execute the For Loop if the array is empty.



RELATED INFORMATION

[Block Diagram Objects](#) on page 126

[Grouping Data with Arrays and Clusters](#) on page 460

[Processing Individual Elements in an Array or a Collection with a Loop](#) on page 1266

[Case Structures: Executing a Section of Code Based on Input Values](#) on page 420

Avoiding Incorrect Dependencies

When a VI depends on other files, the top-level VI saves paths to its subVIs. This capability allows LabVIEW to load all subVIs when it loads the top-level VI.

If LabVIEW cannot find a subVI, it searches other locations on disk to find the file. [Cross-linking](#) occurs when LabVIEW locates an incorrect subVI with the same filename.

Consider the following suggestions to avoid loading incorrect dependencies:

- Place all of your VIs in a project, and use the [Project Explorer](#) window to move files on disk. When you move an individual subVI, LabVIEW modifies the VI that depends on that subVI and saves the changes.
- When you move a subVI on disk and your files are not in the Project Explorer window, make sure you move the directory that contains all of the VIs involved in your application so that the [relative path](#) to the files remains the same. However, you do not need to move the subVIs that link to top-level VIs if you reference a subVI in a [directory that installs with LabVIEW](#), such as `vi.lib`, `instr.lib`, and `user.lib`.
- If you have two VIs with the same filename, use project libraries to qualify the names of the VIs so that the name of the project library becomes part of the filename. If your application refers to two VIs with the same name, LabVIEW might load the incorrect VI or try to load both VIs. If LabVIEW tries to load two files with the same filename, the Conflict Resolution dialog box appears with a list of conflicts.

Executing a VI before or after a Build

You can execute a VI before or after a build for a stand-alone application, .NET Framework assembly, shared library, or source distribution.

(Real-Time Module) You can execute a VI before or after a build for a stand-alone real-time application.

The following examples demonstrate use cases for executing a pre-build VI, which is a VI that runs before a build:

- Deleting a specific file or all files in the destination directory.
- Copying the latest versions of the VIs in a LabVIEW project from the [source control](#) to the local directory.



NOTE

Changes you make when VIs are open do not save. If you copy VIs from source control, make sure you close the VIs before you make a copy before a build.

The following examples demonstrate use cases for running a post-build VI, which is a VI that runs after a build:

- Copying build files to a different location, such as an export directory, from where you save the build.
- Running tests on the build files with a subset of inputs to test that the build returns the appropriate outputs.
- Running the build files with operations for command line parameters to test the build.

If the VIs you want to execute before or after a build do not already exist in the project, click the appropriate **Generate VI** button on the **Pre/Post Build Actions** page to create these VIs.

Connector Pane Changes That Reset the Prototype of a VI (Shared Libraries)

When you configure a build specification for a shared library, you can use the Define VI Prototype dialog box to customize the prototype that LabVIEW generates for the VI in the shared library. However, LabVIEW does not maintain this custom prototype if you change the connector pane of the VI or any of the controls or indicators assigned to it. Changes to these items cause LabVIEW to restore the default prototype and alert you the next time you try to open the prototype or build the shared library.

The following changes cause LabVIEW to reset the prototype of a VI:

- Changing a control or indicator to a data type that is incompatible with the previous data type. For example, changing a numeric indicator to a string indicator.
- Changing the representation of a control or indicator.
- Adding or removing a control or indicator from the connector pane.
- Modifying items in an enumeration

Cosmetic changes to controls or indicators do not affect custom prototypes.

Picture Plots VIs

Use the Picture Plots VIs to create common types of graphs using the 2D picture control. These graphs include a polar plot, a waveform graph, an XY graph, a Smith plot, a radar plot, and a graph scale.

The Picture Plot VIs use low-level drawing functions to create a graphical display of your data and customize the drawing code to add functionality. These graphical displays are not as interactive as the built-in LabVIEW controls, but you can use them to visualize information in ways the built-in controls currently cannot. For example, you can use the Plot Waveform VI to create a plot with slightly different functionality than built-in waveform graphs.

Using the Polar Plot with Point Options VI as a SubVI

Use the Polar Plot with Point Options VI to draw specific, contiguous quadrants of a polar graph or the entire graph at once. As with the built-in LabVIEW graphs, you can specify the color of the components, include a grid, and specify the range and format for the scales.

The Polar Plot with Point Options VI provides a large amount of functionality in a single VI. Consequently, the VI includes complicated clusters for inputs. You can use default values and custom controls to decrease the complexity of the VI. Instead of creating the default cluster input on the block diagram, copy a custom control from the Simple Polar Plot VI in the `labview\examples\Graphics and Sound\Picture Plots` directory and place it on the front panel.

Using the Plot Waveform and Plot XY VIs as SubVIs

Use the Plot Waveform VI, which emulates the behavior of the built-in waveform graph, to draw waveforms in a variety of styles, including points, connected points, and bars. As with the built-in waveform graphs, you can specify the color of the components, include a grid, and specify the range and format for the scales.

The Plot Waveform VI provides a large amount of functionality in a single VI. Consequently, the VI includes complicated clusters for inputs. You can use default values and custom controls to decrease the complexity of the VI. Instead of creating the default cluster input, copy a custom control from the Waveform and XY Plots VI in the `labview\examples\Graphics and Sound\Picture Plots` directory and place it on the front panel.

The Plot XY VI and the Plot Multi-XY VI are similar to the Plot Waveform VI. You use different controls to specify the cosmetic appearance of the plot because the XY plotting VIs have three additional plot styles—two scatter plot styles and a plot style where a line is drawn at each unique x -position to mark the minimum and maximum y -values for that x -value.

Using the Smith Plot VIs as SubVIs

Use the Smith Plot VIs to study transmission line behavior, such as in the telecommunications industry. A transmission line is a medium through which you transmit energy and signals. A transmission line can be a wire or it can be the atmosphere through which a signal transmits. Transmission lines have an effect on the signal that is transmitting. This effect, called the impedance of the transmission line, can attenuate or phase shift an AC signal.

The impedance of the transmission line is a measure of the resistance and the reactance of the line. The impedance, z , is commonly listed as a complex number of the form $z = r + j x$, where both resistance r and reactance x are components.

Use the Smith Plot VIs to display impedances of transmission lines. The plot consists of circles of constant resistance and reactance.

You can plot a given impedance, $r + j x$, by locating the intersection of the appropriate r circle and x circle. After you plot the impedance, use the Smith Plot VIs as visual aids to match impedance and to calculate the reflection coefficient of a transmission line.

The Smith Plot VIs provide a large amount of functionality in each single VI. Consequently, many of these VIs include complicated clusters for inputs. You can use default values and custom controls to decrease the complexity of the VIs. Instead of creating the default cluster input, copy a custom control from the Smith Plot example VIs in the `labview\examples\Graphics and Sound\Picture Plots` and place it on the front panel.

To avoid losing detail in the Smith plot, use the Normalize Smith Plot VI to normalize the data. You can pass the data you normalize with the Normalize Smith Plot VI directly to the Smith Plot VI. You usually scale Smith plot data with respect to the characteristic impedance (Z_0) of the system.

Picture Functions VIs

Use the Picture Functions VIs to draw shapes and enter text into a 2D picture control. You can draw points, lines, shapes, and pixmaps. Pixmaps of unflattened data are 2D arrays of color, where each value corresponds to a color or to an index into an array of RGB color values, depending on the color depth.

The first row of the Picture Functions palette contains VIs you use to draw points and lines. A point is a cluster of two 16-bit signed integers that represent the x- and y-coordinates of a pixel.

When you use the Picture Functions VIs, the picture remembers the position of the graphics pen. For most of the Picture Functions VIs, you must specify absolute coordinates

—that is, relative to the origin (0, 0). With the Draw Line VI and the Move Pen VI, you can specify either absolute or relative coordinates. Relative coordinates are relative to the current location of the pen. You can use the Move Pen VI to change the location of the pen without drawing. Only the Draw Point VI, Move Pen VI, Draw Line VI, and Draw Multiple Lines VI change the location of the pen.

The second row of the Picture Functions palette contains VIs you use to draw a shape. Each of these VIs draws a shape in a rectangular area of a picture. You specify a rectangle as a cluster of four values that represent the left, top, right, and bottom pixels.

The third row of the Picture Functions palette contains VIs you use to draw text in a picture. The Get Text Rect VI does not draw any text. Instead, you use it to calculate the size of a bounding rectangle of a string.

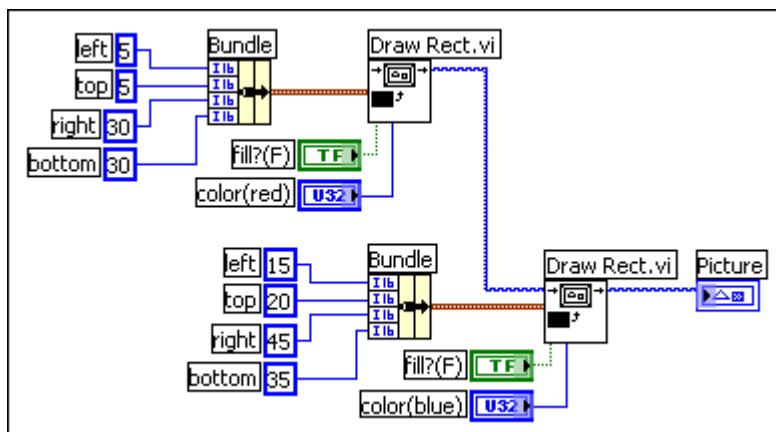
The fourth row of the Picture Functions palette contains VIs you use to draw flattened and unflattened pixmaps in a picture, to apply a mask to an image, to obtain a subset of a source image, and to convert a picture data type to a flattened image data cluster.

The last row of the Picture Functions palette contains the empty picture constant, which you use when you need to start with or make changes to an empty picture. The last row of the palette also contains VIs you use to convert red, green, and blue values to the corresponding RGB color and to convert a color to its respective red, green, and blue components.

You can wire the pictures you create with the Picture Functions VIs only to a 2D picture control or to the **picture** input of a Picture Functions VI. LabVIEW draws the picture when it updates the 2D picture control on an open front panel.

Each Picture Functions VI concatenates its drawing instruction to the drawing instructions wired to the **picture** input and returns the concatenated drawing instructions in the **new picture** output.

The following block diagram uses the Draw Rectangle VI to draw two overlapping rectangles.



Creating and Modifying Colors with the Picture Functions VIs

Many of the Picture Functions VIs have a **color** input to modify the color of the shapes and text. The easiest way to specify a color is to use a color box constant and click the constant to select a color.

To create colors as the result of calculations rather than with color box constants, you need to understand how a color box specifies a color using a numeric value.

A 32-bit signed integer represents a color, and the lower three bytes represent the red, green, and blue components of the color. For a range of blue colors, create an array of 32-bit integers where the blue values of each element change and are greater than the red and green values. To create a range of gray colors, create an array of 32-bit integers where the red, green, and blue values of each element are the same.

Graphics Formats VIs

Use the Graphics Formats VIs to read data from and write data to several standard graphics file formats, including bitmap .bmp, Portable Network Graphics .png, and Joint Photographic Experts Group .jpg files. You can use these VIs to perform the following tasks:

- Read data from a graphics file for display in a 2D picture control.
- Read data for image manipulation.
- Write an image for viewing in other applications.

LabVIEW stores bitmap data in 2D arrays in which each point varies depending on the color depth. For example, in a black-and-white, or 1-bit, image, each point is Boolean. In 4-bit and 8-bit images, each point is an index in a color table. For 24-bit true-color images, each point is a mixture of red, green, and blue (RGB) values.

The VIs that read and write graphics files work with data in a simple, flattened format that is closer to the way graphics files are written to disk, with the data stored in a 1D array. These graphics files are pixmaps, which are similar in concept to bitmaps. You can display this flattened data directly using the Draw Flattened Pixmap VI.

To manipulate the data as a 2D array, you can convert it to the appropriate format using the Unflatten Pixmap VI and Flatten Pixmap VI.

Sound VIs

Use the Sound VIs to integrate sound files and functions into your VIs. You can use these VIs to perform the following tasks:

- Create VIs that play sound files, such as a recorded warning, when users perform certain actions.

- Create a VI that plays a sound file when the VI starts or finishes running or when you reach a certain point in the VI.
- Configure a sound input device to acquire sound data. Use the Sound Input VIs to acquire the sound data. You also can read any sound information coming through the device.
- Configure a sound output device to accept sound data from other Sound VIs. You can control the volume of the sound going through the device, play or pause the sound, and clear the sound from your system.

(Windows) You must have DirectX 8.0 or later to use the Sound VIs. (Linux) You must have the Open Sound System (OSS) driver to use the Sound VIs.

Sound Output Tasks

Use the Sound Output VIs to generate and control a sound on an output sound device, such as a speaker or headphones. For simple operations, use the Play Waveform Express VI to play the sound data with the sound output device.

For more complex operations, use the Sound Output Configure VI to create and configure a new sound output task ID. After you create the task ID, use the Sound Output Start VI to control when the sound starts playing, use the Sound Output Stop VI to stop the sound, use the Sound Output Set Volume VI to control the volume of the sound, use the Sound Output Write VI to send the sound to a sound device, and use the Sound Output Info VI to query the sound device for information about the task ID. Use the Sound Output Clear VI to release any resources the task ID used.

Sound Input Tasks

Use the Sound Input VIs to acquire sound data from a sound input device, such as a microphone, into LabVIEW. Use the Acquire Sound Express VI for simple sound acquisition operations from a sound input device. For more complex operations, use the Sound Input Configure VI to create and configure a new sound input task ID.

After you create the task ID, you can change the state of the sound to running or idle using the Sound Input Start VI and the Sound Input Stop VI, respectively. Use the Sound Input Read VI to read from a sound input device. Use the Sound Input Clear VI to release any resources the task ID used.

Sound Data

LabVIEW stores sound data as an array of waveforms. Each waveform in the array represents a single channel. For example, an array with two waveform elements contains stereophonic, or two-channel, data. Because each channel plays simultaneously, ensure that the **dt** and **to** values and the number of elements in the **Y** array match in each waveform.

The **Y** array of the waveform contains Pulse Code Modulated (PCM) data. In PCM data, each element in the **Y** array is proportional to the amplitude of the signal. If the **Y** array data type is a 16-bit signed or 32-bit signed integer, the values range from the minimum to the

maximum value for the data type and are centered at zero. A buffer of elements each with the value of zero represents silence. If the **Y** array data type is a 8-bit unsigned integer, the values range from 0–255 and are centered at 127. A buffer of elements each with the value of 127 represents silence rather than a buffer of elements each with the value of zero. This might cause an initial clicking sound with 8-bit unsigned integers. For floating-point data, the values range from -1.0 to 1.0, inclusively. The data is coerced into that range. There is no scaling.

Task IDs

The Sound Output Configure, Sound Input Configure, and Play Sound File VIs return task IDs, which are numbers the VIs generate to refer to the sound task at hand. Subsequent sound VIs on the block diagram use the task ID to manipulate the data in a particular sound operation. You must clear the task ID from memory before you can rerun the VI you created. Use the Sound Output Clear VI to clear a task ID for a sound output operation. Use the Sound Input Clear VI to clear a task ID for a sound input operation.

Sound Devices

Sound devices are sound cards or other hardware attached to the computer. The value of the **Device ID** input of the Sound Output Configure, the Sound Input Configure, and the Play Sound File VIs specifies which sound device to use. Enter 0 in **Device ID** for the default sound device on the computer. Enter 1, 2, 3, and so on, for other devices.

Sound Buffers

Sound input and sound output operations use a buffer between LabVIEW and the sound device, which helps provide a smooth transmission of the sound data. Use the Sound VIs that read, write, and configure sound data to manipulate the sound buffer. The tasks these VIs create transfer the sound data between the buffer and the sound device. The task continues running until you stop the task or an error occurs.

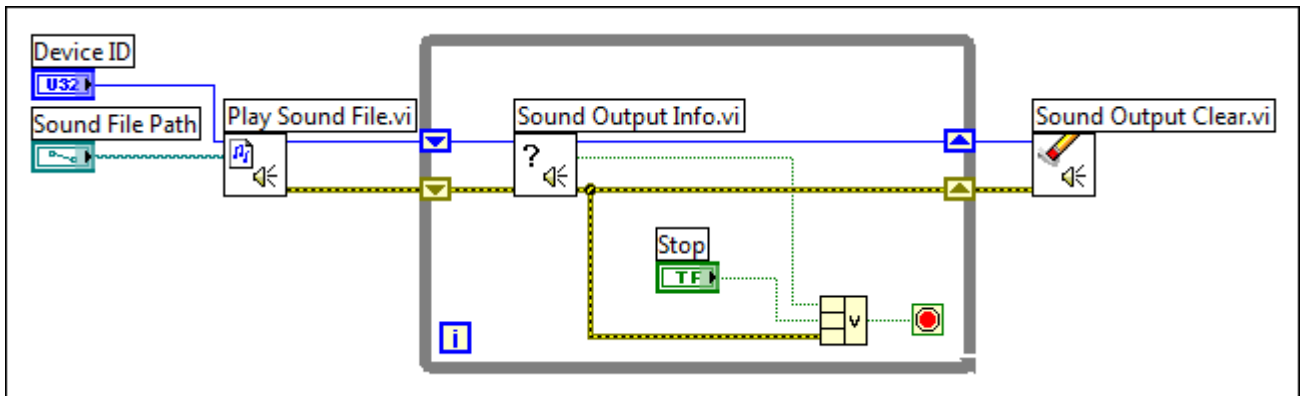
Input operations transfer sound data from the input device to the buffer to the block diagram. Use the Sound Input Read VI to acquire the data from the device. If there is enough data in the buffer to read all the samples requested, LabVIEW reads the samples immediately. If there are more samples requested than are ready, the VI pauses execution until the request completes.

Output operations transfer sound data from the block diagram to the buffer to the output device. Use the Sound Output Write VI to transfer data to the buffer in a sound output operation. If the buffer has enough room, LabVIEW copies the data to the buffer. If the buffer does not have enough room and the buffer is configured for continuous mode, the VI pauses execution until it finishes copying all the data to the buffer.

Configuring the sound data sets the format of the buffered data. LabVIEW automatically converts the format of LabVIEW data to match the buffer data format. For example, if you configure an output task to use stereophonic data, but you write data with only one channel, LabVIEW sends the data you write to both of the stereophonic channels.

Playing a Sound File

Use the Play Sound File VI and a While Loop to continually play a sound file, as shown in the following block diagram. The Play Sound File VI reads the file specified in the **Sound File Path** input and starts playing the sound through the sound device the **Device ID** input specifies. The VI returns a *task ID*, and the While Loop continually passes the task ID through until the sound finishes playing, an error occurs, or a users presses a stop button on the front panel. The Sound Output Clear VI clears the task ID from the computer memory.

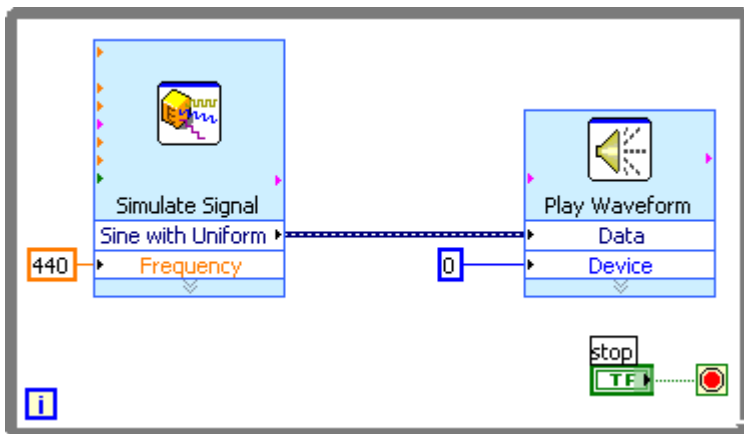


Use the Sound VIs to configure how the sound file plays, such as the bit rate of the file and the channels through which the sound plays (mono, stereo, and so on). You also can pause, stop, and play the file at run time.

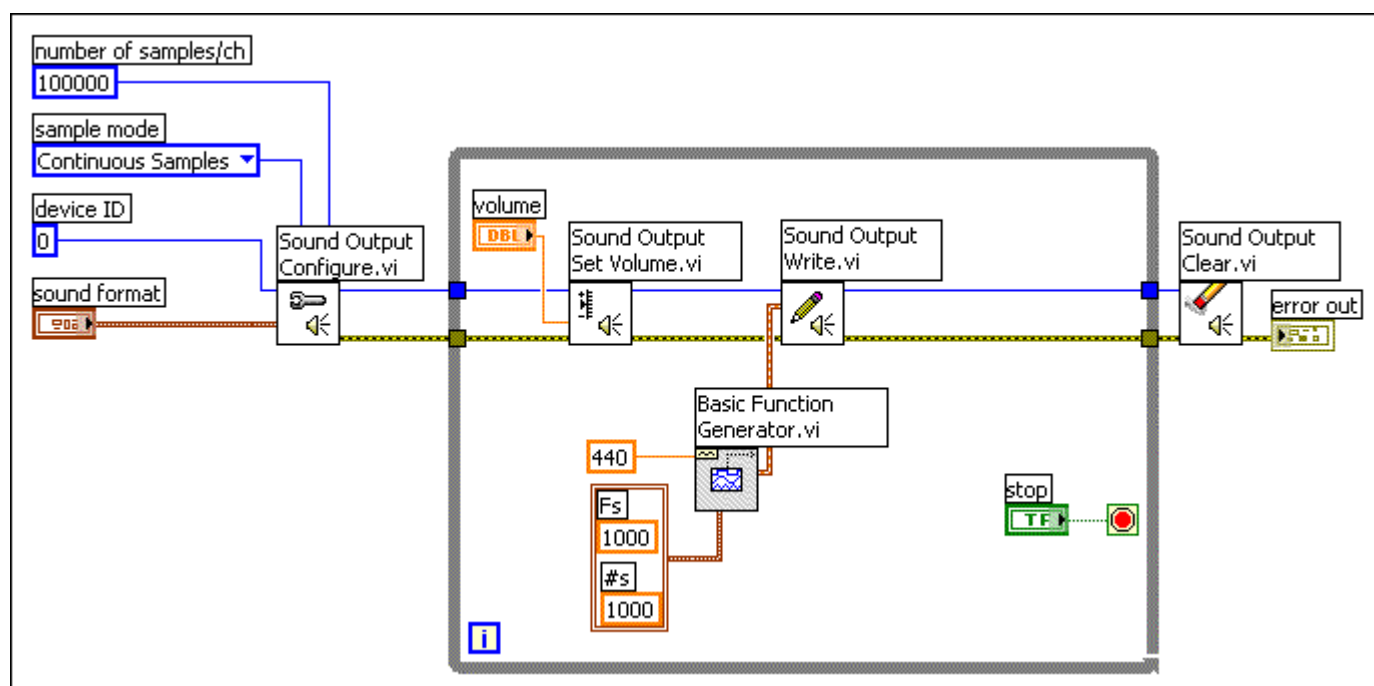
Refer to the Sound File to Sound Output VI in the labview\examples\Graphics and Sound\Sound directory for an example of generating sound data.

Generating and Playing Sound Data

Use the Waveform Generation VIs and the Simulate Signal Express VI to generate sound data, as shown in the following block diagram. The Simulate Signal Express VI generates a sine wave, and the Play Waveform Express VI plays the generated sound data through the sound device the **Device** input specifies.



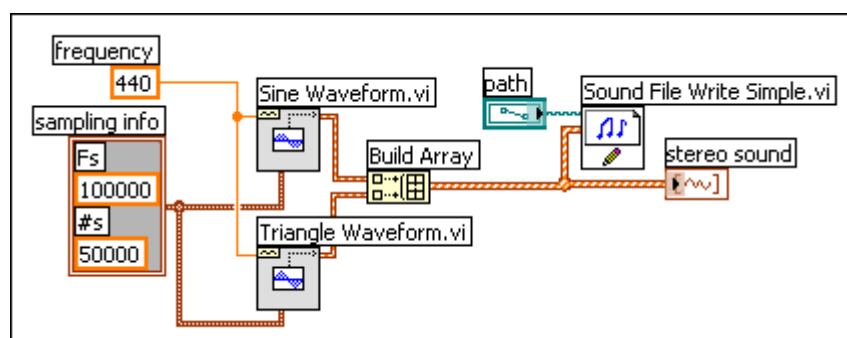
Use the Sound Output VIs for more control of the sound data you generate. For example, use the Sound Output Configure VI to control the bit rate, sample rate, and number of channels of the sound and use the Sound Output Set Volume VI to control the volume of the sound, as shown in the following block diagram.



Refer to the Generate Sound VI in the `labview\examples\Graphics and Sound\Sound` directory for an example of generating sound data.

Writing Data to a Sound File

Use the Sound File Write Simple and the Sound File Write VIs to write the data you acquire or generate to a sound file. The following block diagram uses the Sound File Write Simple VI to save a stereophonic sound that consists of a sine wave tone and a triangle wave tone of 50,000 samples each to the file the **path** input specifies.



Use the Sound File Write VI for more control over the creation of a sound file. The Sound File Write VI requires you to open a reference to an existing sound file or create a new sound file with the Sound File Open VI and close the reference to the sound file with the Sound File Close VI. If you do not close the sound file, the sound file stays in memory and is not accessible from other applications or to other users.

Refer to the Sound Input to File VI in the `labview\examples\Graphics and Sound\Sound` directory for an example of acquiring a sound and saving it to a sound file.

Graphics and Sound VIs

Use the Graphics & Sound VIs to display or modify graphics and sound in VIs.



RELATED INFORMATION

[Picture Plots VIs](#) on page 230

[Picture Functions VIs](#) on page 231

[Graphics Formats VIs](#) on page 233

[Sound VIs](#) on page 233

[3D Picture Control VIs](#) on page 239

[Displaying Pixels Not Visible in a 2D Picture Control](#) on page 1233

[Using Sound Input and Sound Output Devices](#) on page 1233

[Creating a 3D Scene with the 3D Picture Control](#) on page 1234

[Creating Relationships between 3D Objects](#) on page 1241

[Configuring a 3D Scene Window](#) on page 1244

Documenting and Printing VIs

You can use LabVIEW to document and print VIs and information about VIs. Document VIs to record the purpose of the VIs and to give instructions for using the VIs. Print VIs and VI documentation to keep a record of information about the VIs. You also can save VIs and VI documentation to HTML, RTF, or text files.

LabVIEW includes three primary ways to print VIs. You can choose to print only the active window, include more comprehensive information about a VI such as VI documentation, or programmatically print data a VI returns. The Print dialog box and the VI Properties dialog box include options you can use to configure what LabVIEW prints.



RELATED INFORMATION

[Documenting VIs](#) on page 239

[Printing VIs](#) on page 240

[Printing VIs](#) on page 240

[Documenting VIs](#) on page 239

[Printing VIs](#) on page 240

[Documenting VIs](#) on page 1440

[Generating Reports](#) on page 1457

3D Picture Control VIs

The 3D Picture Control VIs appear on the Graphics & Sound palette. Use the 3D Picture Control VIs to design a scene that consists of objects and the lighting, transformations, and other characteristics you apply to the objects. After you construct the scene, wire the object or objects to the 3D picture front panel control and run the VI to generate the scene.

Use the following properties and methods with the 3D Picture Control VIs to create a 3D scene:

- The SceneObject properties and SceneObject methods place 3D objects in the scene and assign the objects geometries, transformations, and other characteristics.
- The SceneClipPlane properties specify planes in the scene in which an object appears or is cut off.
- The SceneGeometry properties and SceneGeometry methods specify the geometrical form an object takes.
- The SceneTexture properties and SceneTexture methods apply textures to 3D objects.
- The SceneWindow properties render the scene in a separate window, configure the window, and set the interaction of the camera with the scene.
- The SceneLight properties configure the light sources for the scene.

Refer to the 3D Model of Solar System VI in the `labview\examples\Graphics and Sound\3D Picture Control` directory for an example of creating a 3D scene with the 3D Picture Control VIs.



RELATED INFORMATION

[Front Panel Controls and Indicators](#) on page 589

Documenting VIs

You can use LabVIEW to document a finished VI and create instructions for users of the VI. You can view the documentation within LabVIEW, print it, and save it to HTML, RTF, or text files.

To create effective documentation for VIs, create VI and object descriptions that describe the purpose of the VI or object and give users instructions for using the VI or object.

Create, edit, and view VI descriptions by selecting **File»VI Properties** and selecting **Documentation** from the **Category** pull-down menu. Create, edit, and view object descriptions by right-clicking the object and selecting **Description and Tip** from the shortcut menu. Tip strips are brief descriptions that appear when you move the cursor over a front panel object while a VI runs. If you do not enter a tip in the Description and Tip dialog box, no tip strip appears. The object description appears in the Context Help window when you move the cursor over the object. You can view the VI description in the **Description and Tip** dialog box.

Printing VIs

Print or save VIs and VI documentation to keep a record you can refer to later. You can use the following primary ways to print VIs:

- Select **File»Print Window** to print the contents of the active window.
- Select **File»Print** to print more comprehensive information about a VI, including information about the front panel, block diagram, subVIs, controls, VI history, and so on.
- Programmatically print or save a VI or a report that contains VI documentation or data the VI returns.

You can configure how LabVIEW prints a VI when you select **File»Print Window** or when you *print programmatically* by selecting **File»VI Properties**.

Printing the Active Window

You can print the contents of the active front panel or block diagram window by selecting **File»Print Window**. LabVIEW prints the workspace of the active window, including any objects not in the visible portion of the window. LabVIEW does not print the title bar, menu bar, toolbar, or scroll bars.

Printing Documentation

You can print VI documentation or save it to HTML, RTF, or text files by selecting **File»Print**. You can select whether to print documentation for a single VI or for multiple VIs. You also can select a built-in documentation style or create a custom style for documentation.

The documentation you create can include the following items:

- Icon and connector pane
- Front panel and block diagram
- Controls, indicators, and data type terminals
- Labels and captions for controls and indicators
- VI and object descriptions
- VI hierarchy
- List of subVIs
- Revision history



NOTE

The documentation you create for certain types of VIs cannot include all the previous items. For example, a polymorphic VI does not have a front panel or a block diagram, so you cannot include those items in the documentation you create for a polymorphic VI.

Saving Documentation to HTML, RTF, or Text Files

You can save VI documentation to HTML, RTF, or text files. You can import HTML and RTF files into most word processing applications, and you can use HTML and RTF files to [create compiled help files](#). You also can use the HTML files LabVIEW generates to display VI documentation on the Web. You also can print and save documentation to HTML, RTF, and text files [programmatically](#).

When you save documentation to an RTF file, specify if you want to create a file suitable for help files or for word processing. In the help file format, LabVIEW saves the graphics to external bitmap files. In the word processing file format, LabVIEW embeds the graphics in the document. For HTML files, LabVIEW saves all graphics externally in the JPEG, PNG, or GIF formats.

Selecting Graphic Formats for HTML Files

When you save documentation to an HTML file, you can select the format of the graphics files and the color depth.

The JPEG format compresses graphics well but can lose some graphic detail. This format works best for photos. For line art, front panels, and block diagrams, JPEG compression can result in fuzzy graphics and uneven colors. JPEG graphics are always 24-bit graphics. If you select a lower color depth such as black-and-white, graphics save with the depth you requested, but the result is still a 24-bit graphic.

The PNG format also compresses graphics well, although not always as well as the JPEG format. However, PNG compression does not lose any detail. Also, it supports 1-bit, 4-bit, 8-bit, and 24-bit graphics. For lower bit depth, the resulting graphic compresses much better than JPEG.

LabVIEW does not save graphics as compressed GIF files. Use a graphics format converter to convert the uncompressed GIF files that LabVIEW saves to compressed GIF files. For higher quality compressed GIF files, select the PNG format when you save the documentation and use a graphics format converter to convert the PNG files that LabVIEW saves to GIF files. Starting with the PNG format produces higher quality graphics because the PNG format is an exact reproduction of the original graphic. Modify the HTML file that LabVIEW generated to refer to the GIF files with the .png extension.

Naming Conventions for Graphic Files

When you generate HTML or RTF documentation with external graphics, LabVIEW saves the control and indicator data type terminals to graphic files with consistent names. If a VI has multiple terminals of the same type, LabVIEW creates only one graphic file for that type. For example, if a VI has three 32-bit signed integer inputs, LabVIEW creates a single `ci32.x` file, where `x` is the extension corresponding to the graphic format.

Creating Your Own Help Files

You can use the HTML or RTF files LabVIEW generates to [create your own compiled help files](#). (Windows) You can compile the individual [HTML files](#) LabVIEW generates into an

HTML Help file. (macOS) You can use the individual HTML files LabVIEW generates in Apple Help.

You can compile the *RTF files* LabVIEW generates into a (Windows) WinHelp or (Linux) HyperHelp file.

Providing Custom Documentation for VIs and Applications

To provide custom documentation from the **Help** menu, place a custom help file in the labview\help directory. (Windows) You can use any file type except text files for custom documentation. (macOS and Linux) You can use only HTML .html and .htm and PDF files.



Note (Linux) To use a PDF file for custom documentation, you must have one of the following PDF viewers installed:

- Acroread
- Xpdf
- KPDF
- GGV

Complete the following steps to display custom help in the **Help** menu:

1. Place the custom help file in the labview\help directory.
2. Place a text file that has the same name as the custom help file in the labview\help directory. Within the text file, include the help title you want to display in the **Help** menu. For example, if you name the help file My Custom Help.html, you must name the text file that contains the menu title My Custom Help.txt.



Note The only information you must provide in the text file is the custom help title you want to display in the **Help** menu.

Linking to Web-Based Help from a VI

To link a VI to web-based documentation from the Context Help window, select the **Web-based Help** option from the drop-down menu on the Documentation page of the VI Properties dialog box. Use the **Help URL** text box to specify the URL of the topic to which you want to link.

You also can use the Use Web URL and the Document Web URL properties to link a VI to web-based documentation programmatically.

Linking to Local Help from a VI

To link a VI to local documentation from the Context Help window, select the **Local Help File** option from the drop-down menu on the Documentation page of the VI Properties dialog box. Use the **Help path** text box to browse to custom documentation on the local computer. Optionally, use the **Help tag** text box to [link the VI to a specific topic](#) within a compiled help file or to link to a bookmark within an HTML help file. Otherwise, leave the **Help tag** text box empty. For example, if you specify to use a compiled help file named `My Custom Help.chm` in the **Help path** text box, you can enter `GettingStarted.html#Intro` in the **Help tag** text box so the **Detailed help** link in the **Context Help** window links directly to introductory information in the compiled help you provide. However, if you specify to use a PDF file named `My Custom Help.pdf` in the **Help path** text box, leave **Help tag** empty because you can link a VI to a specific topic or a bookmark within a topic only if the topic is an HTML file or an HTML file within compiled help.

You also can use the Document Tag and the Document Path properties to programmatically link a VI to a local help file.

Printing VIs Programmatically

Use any of the following ways to programmatically print VIs or data the VIs return:

- Set a VI to automatically print its front panel every time it finishes running.
- Create a subVI to print the VI.
- Use the Report Generation VIs to print reports or to save HTML reports that contain VI documentation or data the VI returns.
- Use the [VI Server](#) to programmatically print a VI window or to print VI documentation or save it to HTML, RTF, or text files at any time.



Note If you print VI documentation from a stand-alone application, you can print only the front panels.

Printing the Front Panel of a VI after the VI Runs

Select **Operate»Print at Completion** to print the front panel of a VI when it finishes running. You also can select **File»VI Properties**, select **Print Options** from the **Category** pull-down menu, and place a checkmark in the **Automatically Print Panel Every Time VI Completes Execution** checkbox.

Selecting these options is similar to selecting **File»Print Window** when the front panel is the active window.

If you use the VI as a subVI, LabVIEW prints when that subVI finishes running and before the subVI returns to the caller.

Using a SubVI to Print Data from a Higher Level VI

In some cases, you might not want a VI to print every time it finishes running. You might want printing to occur only if the user clicks a button or if some condition occurs, such as a test failure. You also might want more control over the format for the printout, or you might want to print only a subset of the controls. In these cases, you can use a subVI that is set to print at completion.

Create a subVI and format the front panel the way you want LabVIEW to print it. Instead of selecting **Operate»Print at Completion** in the higher level VI, select it in the subVI. When you want to print, call the subVI and wire the data you want to print to the subVI.

Generating and Printing Reports

Use the Report Generation VIs to print reports or to save HTML reports that contain VI documentation or data the VI returns. Use the Print VI Panel or Documentation VI to generate a basic report that contains VI documentation. Use the Create Easy Text Report VI to generate a basic report that contains data the VI returns. Use the other Report Generation VIs to generate more complex reports.

Use the Report Express VI to generate a preformatted report that contains VI documentation, data the VI returns, and report properties, such as the author, company, and number of pages.

Use the Report Generation VIs to perform the following tasks:

- Append text, graphics, tables, or VI documentation to a report.
- Set text font, size, style, and color.
- Set the report orientation—portrait or landscape.
- Set the report headers and footers.
- Set margins and tabs.

Additional Printing Techniques

If standard LabVIEW printing methods do not meet your needs, you can use the following additional techniques:

- Export data to other applications, such as Microsoft Excel, save the data to a file, and print from the other application.
- Use the System Exec VI.
- (Windows) Use [ActiveX](#) to make another application print data.

Programmatically Controlling VIs

The VI Server allows you to programmatically control VIs and LabVIEW. You can access the VI Server through block diagrams, ActiveX technology, and the TCP protocol. You can perform VI Server operations on a local computer or remotely across a network.

Refer to the `labview\examples` directory for examples of programmatically controlling VIs.

Find related examples



RELATED INFORMATION

[Capabilities of the VI Server](#) on page 259

[Building VI Server Applications](#) on page 260

[Manipulating Application and VI Settings](#) on page 261

[Dynamically Loading and Calling VIs](#) on page 245

[Asynchronously Calling VIs](#) on page 246

[Asynchronously Calling VIs](#) on page 981

[Configuring the VI Server](#) on page 989

[Configuring VI Calls](#) on page 989

[Controlling Front Panel Objects Programmatically from a SubVI](#) on page 990

[Creating a VI Server Application](#) on page 992

Dynamically Loading and Calling VIs

You can dynamically load VIs instead of using statically linked subVI calls. A statically linked subVI is one you place directly on the block diagram of a caller VI. It loads at the same time the caller VI loads.

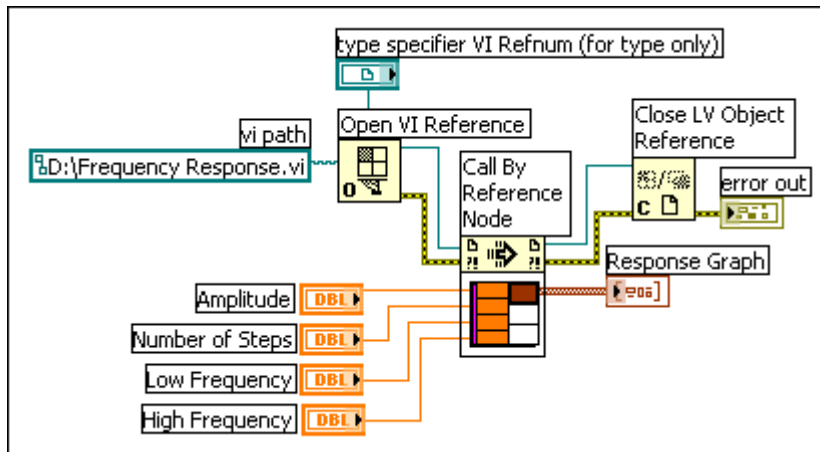
Unlike statically linked subVIs, dynamically loaded VIs do not load until the caller VI loads them with the Open VI Reference. If you have a large caller VI, you can save load time and memory by dynamically loading the VI because the VI does not load until the caller VI needs it, and you can release it from memory after the operation completes. You also can use the VI Call Configuration dialog box to [configure when to load the subVI](#).

Call By Reference Nodes and Strictly Typed VI Refnums

Use the Call By Reference node or the Start Asynchronous Call node to dynamically call VIs. Both nodes begin a call to the subVI specified by their **reference** input. However, the Start Asynchronous Call node [calls the subVI asynchronously](#) while data flow continues in the calling VI.

The Call By Reference node requires a strictly typed VI refnum. The strictly typed VI reference identifies both the VI you are calling and its connector pane. It does not create a permanent association to the VI or contain other VI information, such as the name and location. You can wire the Call By Reference node inputs and outputs just like you wire any other VI.

The following block diagram shows how to use the Call By Reference node to dynamically call the Frequency Response VI. The Call By Reference node requires the use of the Open VI Reference and Close Reference functions, similar to the functions you use for the Property Node and the Invoke Node.



The VI you specify for strictly typed refnums provides only the connector pane information. That is, no permanent association is made between the refnum and the VI. In particular, avoid confusing selecting the VI connector pane with getting a refnum to the selected VI. You specify a particular VI using the **vi path** input on the Open VI Reference function.

Asynchronously Calling VIs

When you make a standard call to a target VI with either a subVI node or the Call By Reference node, data flow stops at that node until the target VI returns its results, at which point data flow continues from the output terminals of the node.

In contrast, an asynchronous call to a subVI starts execution of the subVI while allowing data flow to continue in the calling VI. You might want to call subVIs asynchronously when the calling VI does not require the results of the subVI immediately. In those cases, the asynchronous call can drastically improve the execution time of the calling VI by allowing the subVI to run in parallel with the calling VI. Refer to the following list for examples of programming scenarios that benefit from asynchronous VI calls:

- Delegating work to subVIs while continuing to run
- Dividing work for a single task among multiple instances of the same target VI, especially when you cannot know until run time how many instances the task requires
- Launching complex user interfaces whose components are controlled by separate VIs that must run in parallel and collect results simultaneously

labview\examples\Application Control\VI Server\Asynchronous Call By Reference

Overview of Constructing an Asynchronous VI Call

In LabVIEW, all asynchronous VI calls consist of a combination of the following steps.

1. **Prepare a VI reference to be called asynchronously** —You prepare a VI reference for asynchronous execution when you obtain the reference with the Open VI Reference function and include one of the asynchronous option flags 0x80 or 0x100 as part of the options input. This step allows LabVIEW to allocate resources for the asynchronous call.
2. **Start the asynchronous VI call** —Wire the VI reference to the Start Asynchronous Call node to start the execution of the target VI. While the target VI runs, data flow continues in the calling VI.
3. **(Optional) Collect the results of the asynchronous VI call** —If your application requires the outputs of a target VI that it calls asynchronously, wire the VI reference to the Wait On Asynchronous Call node to collect the results. The Wait On Asynchronous Call node waits until the associated subVI finishes executing or until the Wait On Asynchronous Call node times out.
4. **Close the VI reference** —Use the Close Reference function to inform LabVIEW that you are finished using the VI reference. LabVIEW then can free the associated resources.

Types of Asynchronous VI Calls

To make an asynchronous VI call, you must determine which of the following usage models applies to your use case. Refer to the following topics for more information about implementing each model:

- **Call-and-forget** —A calling VI starts the execution of a subVI but does not require outputs from the subVI.
- **Call-and-collect** —A calling VI starts the execution of a subVI and needs to collect the results at a later time.

asynchronously start multiple calls to the same VI for parallel execution

How LabVIEW Manages Asynchronous VI Calls: The Asynchronous Call Pool

While most use cases for asynchronously calling VIs do not require knowledge of how LabVIEW manages asynchronous VI calls, understanding the asynchronous call pool can help you control performance and design deterministic applications. When you prepare a VI reference to run asynchronously by including one of the asynchronous option flags as part of the **options** input of the Open VI Reference function, LabVIEW creates a pool of data spaces that track data for individual calls to that VI. Because LabVIEW associates a pool of data spaces rather than a single data space to a VI reference, you can start multiple asynchronous calls with the same VI reference. Every time you use the Start Asynchronous Call node to start an asynchronous call with the VI reference, LabVIEW reserves one of the

data spaces in the pool. When the VI finishes running, LabVIEW stores its outputs in the data space and marks the VI call as complete. If a Wait On Asynchronous Call node is waiting on the outputs of the called VI, LabVIEW returns the results from the data space, thereby freeing the data space for another call to the same VI.

Performance Impact of the Asynchronous Call Pool

For each VI reference, LabVIEW creates one data space in the asynchronous call pool for each CPU core on the target computer. If you start more simultaneous calls to the target VI than there are available data spaces, LabVIEW increases the size of the call pool by allocating more data spaces. Because the process of allocating data spaces does not take a deterministic amount of time, the calling VI might experience an unpredictable delay at any Start Asynchronous Call node that triggers an allocation.

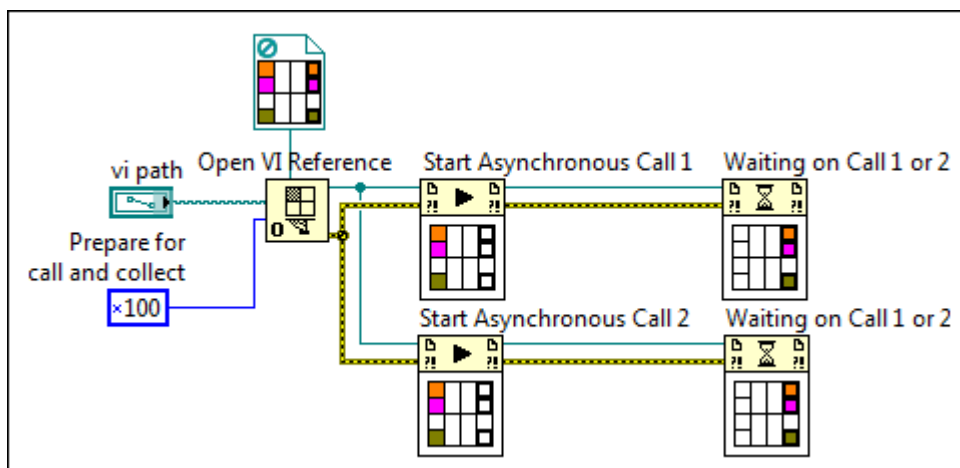
Avoiding Unpredictable Delays at the Start Asynchronous Call Node

To avoid unpredictable delays in VIs that must run on a real-time operating system, preallocate the number of data spaces in the call pool for a given VI reference by calling the Populate Asynchronous Call Pool method before starting any asynchronous calls. Preallocating the data spaces limits delays to the beginning of the program instead of allowing delays to occur at the Start Asynchronous Call node.

Refer to the Populating the Asynchronous Call Pool VI in the `labview\examples\Application Control\VI Server\Asynchronous Call By Reference` directory for more information about how the asynchronous call pool affects performance. Find related examples

Impact of Asynchronous Call Pool on Collecting Outputs from Multiple Calls to the Same VI

When you want to collect the results of an asynchronous VI call, you must pass the reference to the called VI to the Wait On Asynchronous Call node. Because the Wait On Asynchronous Call node receives a reference to a VI rather than a reference to a specific data space, the node cannot collect the outputs of a specific call to the referenced VI. Instead, the node returns the outputs of any finished call to that VI. Refer to the following block diagram for a further explanation of this behavior.



As the labels suggest, when multiple Wait On Asynchronous Call nodes rely on the same VI reference, you cannot predict which node will return the outputs of a particular call to that VI reference. In the previous diagram, the top Wait On Asynchronous Call node is capable of returning the outputs of **Start Asynchronous Call 2** even though it is wired to **Start Asynchronous Call 1**. If you want to make multiple calls to the same VI but need to keep the Start Asynchronous Call node paired with a specific Wait On Asynchronous Call node, use either of the following solutions:

- Create a unique VI reference to the target VI for every asynchronous call that you want to make to it. If a Wait On Asynchronous Call node shares a unique VI reference with a Start Asynchronous Call node, the two nodes are guaranteed to refer to the same instance of the VI. This solution is similar to [collecting outputs from multiple calls to unique VI references](#).
- Modify the target VI to accept a unique number or string as an input and return the same identifier as an output. When you call the target VI asynchronously, compare the identifier that you pass to a particular Start Asynchronous Call node to the identifier that the Wait On Asynchronous Call node returns. If they match, you know which call the Wait On Asynchronous Call node returned.

Callback VIs

A callback VI contains the code you write to handle an ActiveX or .NET event you specify. You must [create a callback VI](#) to handle events from ActiveX controls or .NET objects when the controls or objects generate the registered events. The callback VI runs when the event occurs. When you create a callback VI, LabVIEW creates a [reentrant](#) VI that you can open and edit to handle an event. A callback VI contains the following elements:

- **Event Common Data** contains the following elements:
 - **Event Source** is a numeric control that specifies the source of the event, such as LabVIEW, ActiveX, or .NET. A value of 1 indicates an ActiveX event. A value of 2 indicates a .NET event.
 - **Event Type** specifies which event occurred. This is an enumerated type for user interface events and a 32-bit unsigned integer type for ActiveX, .NET, and other event sources. For ActiveX events, the event type represents the method code, or ID, for the event registered. You can ignore this element for .NET.
 - **Time Stamp** is the time stamp in milliseconds that specifies when the event was generated.
- **Control Ref** is a reference to the ActiveX or automation refnum or .NET object on which the event occurred.
- **Event Data** is a cluster of the parameters specific to the event the callback VI handles. LabVIEW determines the appropriate **Event Data** when you select an event from the Register Event Callback function. If an event does not have any data associated with it, LabVIEW does not create this control in the callback VI.
- **Event Data Out** is a cluster of the modifiable parameters specific to the event the callback VI handles. This element is available only if the ActiveX or .NET event has output parameters.

- (Optional) **user parameter** is data that you want to pass to the callback VI when the ActiveX or .NET object generates the event.

**NOTE**

You can use an existing VI as a callback VI as long as the connector pane of the VI you intend to use matches the connector pane of the event data. The callback VI must be reentrant, and the reference to the callback VI must be strictly typed.

To allow callback VIs to execute without interruption, LabVIEW delays the processing of operating system messages until any callback VIs stop execution or until you load a modal window. When LabVIEW delays the processing of operating system messages, you cannot interact with any LabVIEW front panels. A modal window is a type of window that remains active or remains on top of all other LabVIEW windows until you close the window or open another modal window. You cannot interact with other windows while a modal window is open. Most dialog boxes in LabVIEW are modal windows.

You cannot open a non-modal window from a LabVIEW callback VI nor a DLL while any other process is running. Refer to [Calling Non-Modal Windows Programmatically](#) for more information about calling a non-modal window from a callback VI or DLL.



Note ActiveX objects call ActiveX callback VIs synchronously. This means that the object may not be able to run other code during the execution of the callback and may cause deadlocks. You should carefully consider the interactions between the callback diagram and other code. For example, calling a non-reentrant subroutine inside an ActiveX callback VI could hang if you call the callback while the subroutine is already executing.

Emailing Data from VIs

Use the SMTP Email VIs to send email, including attached data and files, using the Simple Mail Transfer Protocol (SMTP). You cannot use the SMTP Email VIs to receive information. The SMTP Email VIs use the Multipurpose Internet Mail Extensions (MIME) format to encode the message. In this format, you can send multiple files, including binary data files, within an email.

In addition to the recipient(s) email address, you must know the Web address of the SMTP server. If you use the Send Email Express VI, you specify a mail server by entering an SMTP server address in the **Outgoing Mail Server (SMTP)** textbox. The server address must be the host name or IP address of an external server computer that can service requests from the computer running the SMTP Email VIs. If you do not know which mail server to use, contact your network administrator for the name of a valid server. After you specify a valid mail server, the SMTP Email VIs open a connection to the server and send the server commands that describe the recipients and the contents of the email. The server sends the message to the individual recipients or forwards it to other SMTP servers.

Alternatively, you also can specify a mail server by wiring a **server address** to the Open Handle VI. The Open Handle VI creates an SMTP client handle that you wire to other SMTP Email VIs to configure the email specifications. Use the Close Handle VI to dispose of the handle and to free system resources. Refer to the `labview\examples\Data Communication\Protocols\SMTP\Send Email using SMTP Client.vi` for an example of using the SMTP Email VIs.

Character Sets and UTF-8 Encoding

LabVIEW supports 8-bit Unicode Transmission Format (UTF-8) character encoding in email messages. UTF-8 is a lossless, variable-length character encoding scheme that uses groups of bytes to represent characters from many languages worldwide. If the system character set is not already encoded in UTF-8 format, the SMTP Email VIs convert input strings encoded in the system character set to UTF-8 format before sending the email. This conversion allows you to send email messages that a wide variety of operating systems can read.

Anatomy of a VI and Common VI Objects

In order to create or modify a VI using [VI Scripting](#), you must understand how the objects that compose a VI relate both to the VI Server class hierarchy and to nearby objects. This topic presents the following three essential types of information about several common VI objects in order to facilitate a variety of use cases.

- **The Anatomy of VIs and Common VI Objects** The images in this topic illustrate different types of objects that you can manipulate with VI Scripting as well as some of their parts and interrelationships. For example, a For Loop is a common VI object that contains many parts. If you want to create a new wire that enters a For Loop, you must interact with several of these parts: the loop itself, a loop tunnel, and the outer and inner terminals of the loop tunnel. By exploring the anatomies of a For Loop and other common VI objects, you can identify all the object elements that a specific scripting action might touch.
- **The VI Server Class of Common VI Objects** Several scripting functions require you to specify the class of the object with which you want to interact. The Class Hierarchy column of each anatomy table in this topic provides the class information for the objects within the associated diagram. Use a class specifier constant to pass this information to a function or VI. Use the information in the Class Hierarchy column to specify a class in the following tasks:
 - Creating a new object with the New VI Object function
 - Obtaining a reference to a pre-existing object with the Open VI Object Reference function
 - Searching for instances of a specific class of object with the Traverse for GObjects VI
 - Downcasting a reference with the To More Specific Class function
 - Identifying what properties and methods in the VI Server class hierarchy are available for a specific object
- **Properties that Return a Reference to Specific Instances of Common VI Objects** Some scripting activities require you to use a reference to one object to obtain references to other objects. In these cases, you must know which properties of the original object return the desired references. The Returns the Object Reference column of each anatomy table provides this property information using the notation of Object Type:Property. For example, the ForLoop:Loop Count notation indicates that you are calling the Loop Count property on a reference to a ForLoop object. To do this, you must first obtain a reference to a For Loop. Then you can wire the ForLoop reference to a Property Node and select the Loop Count property for

that Property Node. Use the information in the Returns the Object Reference columns to obtain the references required in the following tasks:

- Navigating a VI
- Editing VIs
- Creating connections among new VI objects

Block Diagram Anatomy

The following image illustrates several common block diagram objects you can create or modify with VI Scripting.

1 Control/Indicator Terminal	4 Wire	7 SubVI
2 Numeric Constant	5 Increment Function	8 Top Level Diagram
3 Local Variable	6 Compound Arithmetic Function	9 Block Diagram Comment

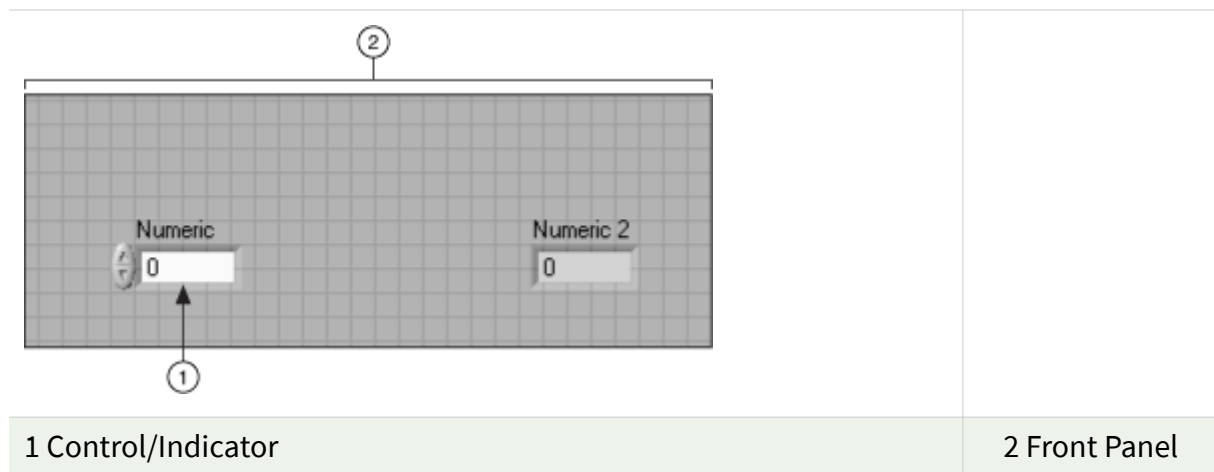
The following table lists the classes and properties you can use to obtain references to the objects in the previous image.

Class Hierarchy	Returns the Object Reference	
1	GObject»Terminal»ControlTerminal	Control:Terminal, Diagram:All Objects[]
2	GObject»Constant»NumericConstant	Diagram:All Objects[]
3	GObject»Node»Local	Control:Local Variables[]
4	GObject»Wire	Terminal:Connected Wire, Diagram:Wires[]

5	GObject»Node»Function	Diagram:Nodes[], Diagram:All Objects[]
6	GObject»Node»GrowableFunction»CompoundArithmetic	Diagram:Nodes[], Diagram:All Objects[]
7	GObject»Node»SubVI	Diagram:SubVIs[], Diagram:Nodes[], Diagram:All Objects[]
8	GObject»AbstractDiagram»Diagram»TopLevelDiagram	VI:Block Diagram
9	GObject»Decoration»Text	Diagram:Decorations[], Diagram:All Objects[]

Front Panel Anatomy

The following image illustrates some common front panel objects.

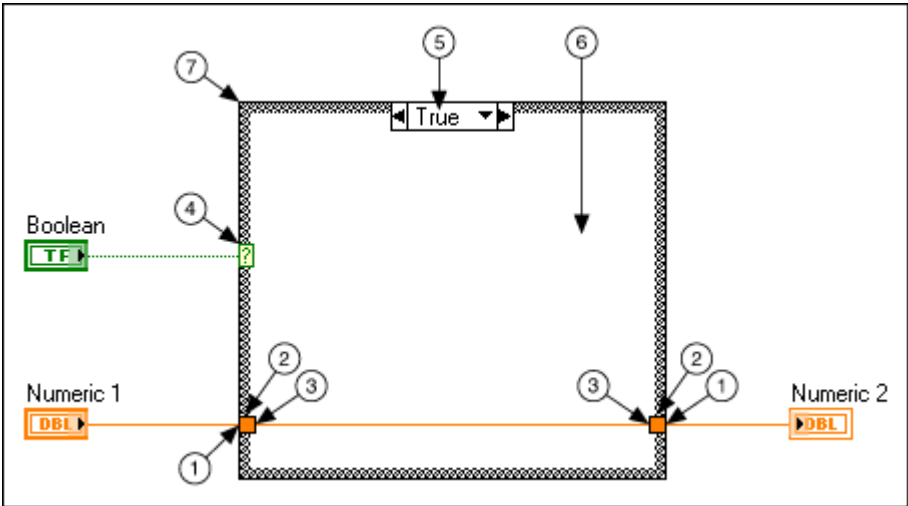


The following table lists the classes and properties you can use to obtain references to the objects in the previous image.

Class Hierarchy	Returns the Object Reference	
1	GObject»Control»Numeric	Panel:Controls[], ControlTerminal:Control
2	GObject»Panel	VI:Front Panel

Case Structure Anatomy

The following image illustrates the various objects that compose a Case structure.



1 Terminal

4
Selector
Tunnel

7 Case
Structure

2 Tunnel

5
Selector
Label

3 Inner Terminal

6
Diagram

The following table lists the classes and properties you can use to obtain references to the objects in the previous image.

Class Hierarchy	Returns the Object Reference	
1	GObject»Terminal»OuterTerminal	Tunnel:Outside Terminal
2	GObject»Tunnel	Structure:Tunnels[]
3	GObject»Terminal»InnerTerminal	Tunnel:Inside Terminals[]
4	GObject»Tunnel»ConditionalTunnel»SelectorTunnel	Structure:Tunnels[], CaseStructure:Selector
5	N/A	CaseStructure:Frame Names
6	GObject»AbstractDiagram»Diagram	Structure:Diagrams[], MultiFrameStructure:Frames[]
7	GObject»Node»Structure»MultiFrameStructure»CaseStructure	Diagram:All Objects[], Diagram:Nodes[]

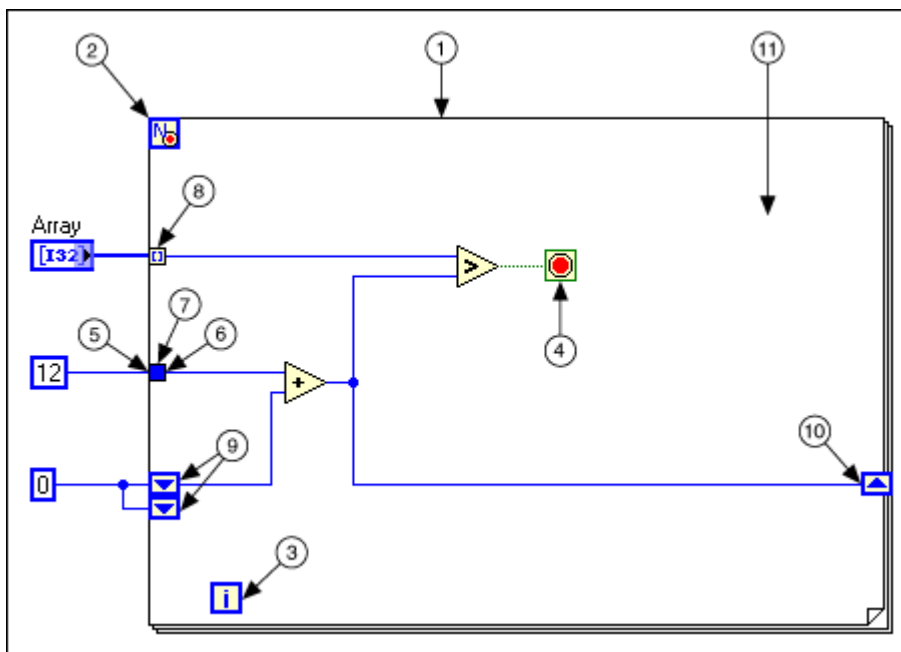
* LabVIEW does not provide a reference to the selector label in a Case structure. The CaseStructure:Frame Names property returns an array of strings rather than an array of references to the diagrams associated with different cases. You can use these name strings

with the CaseStructure:Get Frame Index method to obtain indexes to specific diagrams within the array returned by the MultiFrameStructure:Frames[] property.

The Structures - Case Structure VI demonstrates how to use VI Scripting to access parts of the Case structure anatomy.

For Loop Anatomy

The following image illustrates the various objects that compose a For Loop.



1 For Loop	5 Outer Terminal	9 Left Shift Register
2 Count Terminal (N)	6 Inner Terminal	10 Right Shift Register
3 Terminal (i)	7 Loop Tunnel	11 Diagram
4 Conditional Terminal	8 Auto-indexing Loop Tunnel	

The following table lists the classes and properties you can use to obtain references to the objects in the previous image.

Class Hierarchy	Returns the Object Reference	

1	GObject»Node»Structure»Loop»ForLoop	Diagram:All Objects[], Diagram:Nodes[]
2	GObject»Tunnel	ForLoop:Loop Count
3	GObject»Terminal	Loop:Loop Counter
4	GObject»Terminal	ForLoop:Loop End Ref
5	GObject»Terminal»OuterTerminal	Tunnel:Outside Terminal
6	GObject»Terminal»InnerTerminal	Tunnel:Inside Terminals[]
7	GObject»Tunnel»LoopTunnel	Structure:Tunnels[]
8	GObject»Tunnel»LoopTunnel	Structure:Tunnels[], LoopTunnel:Index Mode = 1 (for Auto-indexing)
9	GObject»Tunnel»LeftShiftRegister	RightShiftRegister:Left Registers[]
10	GObject»Tunnel»RightShiftRegister	Loop:Shift Registers[], LeftShiftRegister:Right Shift Register
11	GObject»AbstractDiagram»Diagram	Loop:Diagram, Structure:Diagrams[]

The Structures - For Loop VI demonstrates how to use VI Scripting to access parts of the For Loop anatomy.

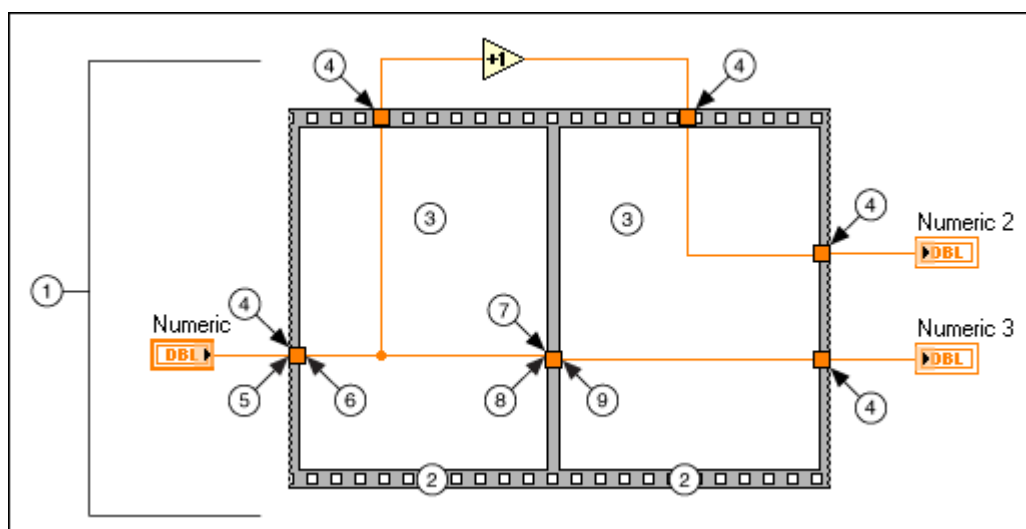
Flat Sequence Structure Anatomy



NOTE

In Flat Sequence structures, you can wire out of the structure from one frame and back into the structure through another frame. This type of data flow distinguishes the Flat Sequence structure from other structures, so that all the VI Server classes are different for the Flat Sequence structure.

The following image illustrates the various objects that compose a Flat Sequence structure.



1 Flat Sequence	4 Flat Sequence Outer Tunnel	7 Flat Sequence Inner Tunnel
2 Flat Sequence Frame	5 Flat Sequence Outer Terminal (of Outer Tunnel)	8 Flat Sequence Left Terminal (of Inner Tunnel)
3 Diagram	6 Flat Sequence Inner Terminal (of Outer Tunnel)	9 Flat Sequence Right Terminal (of Inner Tunnel)

The following table lists the classes and properties you can use to obtain references to the objects in the previous image.

Class Hierarchy	Returns the Object Reference	
1	GObject»FlatSequence	Diagram:All Objects[]
2	Generic»FlatSequenceFrame	FlatSequence:Frames[], Diagram:Firing Order[], FlatSequenceOuterTunnel:Frame, FlatSequenceInnerTunnel:Left Frame, FlatSequenceInnerTunnel:Right Frame
3	GObject»AbstractDiagram»Diagram	FlatSequenceFrame:Diagram
4	GObject»FlatSequenceOuterTunnel	FlatSequenceFrame:Outer Tunnels[]

5	GObject»Terminal	FlatSequence:Terminals[], FlatSequenceOuterTunnel:Outer Terminal
6	GObject»Terminal	FlatSequenceOuterTunnel:Inner Terminal
7	GObject»FlatSequenceInnerTunnel	FlatSequenceFrame:Left Side Inner Tunnels[], FlatSequenceFrame:Right Side Inner Tunnels[]
8	GObject»Terminal	FlatSequenceInnerTunnel:Left Terminal
9	GObject»Terminal	FlatSequenceInnerTunnel:Right Terminal

The Structures - Flat Sequence Structure VI demonstrates how to use VI Scripting to access parts of the Flat Sequence Structure anatomy.

For Additional Information

This topic highlights some of the most common VI objects. For information about objects not addressed here, consult the following resources:

- Generic:Class Name - Use this property to discover the class name of any VI object. Then find the class name within the VI Server class hierarchy to identify the entire class hierarchy for the object.
- VI Server Class Hierarchy - If you know the class of a particular object, the VI Server class hierarchy can tell you properties or methods that provide references to other parts of that object.

VI Scripting

Requires: VI Scripting

Learn how to complete many common VI Scripting tasks.

- Instructions for enabling VI Scripting
- Instructions and examples for inspecting, editing, and creating target VIs

- A basic tutorial for using VI Scripting to create a blank VI, add objects to the block diagram, reposition those objects, and wire the objects together



RELATED INFORMATION

[Enabling VI Scripting](#) on page 995

[Common VI Scripting Tasks](#) on page 996

[Enabling VI Scripting](#) on page 995

[Common VI Scripting Tasks](#) on page 996

[VI Scripting Tutorial](#) on page 1026

Capabilities of the VI Server

The VI Server is a set of functions that allows you to dynamically control front panel objects, VIs, and the LabVIEW environment. These functions are located on the Application Control palette. Use the VI Server to perform the following programmatic operations:

- [Dynamically load VIs](#) into memory when another VI needs to call them, rather than loading all subVIs when you open a VI.
- Call a VI remotely.
- Configure a LabVIEW [application instance](#) to be a server that exports VIs you can call from other instances of LabVIEW on the Web. For example, if you have a data acquisition application that acquires and logs data at a remote site, you can sample that data occasionally from your local computer. By changing the VI Server preferences in the Options dialog box, you can make some VIs accessible on the Web so that transferring the latest data is as easy as a dynamic VI call. The VI Server handles the networking details. The VI Server also works across platforms, so the client and the server can run on different platforms.
- Edit the properties of a VI and LabVIEW. For example, you can dynamically determine the location of a VI window or scroll a front panel so that a part of it is visible. You also can programmatically save any changes to disk.
- Update the properties of multiple VIs rather than manually using the VI Properties dialog box for each VI.
- Retrieve information about an application instance, such as the version number and edition. You also can retrieve environment information, such as the platform on which LabVIEW is running.
- Create a plug-in architecture for the application to add functionality to the application after you distribute it to customers. For example, you might have a set of data filtering VIs, all of which take the same parameters. By designing the application to dynamically load these VIs from a plug-in directory, you can ship the application with a partial set of these VIs and make more filtering options available to users by placing the new filtering VIs in the plug-in directory.
- [Control front panel objects.](#)

Building VI Server Applications

The programming model for VI Server applications is based on [refnums](#). Refnums also are used in file I/O, network connections, and other objects in LabVIEW.

Typically, you open a refnum to an [application instance](#) or to a VI. You then use the refnum as a parameter to other VIs. The VIs get (read) or set (write) properties, execute methods, or dynamically load and call a referenced VI. Finally, you close the refnum, which releases the referenced VI from memory.

Use the following Application Control functions and nodes to build VI Server applications:

- **Open Application Reference** —Opens a reference to a local or remote application instance.
- **Open VI Reference** —Opens a reference to a VI on the local or remote computer or dynamically loads a VI from disk.
- **Property Node** —Gets and sets VI, object, or application [properties](#).
- **Invoke Node** —Invokes [methods](#) on a VI, object, or application.
- **Call By Reference Node** —Calls a dynamically loaded VI.
- **Start Asynchronous Call Node** —Starts an asynchronous call to a dynamically loaded VI.
- **Wait On Asynchronous Call Node** —Collects the results of an asynchronously called VI.
- **Close Reference** —Closes open references to the VI, object, or application you accessed using the VI Server.

Application and VI References

You access VI Server functionality through references to two main classes of objects—the Application object and the VI object. After you create a reference to one of these objects, you can pass the reference to a VI or function that performs an operation on the object.

An Application reference refers to a local or remote application instance. You can use Application properties and methods to change LabVIEW preferences and return system information. A VI refnum refers to a VI in an application instance.

With a reference to an application instance, you can retrieve information about the LabVIEW environment, such as the platform on which LabVIEW is running, the version number, or a list of all VIs currently in memory. You also can set information, such as the list of VIs exported to other application instances. Because you can open multiple application instances at once, you must use an application reference when you are using VI server properties and methods in one application instance, and you want to interact with a different application instance. For example, LabVIEW opens a new application instance when you create a LabVIEW project or a target for a LabVIEW project.

When you create a refnum to a VI, LabVIEW loads the VI into memory. The VI stays in memory until you close the reference and until the VI meets the following conditions:

- There are no other open references to the referenced VI.
- The front panel of the VI is not open.
- The VI is not a subVI of another VI in memory.
- The VI is not a member of an open project library.

With a refnum to a VI, you can update all the properties of the VI available in the VI Properties dialog box as well as dynamic properties, such as the position of the owning pane. You also can programmatically [print the VI documentation](#), save the VI to another location, and [export and import its strings](#) to translate into another language.

Manipulating Application and VI Settings

Properties are attributes of a VI, object, or application. Methods perform actions on a VI, object, or application. You can get or set properties and methods on local or remote [application instances](#), VIs, and objects by using the Property Node and the Invoke Node. You can get and set many application, VI, and object settings only through properties and methods.

Refer to the `labview\examples\viserver` directory for examples of using the Application and VI Class properties and methods.

Find related examples

Property Nodes

Use the Property Node to get and set various properties on an application or VI. Select properties from the node by using the Operating tool to click the property terminal or by right-clicking the white area of the node and selecting **Properties** from the shortcut menu.

The following are examples of how properties can enhance ease of use in an application or VI:

- Set the text focus to the main, most commonly used control.
- Disable or hide controls that are not currently relevant or valid.
- Guide the user through steps by highlighting controls.
- Change colors to bring attention to error conditions.

You can read or write multiple properties using a single node. However, some properties are not readable and some are not writable. Use the Positioning tool to resize the Property Node to add new terminals. A small direction arrow to the right of the property indicates a property you read. A small direction arrow to the left of the property indicates a property you write. Right-click the property and select **Change to Read** or **Change to Write** from the shortcut menu to change the operation of the property.

The node executes from top to bottom. The Property Node does not execute if an error occurs before it executes, so always check for the possibility of errors. If an error occurs in a

property, LabVIEW ignores the remaining properties and returns an error. If you right-click the Property Node and select **Ignore Errors inside Node**, LabVIEW executes the remaining properties on the Property Node. Only the first error is returned by the Property Node. The **error out** cluster contains information about which property caused the error.

If the Property Node opens and returns a reference to an object, use the Close Reference function to close the reference.

Implicitly Linked Property Nodes

When you create a property from a front panel object by right-clicking the object, selecting **Create»Property Node**, and selecting a property from the shortcut menu, LabVIEW creates a Property Node on the block diagram that is implicitly linked to the front panel object. Because such a Property Node is implicitly linked to the front panel object that you created it from, the node has no reference input and you do not need to wire the node to the terminal of the front panel object or the [control reference](#). To remove the link to the front panel object, right-click the Property Node on the block diagram and select **Disconnect From Control** from the shortcut menu.

Invoke Nodes

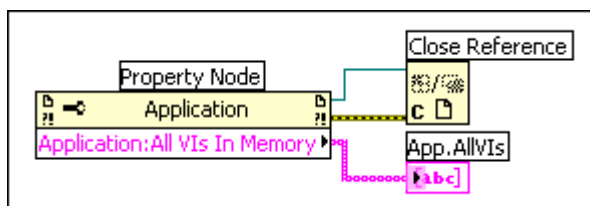
Use the Invoke Node to perform actions, or methods, on an application or VI. Unlike the Property Node, a single Invoke Node executes only a single method on an application or VI. Select a method by using the Operating tool to click the method terminal or by right-clicking the white area of the node and selecting **Methods** from the shortcut menu. You also can create an implicitly linked Invoke Node by right-clicking a front panel object, selecting **Create»Invoke Node**, and selecting a method from the shortcut menu.

The name of the method is always the first terminal in the list of parameters in the Invoke Node. If the method returns a value, the method terminal displays the return value. Otherwise, the method terminal has no value.

The Invoke Node lists the parameters from top to bottom with the name of the method at the top and the optional parameters, which are dimmed, at the bottom.

Manipulating Application Class Properties and Methods

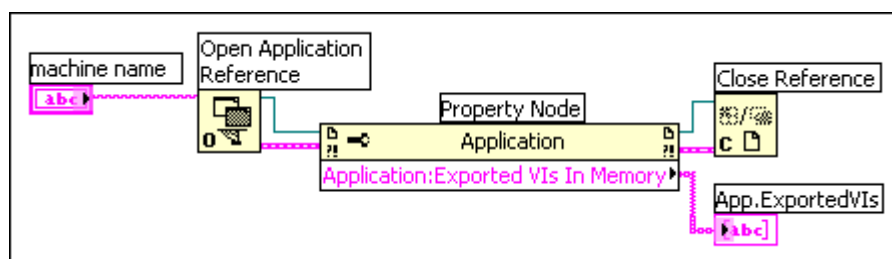
You can get or set properties on a local or remote application instance, perform methods on LabVIEW, or both. The following block diagram shows how to display all VIs in memory on a local computer in a string array on the front panel.



If you do not wire a refnum to the **reference** input, the Property Node or Invoke Node uses a reference to the current application instance. If you want to manipulate the properties or

methods of another application instance, you must wire an application reference to the **reference** input. If you have multiple application instances open simultaneously, be sure to wire an application reference to the **reference** input. For example, LabVIEW opens a new application instance each time you create a LabVIEW project or a target for a LabVIEW project. LabVIEW also creates a main application instance, which contains open VIs that are not part of a project and VIs that you did not open from a project.

To find the exported VIs in memory on a remote computer, wire a string control to the **machine name** input of the Open Application Reference function, as shown in the following block diagram, and enter the IP address or domain name. You also must select the Exported VIs In Memory property because the All VIs In Memory property used in the previous block diagram applies only to local application instances.



You also can use the Default:Application property to return the default application reference programmatically. Use the Application property to open the target application instance programmatically.

Manipulating VI Class Properties and Methods

You can get or set properties of a VI, perform methods on a VI, or both. Use the Open VI Reference function to return a VI reference, and then use Property Nodes and Invoke Nodes as you would when manipulating Application Class properties and methods.

If you do not wire a refnum to the **reference** input, the property or method uses a reference to the VI containing the Property Node or Invoke Node. If you want to manipulate the properties or methods of another VI, you must wire a VI refnum to the **reference** input.

The Property Node operates similarly to the Invoke Node. After you wire a VI refnum to the Property Node, you can access all the VI Class properties.

Manipulating Application and VI Class Properties and Methods

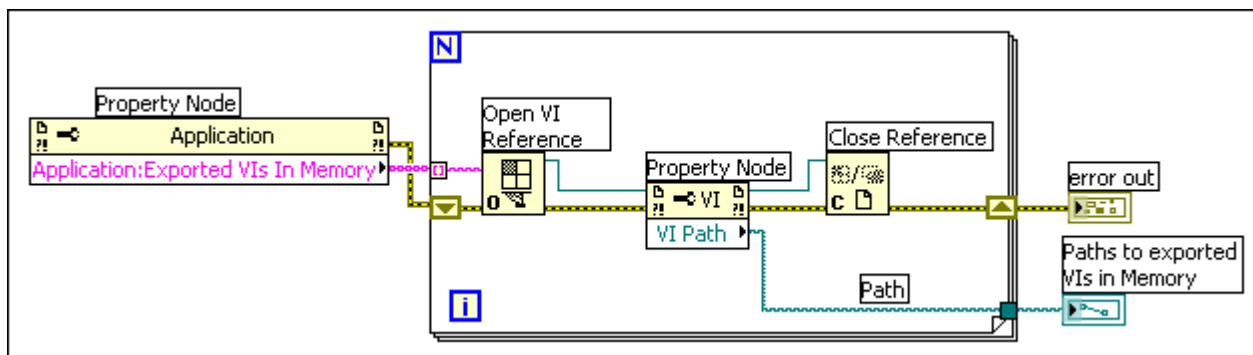
In some VIs, you must access both Application and VI Class properties or methods. You must open and close the Application and VI Class refnums separately, as shown in the following block diagram.



NOTE

The application reference on this block diagram is an implicit reference, and therefore does not need to be closed. Refer to the Closing References in LabVIEW tutorial for more information about when to close references in LabVIEW.

The following block diagram shows how to determine the exported VIs in memory on a local computer and to display the path to each of the VIs on the front panel. To find the exported VIs in memory, you must access an Application Class property. To determine the paths to each of these VIs, you must access a VI Class property. The number of exported VIs in memory determines the number of times the For Loop executes. Place the Open VI Reference and Close Reference functions inside the For Loop because you need a VI refnum for each VI in memory.



Editing and Running VIs on Remote Computers

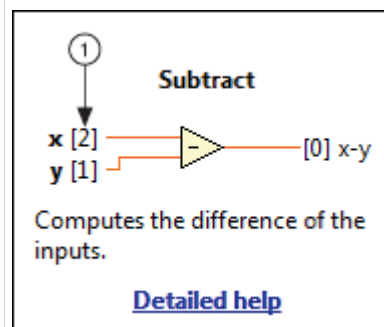
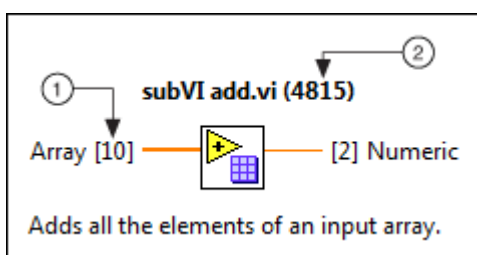
An important aspect of both Application and VI refnums is their network transparency. This means you can open refnums to objects on remote computers in the same way you open refnums to those objects on your computer.

After you open a refnum to a remote object, you can treat it in exactly the same way as a local object, with a few restrictions. For operations on a remote object, the VI Server sends the information about the operation across the network and sends the results back. The application looks almost identical regardless of whether the operation is remote or local.

VI Scripting Information in the Context Help Window

In order to complete VI Scripting tasks that involve the terminals of a node or a connector pane, you must use terminal and connector pane pattern indexes. You can configure the Context Help window to display these indexes for most nodes and VIs, thereby making this information readily available while you program in LabVIEW.

To display terminal and connector pane pattern indexes in the Context Help window, enable VI Scripting and place a checkmark in the **Display additional VI Scripting information in Context Help window** checkbox on the VI Server page of the Options dialog box. After you make this selection, the Context Help window looks similar to one of the following illustrations:



1 Terminal index—Identifies the position of the associated terminal within the array of terminal references returned by the `Node:Terminals[]` property of the displayed node. You can use the `Terminals[]` array and the terminal index with the `Index Array` function to obtain a reference to a specific terminal. You commonly must perform this task to navigate from a node to a connected wire or to wire data to a desired terminal using VI Scripting.

2 Pattern index—Identifies the connector pane pattern of the displayed VI. You can use the pattern index to configure the connector pane of a VI using VI Scripting. Only VIs display a pattern index in their Context Help window.

Refer to the Connector Pane Pattern Reference VI located in the `labview\examples\Application Control\VI Scripting\Connector Pane` directory for an illustration of the connector pane pattern associated with each pattern index.



RELATED INFORMATION

[Enabling VI Scripting](#) on page 995

[Navigating from a Node to a Wire Using VI Scripting](#) on page 1020

[Tutorial: Wiring Objects Using VI Scripting](#) on page 1031

[Configuring a Connector Pane Using VI Scripting](#) on page 1011

VI Execution Speed

Although LabVIEW compiles VIs and produces code that generally executes very quickly, you want to obtain the best performance possible when working on time-critical VIs. This section discusses factors that affect execution speed and suggests some programming techniques to help you obtain the best performance possible.

Examine the following items to determine the causes of slow performance:

- Input/Output (files, GPIB, data acquisition, networking)
- Screen Display (large controls, overlapping controls, too many displays)
- Memory Management (inefficient usage of arrays and strings, inefficient data structures)

- Compiler Optimizations (choices the compiler makes to balance editor responsiveness with VI execution speed)

Other factors, such as execution overhead and subVI call overhead, usually have minimal effects on execution speed.

Input/Output

Input/Output (I/O) calls generally incur a large amount of overhead. They often take much more time than a computational operation. For example, a simple serial port read operation might have an associated overhead of several milliseconds. This overhead occurs in any application that uses serial ports because an I/O call involves transferring information through several layers of an operating system.

The best way to address too much overhead is to minimize the number of I/O calls you make. Performance improves if you can structure the VI so that you transfer a large amount of data with each call, instead of making multiple I/O calls that transfer smaller amounts of data.

For example, if you are creating a data acquisition (NI-DAQmx) VI, you have two options for reading data. You can use a single-point data transfer function such as the AI Sample Channel VI, or you can use a multipoint data transfer function such as the AI Acquire Waveform VI. If you must acquire 100 points, use the AI Sample Channel VI in a loop with a Wait function to establish the timing. You also can use the AI Acquire Waveform VI with an input indicating you want 100 points.

You can produce much higher and more accurate data sampling rates by using the AI Acquire Waveform VI, because it uses hardware timers to manage the data sampling. In addition, overhead for the AI Acquire Waveform VI is roughly the same as the overhead for a single call to the AI Sample Channel VI, even though it is transferring much more data.

Screen Display

Frequently updating controls on a front panel can be one of the most time-consuming operations in an application. This is especially true if you use some of the more complicated displays, such as graphs and charts. Although most indicators do not redraw when they receive new data that is the same as the old data, graphs and charts always redraw. If redraw rate becomes a problem, the best solutions are to reduce the number of front panel objects and keep the front panel displays as simple as possible. In the case of graphs and charts, you can turn off autoscaling, scale markers, anti-aliased line drawing, and grids to speed up displays.

As with other kinds of I/O, there is a certain amount of fixed overhead in the display of a control. You can pass multiple points to an indicator at one time using certain controls, such as charts. You can minimize the number of chart updates you make by passing more data to the chart each time. You can see much higher data display rates if you collect your chart data into arrays to display multiple points at a time, instead of displaying each point as it comes in.

When you design subVIs whose front panels are closed during execution, do not be concerned about display overhead. If the front panel is closed, you do not have the drawing overhead for controls, so graphs are no more expensive than arrays.

In multithreaded systems, you can use the **Advanced»Synchronous Display** shortcut menu item to set whether to defer updates for controls and indicators. By default, controls and indicators use asynchronous displays, which means that after the execution system passes data to front panel controls and indicators, it can immediately continue execution. At some point thereafter, the user interface system notices that the control or indicator needs to be updated, and it redraws to show the new data. If the execution system attempts to update the control multiple times in rapid succession, you might not see some of the intervening updates.

In most applications, asynchronous displays significantly speed up execution without affecting what the user sees. For example, you can update a Boolean value hundreds of times in a second, which is more updates than the human eye can discern. Asynchronous displays permit the execution system to spend more time executing VIs, with updates automatically reduced to a slower rate by the user interface thread.

If you want synchronous displays, right-click the control or indicator and select **Advanced»Synchronous Display** from the shortcut menu to place a checkmark next to the menu item.



NOTE

Turn on synchronous display only when it is necessary to display every data value. Using synchronous display results in a large performance penalty on multithreaded systems.

You also can use the Defer Panel Updates property to defer all new requests for front panel updates.

Monitor settings and the controls that you place on a front panel also can improve the performance of a VI. Lower the color depth and resolution of your monitor and enable hardware acceleration for your monitor. Refer to the documentation for your operating system for more information about hardware acceleration. Using controls from the **Classic** palette instead of controls from the **Modern** palette also improves the performance of a VI.

Memory Management

Consider the following suggestions to manage the *memory in a VI* for optimal execution speed.

- *Avoid constantly resizing data* so that LabVIEW does not have to make copies of the data.
- *Develop efficient data structures* with data types that are easy to manipulate to avoid making extra copies of data.
- Save VIs without separate compiled code in the current version of LabVIEW.

Compiler Optimizations

To improve editor responsiveness for large VIs, the LabVIEW compiler limits optimizations that could improve the execution speed of these VIs. When you finish developing a large VI, you can configure LabVIEW to *prioritize execution speed over editor responsiveness*. You can also configure LabVIEW to *apply the full set of compiler optimizations to an entire build specification*.

Passing Data within an Application

You can choose from many techniques to pass data within a LabVIEW application. The following list orders the efficiency of the most common ways.

1. **Wires** —Use wires to pass data and allow LabVIEW to control performance optimization. No other techniques are faster in a dataflow language because data has a single writer and one or more readers.
2. **Feedback Node** —Use a Feedback Node to store data from a previous VI or loop execution. A Feedback Node automatically appears when you wire the output of a subVI, function, or group of subVIs and functions to the input of that same VI, function, or group and enable **Auto-insert Feedback Node in cycles** on the Block Diagram page in the Options dialog box. LabVIEW enables **Auto-insert Feedback Node in cycles** by default. A Feedback Node holds the data from the last execution of the VI, function, or group and can pass it to the next execution or set a new initial value, depending on whether you *move the initializer terminal*. In loops, shift registers store data in the same way as the Feedback Node but require wires spanning the loop.
3. **Shift registers** —Use *shift registers* if you need storage or feedback within a loop. Shift registers pass data through one outside writer and reader, and one inside writer and reader. This tight scoping on data access maximizes the efficiency of LabVIEW. You can replace shift registers with a Feedback Node to achieve the same functionality without wires spanning the loop.
4. **Global Variables and Functional Global Variables** —Use *global variables* for simple data and simple access. For large or complex data, a global variable reads and passes all of the data. Use *functional global variables* to control how much data LabVIEW returns.

Using Controls, Control References, and Property Nodes as Variables

Though you can use controls, control references, and Property Nodes to pass data between VIs, they were not designed for use as variables because they work through the user interface. Use local variables and the Value property only when performing user interface actions or when stopping parallel loops.

User interface actions are historically slow on computers. LabVIEW passes a double value through a wire in nanoseconds, and draws a piece of text in hundreds of microseconds to milliseconds. For example, LabVIEW can pass a 100K array through a wire in 0 nanoseconds to a few microseconds. Drawing a graph of this 100K array takes tens of milliseconds. Because controls have a user interface attached, using controls to pass data has the side effect of redrawing controls, which adds memory expense and slows performance. If the controls are hidden, LabVIEW passes the data faster, but because the control can be displayed at anytime, LabVIEW still needs to update the control.

How Multithreading Affects User Interface Actions

Completing user interface actions uses more memory because *LabVIEW switches from the execution thread to the user interface thread*. For example, when you set the Value property, LabVIEW simulates a user changing the value of the control, stopping the execution thread and switching to the user interface thread to change the value. Then LabVIEW updates the operation buffer that stores data at the control and redraws the control if the front panel is open. LabVIEW then sends the data back to the execution thread in a protected area of memory called the transfer buffer. LabVIEW then switches back to the execution thread. The next time the execution thread reads from the control, LabVIEW finds the data in the transfer buffer and receives the new value for the execution buffer that stores data on the block diagram.

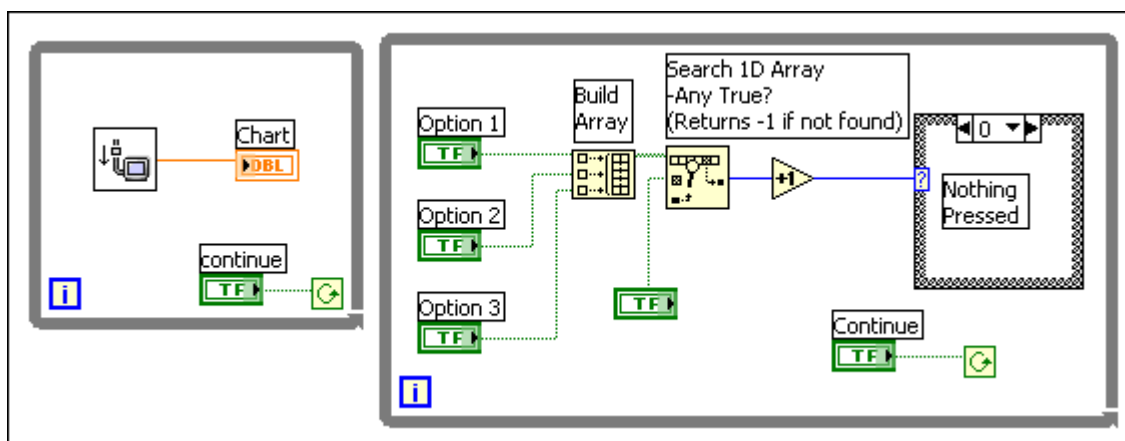
When you write to a local or global variable, LabVIEW does not switch to the user interface thread immediately. LabVIEW instead writes the value to the transfer buffer. The user interface updates at the next scheduled update time. It is possible to update a variable multiple times before a single thread switch or user interface update occurs. This is possible because variables operate solely in the execution thread.

Functional global variables can be more efficient than ordinary global variables because they do not use transfer buffers. Functional global variables exist only within the execution thread and do not use transfer buffers, unless you display their values on an open front panel.

Parallel Block Diagrams

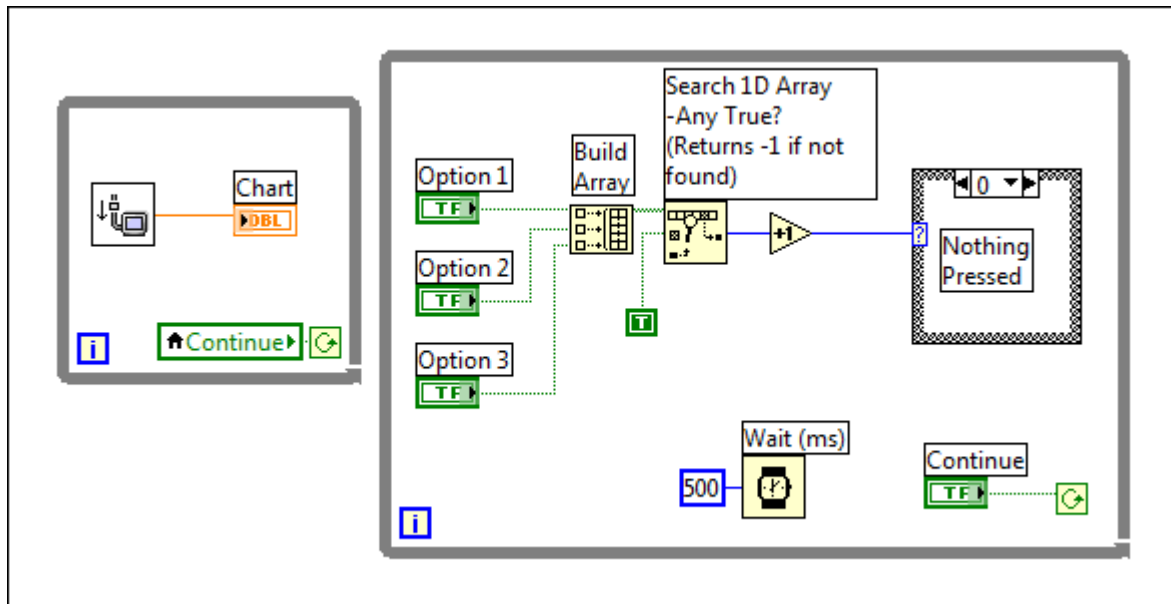
When you have multiple block diagrams running in parallel, the execution system switches between them periodically. If some of these loops are less important than others, use the Wait (ms) function to ensure the less important loops use less time.

For example, consider the following block diagram.



There are two loops in parallel. One of the loops is acquiring data and needs to execute as frequently as possible. The other loop is monitoring user input. The loops receive equal time because of the way this program is structured. The loop monitoring the user's action has a chance to run several times a second.

In practice, it is usually acceptable if the loop monitoring the button executes only once every half second, or even less often. By calling the Wait (ms) function in the user interface loop, you allot significantly more time to the other loop.



Parallel For Loop Iterations

When you use a For Loop, LabVIEW executes the loop iterations sequentially. If a For Loop is computationally intensive, consider *running the loop iterations in parallel* to improve performance. Parallel loop iterations allow LabVIEW to take advantage of multiple processors to execute the For Loop faster. However, parallel loop iterations must be independent of all other iterations. Use the Find Parallelizable Loops Results dialog box to determine which For Loops are parallelizable. To enable parallel loop iterations, right-click a For Loop border and select **Configure Iteration Parallelism** from the shortcut menu to display the For Loop Iteration Parallelism dialog box.

After you enable parallel loop iterations, the parallel instances terminal **P** appears below the count terminal **N**. You can use this terminal to specify the number of loop instances for LabVIEW to use as well as to identify which instance is executing a particular iteration.

Review the *performance considerations* before enabling parallel For Loop iterations.

National Instruments recommends that you *display warnings* in the Error list window so LabVIEW can warn you when parallel loop iterations might cause problems in a VI.

SubVI Overhead

When you call a subVI, there is a certain amount of overhead associated with the call. This overhead is fairly small (on the order of tens of microseconds), especially in comparison to I/O overhead and display overhead, which can range from milliseconds to tens of milliseconds. However, this overhead can add up in some cases. For example, if you call a subVI 10,000 times in a loop, this overhead could significantly affect execution speed. In this case, you might want to consider embedding the loop in the subVI.

Another way to minimize subVI overhead is to turn subVIs into subroutines by selecting Execution from the **Category** pull-down menu in the VI Properties dialog box and then selecting **subroutine** from the **Priority** pull-down menu. However, there are some trade-offs. Subroutines cannot display front panel data, call timing or dialog box functions, allow debugging or automatic error-handling, or multitask with other VIs. Subroutines are generally most appropriate for VIs that do not require user interaction and are short, frequently executed tasks.

A third way to minimize subVI overhead is to inline subVIs into their calling VIs. When you inline a subVI, LabVIEW inserts the compiled code of the subVI into the compiled code of the calling VI. If you then make changes to the subVI, LabVIEW recompiles all calling VIs of that subVI to include those changes. Essentially, inlining a subVI removes the need to call the subVI at run time. Instead, LabVIEW executes the subVI code inside the compiled code of the calling VI.

Inlining subVIs is most useful for small subVIs, subVIs within a loop, subVIs with unwired outputs, or subVIs you call only once. To inline a subVI, place a checkmark in the **Inline subVI into calling VIs** checkbox on the Execution page of the VI Properties dialog box. You must also select **Preallocated clone reentrant execution** on the same page of the dialog box. LabVIEW automatically preallocates clones for each instance when it inlines the subVI.

A subVI that you want to inline cannot have any of the following characteristics:

- Contains recursion
- Uses automatic error handling
- Contains certain block diagram nodes, such as Property Nodes and Invoke Nodes. These nodes generate edit-time error messages if you enable subVI inlining.
- Allows debugging. You must **disable debugging** for subVIs that you want to inline.
- Is one of the following types of VIs:
 - Dynamic dispatch VI of a LabVIEW class
 - Targeted for CGen or FPGA
 - Control VI - VIs that define a custom control or indicator
 - Polymorphic VI
 - Global variable VI
 - XControl VI
 - Method VI of an XControl Library
 - VI that can only be placed on a simulation diagram in the LabVIEW Control Design and Simulation Module

Even if the subVI does not have any of the previous characteristics, LabVIEW does not inline the subVI into the following kinds of calls:

- Dynamic calls to the subVI
- Calls to the subVI that enable database access

Finally, when you inline a subVI, LabVIEW ignores any priority or preferred execution settings.

Refer to the VI Execution Properties VI in the `examples\Performance\VI Properties` for an example of the performance improvements that result from inlining a subVI.

Shared Variable Overhead

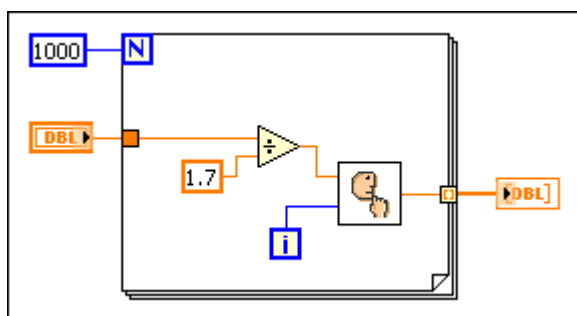
Network-published shared variables incur significant CPU and memory overhead. If your application uses a large number of network-published shared variables, consider using the Simple Messaging Reference Library (STM) instead. Refer to the support document at ni.com to learn more about STM.

If you need to send a continuous stream of data from one computing device to one other computing device, use the Network Streams functions.

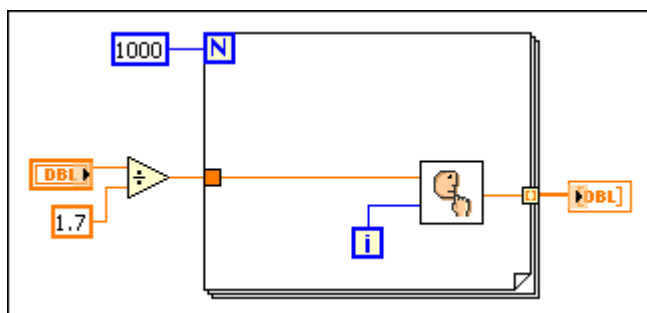
Unnecessary Computation in Loops

Avoid putting a calculation in a loop if the calculation produces the same value for every iteration. Instead, move the calculation out of the loop and pass the result into the loop.

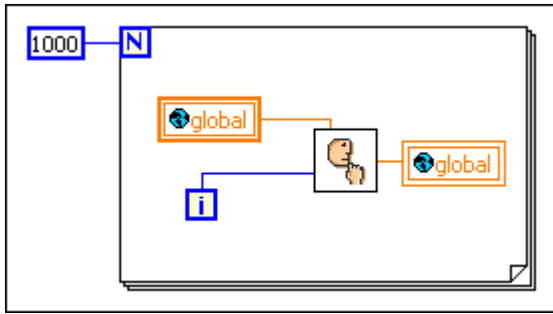
For example, examine the following block diagram.



The result of the division is the same every time through the loop; therefore you can increase performance by moving the division out of the loop, as shown in the following block diagram.

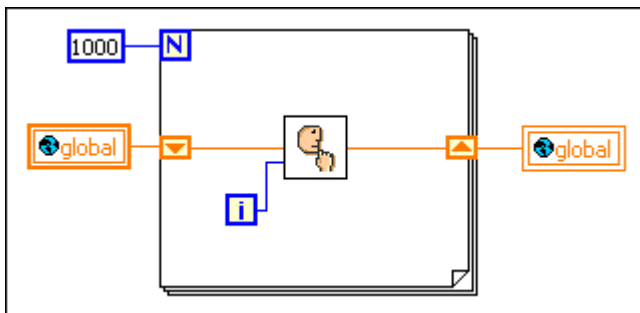


Refer to the following block diagram.

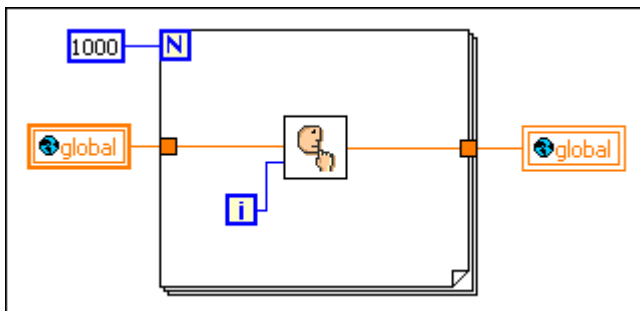


If you know the value of the global variable is not going to be changed by another concurrent block diagram or VI during this loop, this block diagram wastes time by reading from the global variable and writing to the global every time through the loop.

If you do not require the global variable to be read from or written to by another block diagram during this loop, you might use the following block diagram instead.



Notice that the shift registers must pass the new value from the subVI to the next iteration of the loop. The following block diagram shows a common mistake some beginning users make. Since there is no shift register, the results from the subVI never pass back to the subVI as its new input value.



VI Memory Usage

LabVIEW handles many of the details that you must handle in a text-based programming language. One of the main challenges of a text-based language is memory usage. In a text-based language, you, the programmer, have to take care of allocating memory before you use it and deallocating it when you finish. You also must be careful not to write past the end of the memory you allocated in the first place. Failure to allocate memory or to allocate enough memory is one of the biggest mistakes programmers make in text-based languages. Inadequate memory allocation is also a difficult problem to debug.

The dataflow paradigm for LabVIEW removes much of the difficulty of managing memory. In LabVIEW, you do not allocate variables, nor assign values to and from them. Instead, you create a block diagram with connections representing the transition of data.

Functions that generate data take care of allocating the storage for that data. When data is no longer being used, the associated memory is deallocated. When you add new information to an array or a string, enough memory is automatically allocated to manage the new information.

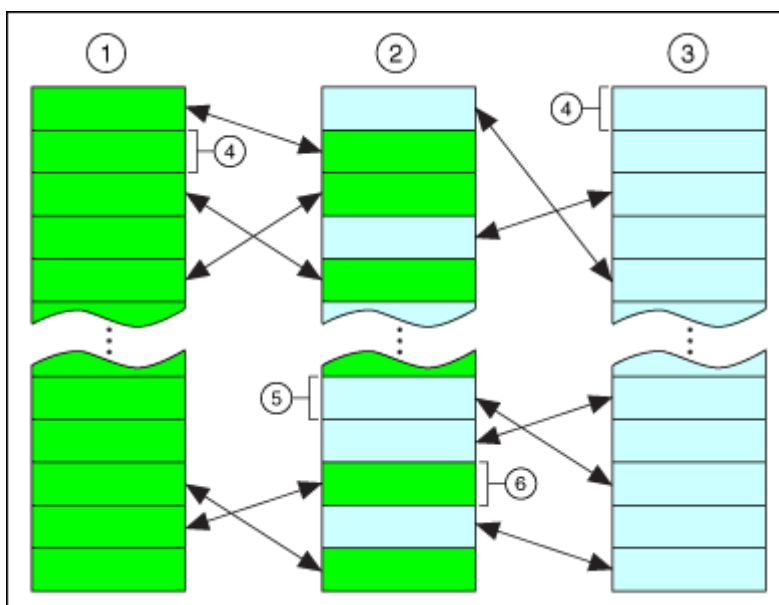
This automatic memory handling is one of the chief benefits of LabVIEW. However, because it is automatic, you have less control of when it happens. If your program works with large sets of data, it is important to have some understanding of when memory allocation takes place. An understanding of the principles involved can result in programs with significantly smaller memory requirements. Also, an understanding of how to minimize memory usage also can help to increase VI execution speeds, because memory allocation and copying data can take a considerable amount of time.

Virtual Memory

Operating systems use virtual memory to allow applications to access more memory than what is available in physical RAM. The OS partitions physical RAM into blocks called pages. When an application or *process* assigns an address to a block of memory, the address does not refer to a direct location in physical RAM, but instead refers to memory in a page. The OS can swap these pages between physical RAM and the hard disk.

If an application or process needs to access a certain block or page of memory that is not in physical RAM, the OS can move a page of physical RAM that an application or process is not using to the hard disk and replace it with the required page. The OS keeps track of the pages in memory and translates virtual addresses to pages into real addresses in physical RAM when an application or process needs to access that memory.

The following image illustrates how two processes can swap pages in and out of physical RAM. For this example, Process A and Process B are running simultaneously.



1 Process A	3 Process B	5 Page of memory from Process B
2 Physical RAM	4 Page of virtual memory	6 Page of memory from Process A

Because the number of pages an application or process uses depends on available disk space and not on available physical RAM, an application can use more memory than actually is available in physical RAM. The size of the memory addresses an application uses limits the amount of virtual memory that application can access.

LabVIEW Virtual Memory Usage

LabVIEW (32-bit) uses 32-bit addresses and is large address aware.

LabVIEW (64-bit) can access up to 8 TB or 128 TB of virtual memory depending on the version of Windows. The actual size of virtual memory that LabVIEW (64-bit) can access also depends on the size of the physical RAM and the maximum paging file size.

VI Component Memory Management

A VI has the following four major components:

- Front panel
- Block diagram
- Code (diagram compiled to machine code)
- Data (control and indicator values, default data, diagram constant data, and so on)

When a VI loads, the front panel, the code (if it matches the platform), and the data for the VI are loaded into memory. If the VI needs to be recompiled because of a change in platform or a change in the interface of a subVI, the block diagram is loaded into memory as well.

The VI also loads the code and data space of its subVIs into memory. Under certain circumstances, the front panel of some subVIs might be loaded into memory as well. For example, this can occur if the subVI uses Property Node, because Property Nodes manipulate state information for front panel controls.

An important point in the organization of VI components is that you generally do not use much more memory when you convert a section of your VI into a subVI. If you create a single, huge VI with no subVIs, you end up with the front panel, code, and data for that top-level VI in memory. However, if you break the VI into subVIs, the code for the top-level VI is smaller, and the code and data of the subVIs reside in memory. In some cases, you might see lower run-time [memory usage](#).

Large VIs also take longer to edit. You can avoid this problem in the following ways:

- (Recommended) Divide your VI into subVIs. The editor can handle smaller VIs more efficiently, and a more hierarchical VI organization is generally easier to maintain and read.
- Configure LabVIEW to *prioritize editor responsiveness over execution time* based on the *compiled code complexity* of the large VI.

**NOTE**

If the front panel or block diagram of a given VI is much larger than a screen, you might want to break it into subVIs to make it more accessible.

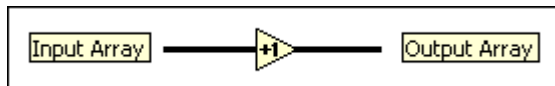
Dataflow Programming and Data Buffers

In dataflow programming, you generally do not use variables. Dataflow models usually describe nodes as consuming data inputs and producing data outputs. A literal implementation of this model produces applications that can use very large amounts of memory with sluggish performance. Every function produces a copy of data for every destination to which an output is passed. LabVIEW improves on this implementation through inplaceness.

LabVIEW uses inplaceness to determine when to reuse memory and whether to make copies for each output terminal. When LabVIEW does not copy data from an input to an output, the data of the input and output is in-place.

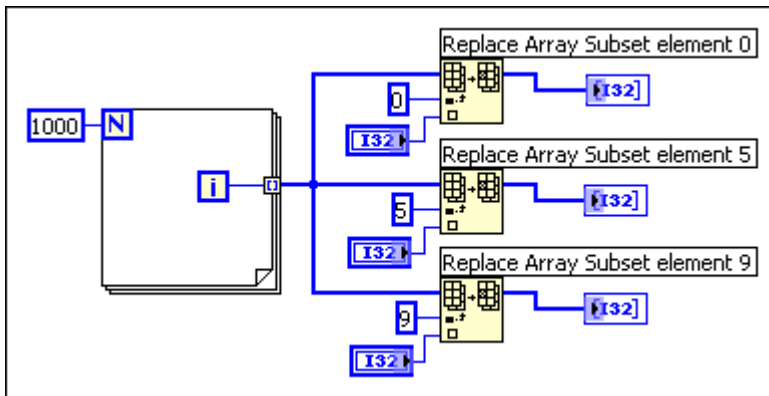
You can use the Show Buffer Allocations window to identify where LabVIEW can create copies of data.

For example, in a more traditional approach to the compiler, the following block diagram uses two blocks of data memory, one for the input and one for the output.



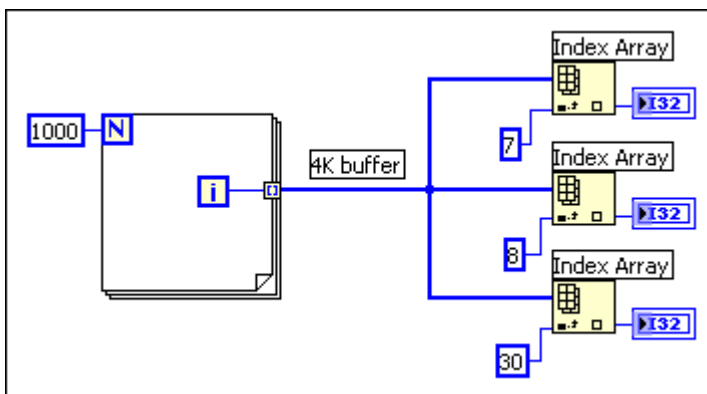
The input array and the output array contain the same number of elements, and the data type for both arrays is the same. Think of the incoming array as a buffer of data. Instead of creating a new buffer for the output, the compiler reuses the memory of the input buffer for the output buffer. This example of inplaceness saves memory and also results in faster execution because no memory allocation needs to take place at run time.

However, the compiler cannot reuse memory buffers in all cases, as shown in the following block diagram.



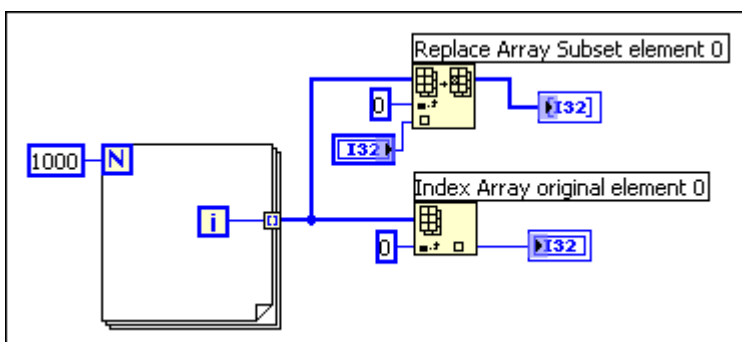
A signal passes a single source of data to multiple destinations. The Replace Array Subset function modifies the input array to produce the output array. In this case, the data for one of the functions is in-place and the output reuses the data for the input array, while the other outputs do not. Thus, the compiler creates new data buffers for two of the functions and copies the array data into the buffers. This block diagram uses about 12 KB (4 KB for the original array and 4 KB for each of the extra two data buffers).

Now, examine the following block diagram.



As before, the input branches to three functions. However, in this case the Index Array function does not modify the input array. If you pass data to multiple locations, all of which read the data without modifying it, LabVIEW does not make a copy of the data. As a result, all the data is in-place. This block diagram uses about 4 KB of memory.

Finally, consider the following block diagram.



In this case, the input branches to two functions, one of which modifies the data. There is no dependency between the two functions. Therefore, you can predict that at least one copy needs to be made so the Replace Array Subset function can safely modify the data. In this case, however, the compiler schedules the execution of the functions in such a way that the function that reads the data executes first, and the function that modifies the data executes last. This way, the Replace Array Subset function reuses the incoming array buffer without generating a duplicate array, making the data for this function in-place. If the ordering of the nodes is important, make the ordering explicit by using either a sequence or an output of one node for the input of another.

In practice, the analysis of block diagrams by the compiler is not perfect. In some cases, the compiler might not be able to determine the optimal method for reusing block diagram memory.

Conditional Indicators and Data Buffers

The way you build a block diagram can prevent LabVIEW from reusing data buffers. Using a conditional indicator in a subVI prevents LabVIEW from optimizing data buffer usage. A conditional indicator is an indicator inside a Case structure or For Loop. Placing an indicator in a conditionally executed code path will break the flow of data through the system and LabVIEW will not reuse the data buffer from the input, but force a data copy into the indicator instead. When you place indicators outside of Case structures and For Loops, LabVIEW directly modifies the data inside the loop or structure and passes the data to the indicator instead of creating a data copy. You can create constants for alternate cases instead of placing indicators inside the Case structure.

Monitoring Memory Usage

You can determine memory usage using several methods.

To view memory usage for the current VI, select **File»VI Properties** and select Memory Usage from the top pull-down menu. Notice that the information does not include memory use of subVIs. You can use the Profile Performance and Memory window to monitor the memory used by all VIs in memory and identify a subVI with performance issues. The VI Profile Performance and Memory window *keeps statistics* on the minimum, maximum, and average number of bytes and blocks used by each VI per run.

Note To monitor accurate VI run-time performance, save all VIs without *separate compiled code* in the current version of LabVIEW for the following reasons:

- Using the Undo feature creates temporary copies of objects and data, which can increase the reported memory requirements of a VI. Saving the VI purges the copies generated by Undo, resulting in accurate reports of memory information.
 - Running unsaved VIs from previous versions of LabVIEW can increase reported memory requirements and decrease VI run-time performance.
-

Use the Show Buffer Allocations window to identify specific areas on the block diagram where LabVIEW allocates memory space in the form of a buffer.

Note The Show Buffer Allocations window is available only in the LabVIEW Full and Professional Development Systems.

Select **Tools»Profile»Show Buffer Allocations** to display the Show Buffer Allocations window. Place a checkmark next to the data type(s) you want to see buffers for and click the **Refresh** button. The black squares that appear on the block diagram indicate where LabVIEW creates buffers to allocate space for data.

The amount of memory LabVIEW allocates for each buffer is the same size as the largest [data type](#). When you run a VI, LabVIEW might or might not use those buffers to store data. You cannot know whether LabVIEW makes a copy of the data because LabVIEW makes that decision at run time or the VI might depend on dynamic data.

If the VI needs memory for a buffer allocation, LabVIEW makes a data copy for the buffer. If LabVIEW is not sure whether the buffer needs a data copy, LabVIEW sometimes still makes a copy for the buffer.

Note The Show Buffer Allocations window identifies only execution buffers with the use of the black squares on the block diagram. Using this window does not identify operation buffers on the front panel.

Once you know where LabVIEW creates buffers, you might be able to [edit the VI to reduce the amount of memory](#) LabVIEW requires to run the VI. LabVIEW must allocate memory to run the VI, so you cannot eliminate all the buffers.

If you make a change to a VI that requires LabVIEW to recompile the VI, the black squares disappear because the buffer information might no longer be correct. Click the **Refresh** button on the Show Buffer Allocations window to recompile the VI and display the black squares. When you close the **Show Buffer Allocations** window, the black squares disappear.

Note You also can use the LabVIEW Desktop Execution Trace Toolkit to monitor thread usage, memory leaks, and other aspects of LabVIEW programming.

Select **Help»About LabVIEW** to view a statistic that summarizes the total amount of memory used by your application. This includes memory for VIs as well as memory the application uses. You can check this amount before and after execution of a set of VIs to obtain a rough idea of how much memory the VIs are using.

Rules for Better Memory Usage

The main point of the previous section is that the compiler attempts to reuse memory intelligently. The rules for when the compiler can and cannot reuse memory are complex. In practice, the following rules can help you to create VIs that use memory efficiently:

- Breaking a VI into subVIs generally does not hurt your memory usage. In many cases, memory usage improves because the execution system can reclaim subVI data memory when the subVI is not executing.
- Do not worry too much about copies of scalar values; it takes a lot of scalars to negatively affect memory usage.
- Do not overuse global and local variables when working with arrays or strings. Reading a global or local variable causes LabVIEW to generate a copy of the data.

- Do not display large arrays and strings on open front panels unless it is necessary. Controls and indicators on open front panels retain a copy of the data they display.

Tip If a chart indicator is necessary, know that the chart history will retain a copy of the data it displays. This continues until the chart history is filled, then LabVIEW stops taking more memory. LabVIEW does not automatically clear the chart history when the VI restarts, so it helps to clear the chart history throughout the execution of the program. To do this, write empty arrays to the History Data attribute node for the chart.

- Use the Defer Panel Updates property. When you set this property to TRUE, the front panel indicator values do not change, even if you change the value of a control. Your operating system does not have to use any memory to populate the indicator(s) with the new value(s).

Note LabVIEW does not typically open a subVI front panel when calling the subVI.

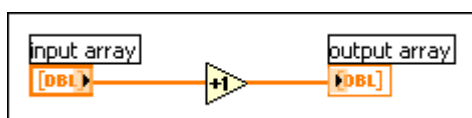
- If the front panel of a subVI is not going to be displayed, do not leave unused Property Nodes on the subVI. Property Nodes cause the front panel of a subVI to remain in memory, which can cause unnecessary memory usage.
- When designing block diagrams, watch for places where the size of an input is different from the size of an output. For example, if you see places where you are frequently increasing the size of an array or string using the Build Array or Concatenate Strings functions, you are generating copies of data.
- Use *consistent data types* for arrays and watch for coercion dots when passing data to subVIs and functions. When you change data types, the execution system makes a copy of the data.
- Do not use complicated, hierarchical data types such as clusters or arrays of clusters containing large arrays or strings. You might end up using more memory. If possible, *use more efficient data types*.
- Do not use transparent and overlapped front panel objects unless they are necessary. Such objects might use more memory.

Refer to the [LabVIEW Style Checklist](#) for more specific ideas on how to optimize the performance of a VI.

Memory Issues in Front Panels

When a front panel is open, controls and indicators keep their own, private copy of the data they display.

The following block diagram shows the increment function, with the addition of front panel controls and indicators.



When you run the VI, the data of the front panel control is passed to the block diagram. The increment function reuses the input buffer. The indicator then makes a copy of the data for display purposes. Thus, there are three copies of the buffer.

This data protection of the front panel control prevents the case in which you enter data into a control, run the associated VI, and see the data change in the control as it is passed in-place to subsequent nodes. Likewise, data is protected in the case of indicators so that they can reliably display the previous contents until they receive new data.

With subVIs, you can use controls and indicators as inputs and outputs. The execution system makes a copy of the control and indicator data of the subVI in the following conditions:

- The front panel is in memory. This can occur for any of the following reasons:
 - The front panel is open.
 - The VI has been changed but not saved (all components of the VI remain in memory until the VI is saved).
- The panel uses data printing.
- The block diagram uses Property Nodes.
- The VI uses local variables.
- The panel uses data logging.

For a Property Node to be able to read the chart history in subVIs with closed panels, the control or indicator needs to display the data passed to it. Because there are numerous other attributes like this, the execution system keeps subVI panels in memory if the subVI uses Property Nodes.

If a front panel uses front panel data logging or data printing, controls and indicators maintain copies of their data. In addition, panels are kept in memory for data printing so the panel can be printed.

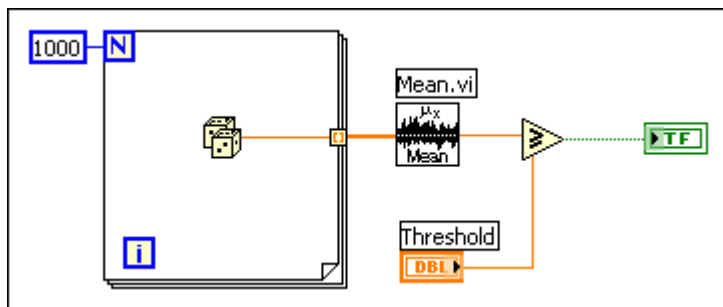
If you set a subVI to display its front panel when called using the VI Properties dialog box or the SubVI Node Setup dialog box, the front panel loads into memory when you call the subVI. If you set the **Close afterwards if originally closed** item, LabVIEW removes the front panel from memory when the subVI finishes execution.

SubVIs Can Reuse Data Memory

In general, a subVI can use data buffers from its caller as easily as if the block diagrams of the subVI were duplicated at the top level. In most cases, you do not use more memory if you convert a section of your block diagram into a subVI. For VIs with special display requirements, as described in the previous section, there might be some additional memory usage for front panels and controls.

Understanding When Memory Is Deallocated

Consider the following block diagram.



After the Mean VI executes, the array of data is no longer needed. Because determining when data is no longer needed can become very complicated in larger block diagrams, the execution does not deallocate the data buffers of a particular VI during its execution.

On macOS, if the execution system is low on memory, it deallocates data buffers used by any VI not currently executing. The execution system does not deallocate memory used by front panel controls, indicators, global variables, or uninitialized shift registers.

Now consider the same VI as a subVI of a larger VI. The array of data is created and used only in the subVI. On macOS, if the subVI is not executing and the system is low on memory, it might deallocate the data in the subVI. This is a case in which using subVIs can save on memory usage.

On Windows and Linux platforms, data buffers are not normally deallocated unless a VI is closed and removed from memory. Memory is allocated from the operating system as needed, and virtual memory works well on these platforms. Due to fragmentation, the application might appear to use more memory than it really does. As memory is allocated and freed, the application tries to consolidate memory usage so it can return unused blocks to the operating system.

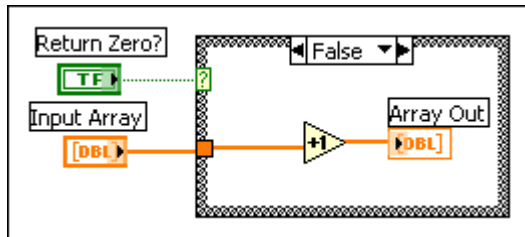
You can use the Request Deallocation function to deallocate unused memory after the VI that contains this function runs. This function is only used for advanced performance optimizations. Deallocating unused memory can improve performance in some cases. However, aggressively deallocating memory can cause LabVIEW to reallocate space repeatedly rather than reusing an allocation. Use this function if your VI allocates a large amount of data but never reuses that allocation. When a top-level VI calls a subVI, LabVIEW allocates a data space of memory in which that subVI runs. When the subVI finishes running, LabVIEW does not deallocate the data space until the top-level VI finishes running or until the entire application stops, which can result in out-of-memory conditions and degradation of performance. Place the Request Deallocation function in the subVI you want to deallocate memory for. When you set the **flag** Boolean input to TRUE, LabVIEW reduces memory usage by deallocating the data space for the subVI.

Determining When Outputs Can Reuse Input Buffers

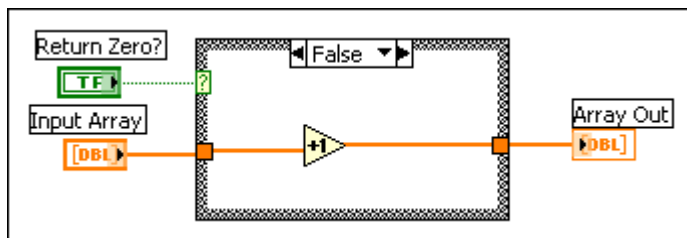
If an output is the same size and data type as an input, and the input is not required elsewhere, the output can reuse the input buffer. As mentioned previously, in some cases even when an input is used elsewhere, the compiler and the execution system can order code execution in such a way that it can reuse the input for an output buffer. However, the rules for this are complex. Do not depend on them.

You can use the Show Buffer Allocations window to see if an output buffer reuses the input buffer. In the following block diagram, placing an indicator in each case of a Case structure breaks the flow of data because LabVIEW creates a copy of the data for each indicator.

LabVIEW does not use the buffer it created for the input array but instead creates a copy of the data for the output array.



If you move the indicator outside the Case structure, the output buffer reuses the input buffer because LabVIEW does not need to make a copy of the data the indicator displays. Because LabVIEW does not require the value of the input array later in the VI, the increment function can directly modify the input array and pass it to the output array. In this situation, LabVIEW does not need to copy the data, so a buffer does not appear on the output array, as shown in the following block diagram.



Optimizing Memory Usage by Matching Data Types

If you use consistent data types, LabVIEW can reuse the memory buffers allocated for an input when generating an output. This can improve the memory usage and execution time of the VI. However, if an input has a different data type from an output, the output cannot reuse the memory LabVIEW allocates for that input.

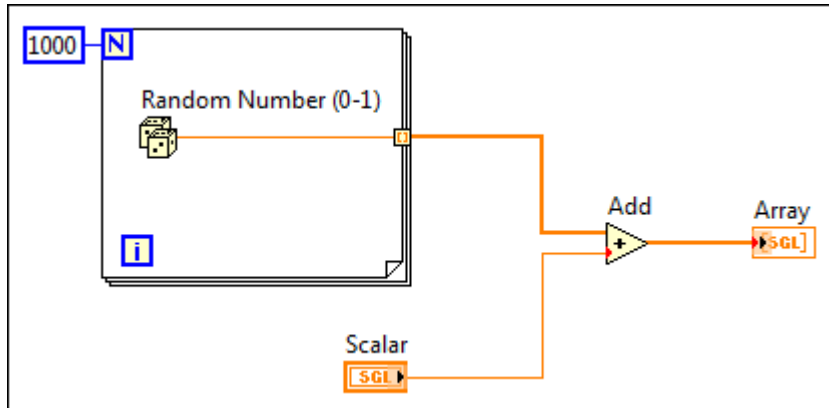
For example, if you add a 32-bit integer to a 16-bit integer, LabVIEW coerces the 16-bit integer to a 32-bit integer. The compiler creates a new buffer for the coerced data, and LabVIEW places a *coercion dot* on the Add function. In this example, LabVIEW can use the 32-bit integer input for the output buffer, assuming the input meets all the other requirements. However, because LabVIEW coerces the 16-bit integer, LabVIEW cannot reuse the memory allocated for that input.

To minimize memory usage, use consistent data types wherever possible. Using consistent data types produces fewer copies of data that result when LabVIEW coerces data to a larger size. In some applications, you also may consider using smaller data types to minimize data usage. For example, you may consider using four-byte, single-precision numbers instead of eight-byte, double-precision numbers. However, to avoid unnecessary coercions, match the data types of objects wired to a subVI with the data type the subVI expects.

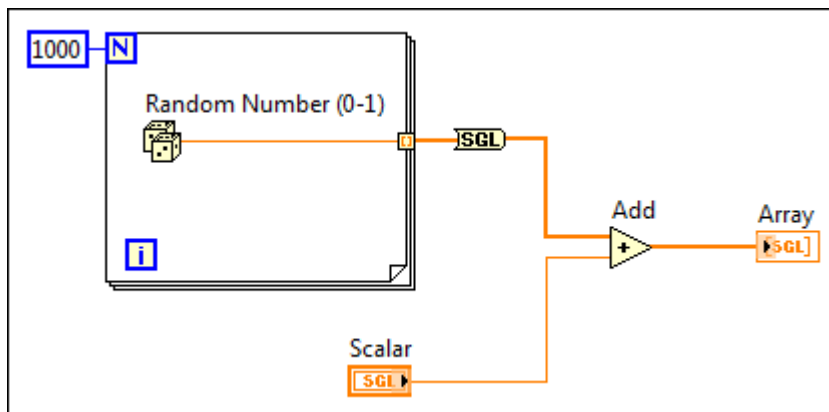
Optimizing Memory Usage When Generating Data of a Specific Type

When you need to generate data of a specific data type, you may have to convert data types on your block diagram. However, you can optimize memory usage by converting data types using Conversion functions before LabVIEW creates large arrays.

In the following example, the output data needs to be of the single-precision, floating-point data type to match the data type of an input of another VI. LabVIEW creates an array of 1,000 random values and adds the array to a scalar value. LabVIEW places a coercion dot on the Add function because LabVIEW coerces the single-precision, floating-point scalar data type to match the random double-precision, floating-point numbers.

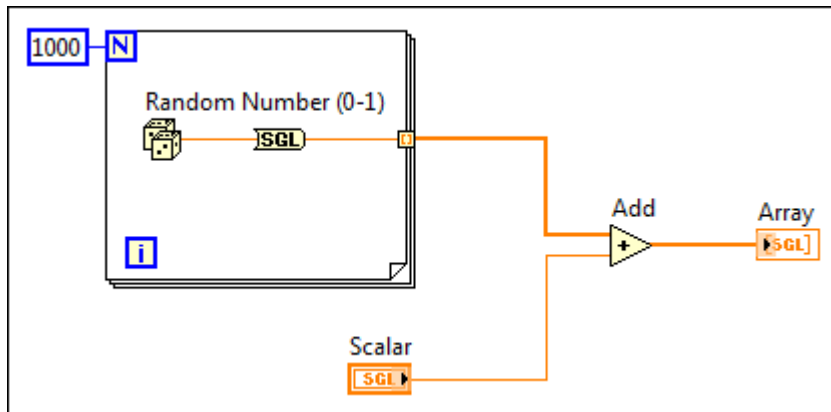


The following block diagram shows an attempt to correct this problem by converting the array of double-precision, floating-point numbers using a To Single Precision Float function.



Because the function still converts the data type of the large array after LabVIEW creates the array, the VI uses the same amount of memory as the example with the coercion dots.

The following block diagram illustrates how to optimize memory usage and execution speed by converting the random number before LabVIEW creates the array.



When you cannot avoid conversions, you can use a Conversion function before LabVIEW creates a large array to avoid converting the large data buffer allocated for the array from one data type to another. Converting data before LabVIEW creates a large array optimizes the memory usage of a VI.

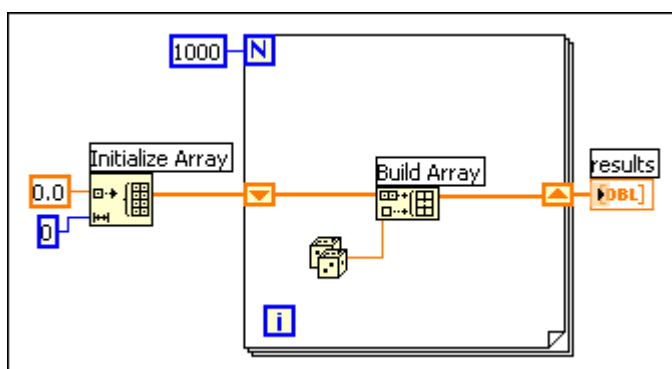
Avoid Constantly Resizing Data

If the size of an output is different from the size of an input, the output does not reuse the input data buffer. This is the case for functions such as Build Array, Concatenate Strings, and Array Subset, which increase or decrease the size of an array or string. When working with arrays and strings, avoid constantly using these functions, because your program uses more data memory and executes more slowly because it is constantly copying data.

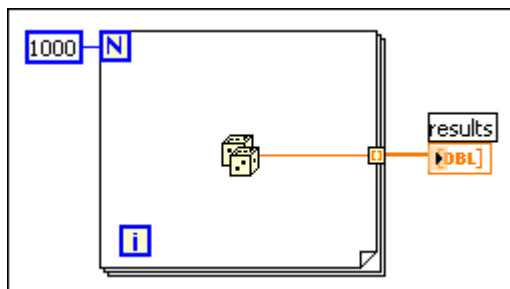
Example 1: Building Arrays

Consider the following block diagram, which is used to create an array of data. This block diagram creates an array in a loop by constantly calling Build Array to concatenate a new element. The input array is reused by Build Array. The VI continually resizes the buffer in each iteration to make room for the new array and appends the new element. The resulting execution speed is slow, especially if the loop is executed many times.

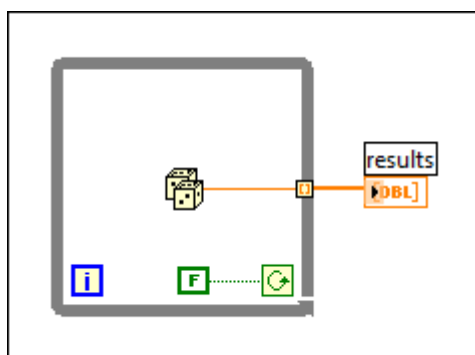
Note When you manipulate arrays, clusters, waveforms, and variants, you can use the In Place Element structure to improve memory usage in VIs.



If you want to add a value to the array with every iteration of the loop, you can see the best performance by using auto-indexing on the edge of a loop. With For Loops, the VI can predetermine the size of the array (based on the value wired to N), and resize the buffer only once.



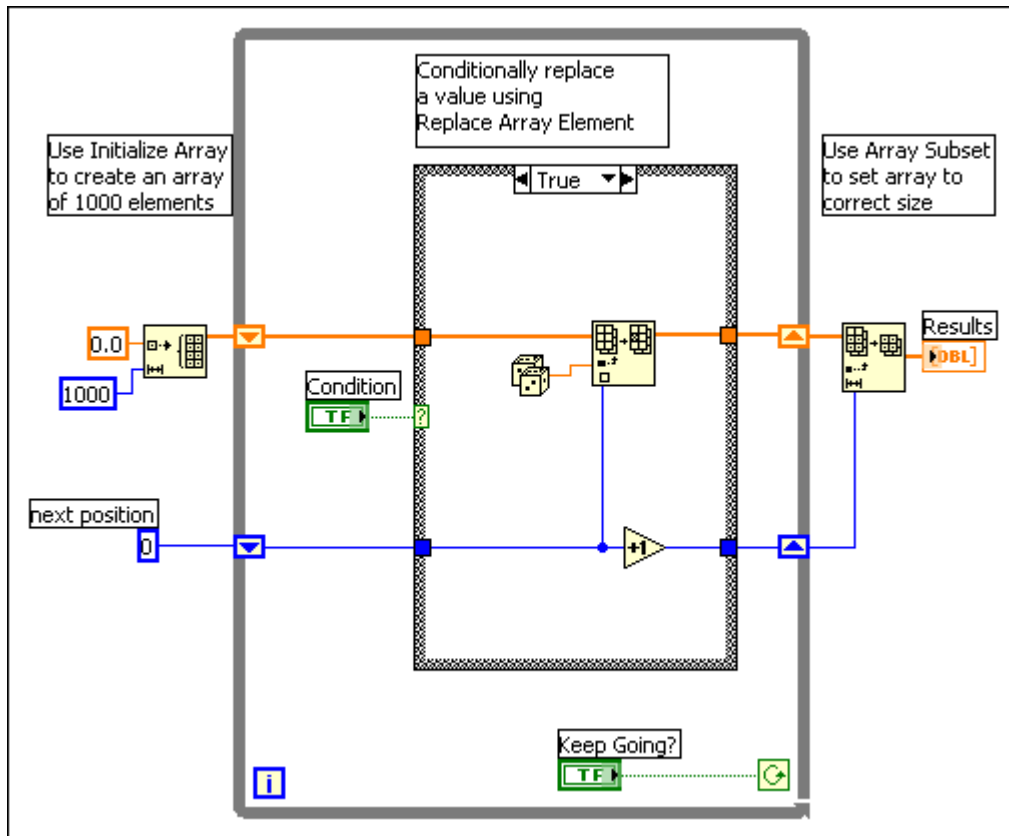
With While Loops, auto-indexing is not quite as efficient, because the end size of the array is not known. However, While Loop auto-indexing avoids resizing the output array with every iteration by increasing the output array size in large increments. When the loop is finished, the output array is resized to the correct size. The performance of While Loop auto-indexing is nearly identical to For Loop auto-indexing.



Auto-indexing assumes you are going to add a value to the resulting array with each iteration of the loop. If you must conditionally add values to an array but can determine an upper limit on the array size, you might consider preallocating the array and using Replace Array Subset to fill the array.

When you finish filling the array values, you can resize the array to the correct size. The array is created only once, and Replace Array Subset can reuse the input buffer for the output buffer. The performance of this is very similar to the performance of loops using auto-indexing. If you use this technique, be careful the array in which you are replacing values is large enough to hold the resulting data, because Replace Array Subset does not resize arrays for you.

An example of this process is shown in the following block diagram.

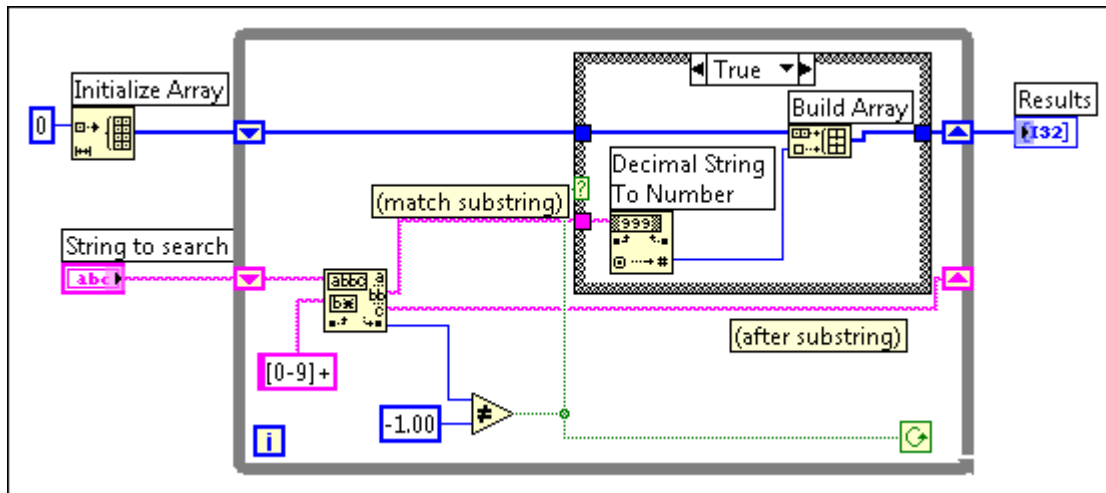


Example 2: Searching through Strings

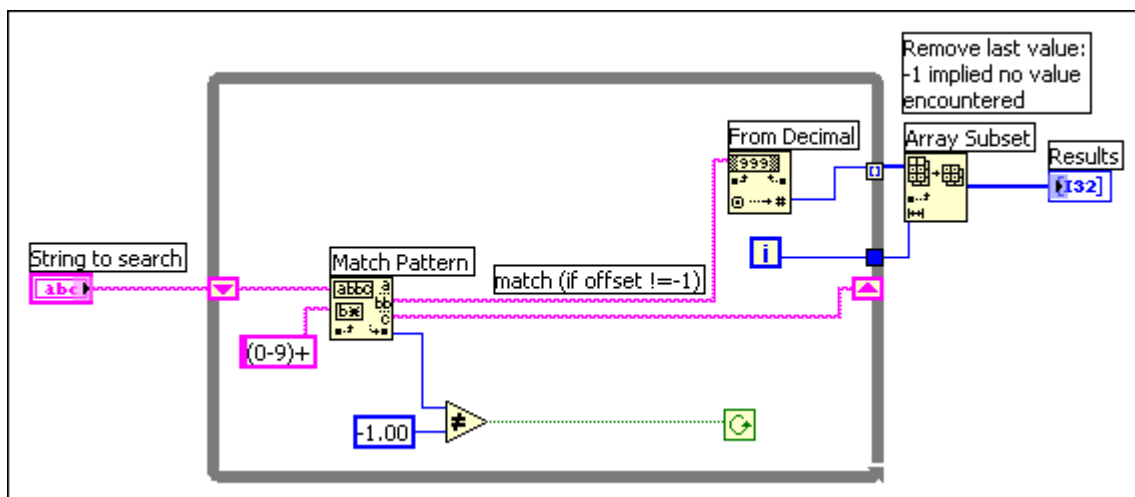
You can use the Match Pattern function to search a string for a pattern. Depending on how you use it, you might slow down performance by unnecessarily creating string data buffers.

Assuming you want to match an integer in a string, you can use `[0-9]+` as the regular expression input to this function. To create an array of all integers in a string, use a loop and call Match Regular Expression repeatedly until the offset value returned is -1.

The following block diagram is one method for scanning for all occurrences of integers in a string. It creates an empty array and then searches the remaining string for the numeric pattern with each iteration of the loop. If the pattern is found (offset is not -1, this block diagram uses Build Array to add the number to a resulting array of numbers. When there are no values left in the string, Match Regular Expression returns -1 and the block diagram completes execution.



One problem with this block diagram is that it uses Build Array in the loop to concatenate the new value to the previous value. Instead, you can use auto-indexing to accumulate values on the edge of the loop. Notice you end up seeing an extra unwanted value in the array from the last iteration of the loop where Match Regular Expression fails to find a match. A solution is to use Array Subset to remove the extra unwanted value. This is shown in the following block diagram.



The other problem with this block diagram is that you create an unnecessary copy of the remaining string every time through the loop. Match Regular Expression has

VI Component Memory Management

A VI has the following four major components:

- Front panel
- Block diagram
- Code (diagram compiled to machine code)
- Data (control and indicator values, default data, diagram constant data, and so on)

When a VI loads, the front panel, the code (if it matches the platform), and the data for the VI are loaded into memory. If the VI needs to be recompiled because of a change in platform or a change in the interface of a subVI, the block diagram is loaded into memory as well.

The VI also loads the code and data space of its subVIs into memory. Under certain circumstances, the front panel of some subVIs might be loaded into memory as well. For example, this can occur if the subVI uses Property Node, because Property Nodes manipulate state information for front panel controls.

An important point in the organization of VI components is that you generally do not use much more memory when you convert a section of your VI into a subVI. If you create a single, huge VI with no subVIs, you end up with the front panel, code, and data for that top-level VI in memory. However, if you break the VI into subVIs, the code for the top-level VI is smaller, and the code and data of the subVIs reside in memory. In some cases, you might see lower run-time [memory usage](#).

Large VIs also take longer to edit. You can avoid this problem in the following ways:

- (Recommended) Divide your VI into subVIs. The editor can handle smaller VIs more efficiently, and a more hierarchical VI organization is generally easier to maintain and read.
- Configure LabVIEW to [prioritize editor responsiveness over execution time](#) based on the [compiled code complexity](#) of the large VI.



Note If the front panel or block diagram of a given VI is much larger than a screen, you might want to break it into subVIs to make it more accessible.

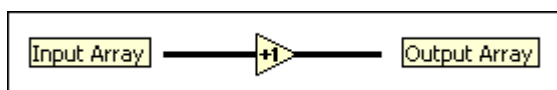
Dataflow Programming and Data Buffers

In dataflow programming, you generally do not use variables. Dataflow models usually describe nodes as consuming data inputs and producing data outputs. A literal implementation of this model produces applications that can use very large amounts of memory with sluggish performance. Every function produces a copy of data for every destination to which an output is passed. LabVIEW improves on this implementation through inplaceness.

LabVIEW uses inplaceness to determine when to reuse memory and whether to make copies for each output terminal. When LabVIEW does not copy data from an input to an output, the data of the input and output is in-place.

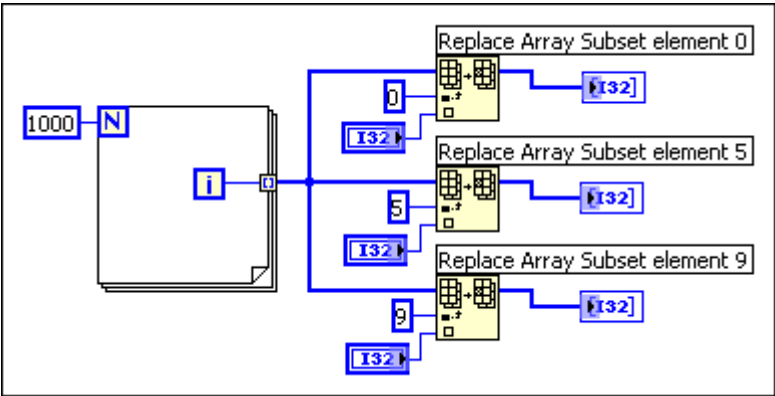
You can use the Show Buffer Allocations window to identify where LabVIEW can create copies of data.

For example, in a more traditional approach to the compiler, the following block diagram uses two blocks of data memory, one for the input and one for the output.



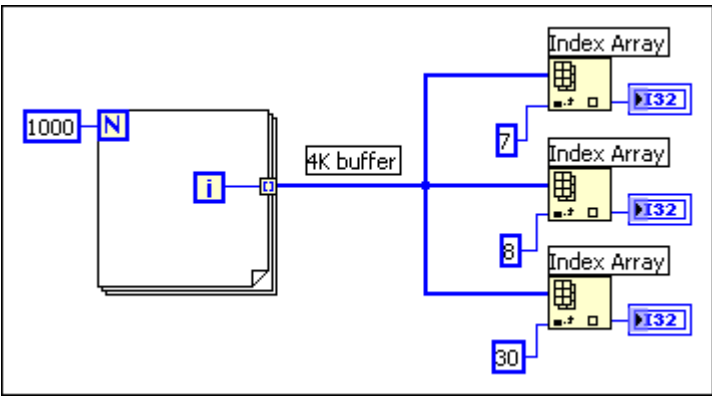
The input array and the output array contain the same number of elements, and the data type for both arrays is the same. Think of the incoming array as a buffer of data. Instead of creating a new buffer for the output, the compiler reuses the memory of the input buffer for the output buffer. This example of inplaceness saves memory and also results in faster execution because no memory allocation needs to take place at run time.

However, the compiler cannot reuse memory buffers in all cases, as shown in the following block diagram.



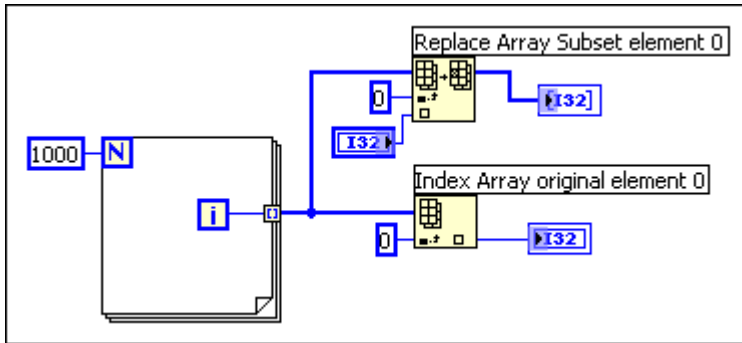
A signal passes a single source of data to multiple destinations. The Replace Array Subset function modifies the input array to produce the output array. In this case, the data for one of the functions is in-place and the output reuses the data for the input array, while the other outputs do not. Thus, the compiler creates new data buffers for two of the functions and copies the array data into the buffers. This block diagram uses about 12 KB (4 KB for the original array and 4 KB for each of the extra two data buffers).

Now, examine the following block diagram.



As before, the input branches to three functions. However, in this case the Index Array function does not modify the input array. If you pass data to multiple locations, all of which read the data without modifying it, LabVIEW does not make a copy of the data. As a result, all the data is in-place. This block diagram uses about 4 KB of memory.

Finally, consider the following block diagram.



In this case, the input branches to two functions, one of which modifies the data. There is no dependency between the two functions. Therefore, you can predict that at least one copy needs to be made so the Replace Array Subset function can safely modify the data. In this case, however, the compiler schedules the execution of the functions in such a way that the function that reads the data executes first, and the function that modifies the data executes last. This way, the Replace Array Subset function reuses the incoming array buffer without generating a duplicate array, making the data for this function in-place. If the ordering of the nodes is important, make the ordering explicit by using either a sequence or an output of one node for the input of another.

In practice, the analysis of block diagrams by the compiler is not perfect. In some cases, the compiler might not be able to determine the optimal method for reusing block diagram memory.

Conditional Indicators and Data Buffers

The way you build a block diagram can prevent LabVIEW from reusing data buffers. Using a conditional indicator in a subVI prevents LabVIEW from optimizing data buffer usage. A conditional indicator is an indicator inside a Case structure or For Loop. Placing an indicator in a conditionally executed code path will break the flow of data through the system and LabVIEW will not reuse the data buffer from the input, but force a data copy into the indicator instead. When you place indicators outside of Case structures and For Loops, LabVIEW directly modifies the data inside the loop or structure and passes the data to the indicator instead of creating a data copy. You can create constants for alternate cases instead of placing indicators inside the Case structure.

Monitoring Memory Usage

You can determine memory usage using several methods.

To view memory usage for the current VI, select **File»VI Properties** and select Memory Usage from the top pull-down menu. Notice that the information does not include memory use of subVIs. You can use the Profile Performance and Memory window to monitor the memory used by all VIs in memory and identify a subVI with performance issues. The VI Profile Performance and Memory window *keeps statistics* on the minimum, maximum, and average number of bytes and blocks used by each VI per run.



Note To monitor accurate VI run-time performance, save all VIs without [separate compiled code](#) in the current version of LabVIEW for the following reasons:

- Using the Undo feature creates temporary copies of objects and data, which can increase the reported memory requirements of a VI. Saving the VI purges the copies generated by Undo, resulting in accurate reports of memory information.
- Running unsaved VIs from previous versions of LabVIEW can increase reported memory requirements and decrease VI run-time performance.

Use the Show Buffer Allocations window to identify specific areas on the block diagram where LabVIEW allocates memory space in the form of a buffer.



Note The Show Buffer Allocations window is available only in the LabVIEW Full and Professional Development Systems.

Select **Tools»Profile»Show Buffer Allocations** to display the Show Buffer Allocations window. Place a checkmark next to the data type(s) you want to see buffers for and click the **Refresh** button. The black squares that appear on the block diagram indicate where LabVIEW creates buffers to allocate space for data.

The amount of memory LabVIEW allocates for each buffer is the same size as the largest [data type](#). When you run a VI, LabVIEW might or might not use those buffers to store data. You cannot know whether LabVIEW makes a copy of the data because LabVIEW makes that decision at run time or the VI might depend on dynamic data.

If the VI needs memory for a buffer allocation, LabVIEW makes a data copy for the buffer. If LabVIEW is not sure whether the buffer needs a data copy, LabVIEW sometimes still makes a copy for the buffer.



Note The Show Buffer Allocations window identifies only execution buffers with the use of the black squares on the block diagram. Using this window does not identify operation buffers on the front panel.

Once you know where LabVIEW creates buffers, you might be able to [edit the VI to reduce the amount of memory](#) LabVIEW requires to run the VI. LabVIEW must allocate memory to run the VI, so you cannot eliminate all the buffers.

If you make a change to a VI that requires LabVIEW to recompile the VI, the black squares disappear because the buffer information might no longer be correct. Click the **Refresh** button on the Show Buffer Allocations window to recompile the VI and display the black squares. When you close the **Show Buffer Allocations** window, the black squares disappear.



Note You also can use the LabVIEW Desktop Execution Trace Toolkit to monitor thread usage, memory leaks, and other aspects of LabVIEW programming.

Select **Help»About LabVIEW** to view a statistic that summarizes the total amount of memory used by your application. This includes memory for VIs as well as memory the application uses. You can check this amount before and after execution of a set of VIs to obtain a rough idea of how much memory the VIs are using.

Rules for Better Memory Usage

The main point of the previous section is that the compiler attempts to reuse memory intelligently. The rules for when the compiler can and cannot reuse memory are complex. In practice, the following rules can help you to create VIs that use memory efficiently:

- Breaking a VI into subVIs generally does not hurt your memory usage. In many cases, memory usage improves because the execution system can reclaim subVI data memory when the subVI is not executing.
- Do not worry too much about copies of scalar values; it takes a lot of scalars to negatively affect memory usage.
- Do not overuse global and local variables when working with arrays or strings. Reading a global or local variable causes LabVIEW to generate a copy of the data.
- Do not display large arrays and strings on open front panels unless it is necessary. Controls and indicators on open front panels retain a copy of the data they display.



Tip If a chart indicator is necessary, know that the chart history will retain a copy of the data it displays. This continues until the chart history is filled, then LabVIEW stops taking more memory. LabVIEW does not automatically clear the chart history when the VI restarts, so it helps to clear the chart history throughout the execution of the program. To do this, write empty arrays to the History Data attribute node for the chart.

- Use the Defer Panel Updates property. When you set this property to TRUE, the front panel indicator values do not change, even if you change the value of a control. Your operating system does not have to use any memory to populate the indicator(s) with the new value(s).



Note LabVIEW does not typically open a subVI front panel when calling the subVI.

- If the front panel of a subVI is not going to be displayed, do not leave unused Property Nodes on the subVI. Property Nodes cause the front panel of a subVI to remain in memory, which can cause unnecessary memory usage.
- When designing block diagrams, watch for places where the size of an input is different from the size of an output. For example, if you see places where you are frequently increasing the size of an array or string using the Build Array or Concatenate Strings functions, you are generating copies of data.
- Use **consistent data types** for arrays and watch for coercion dots when passing data to subVIs and functions. When you change data types, the execution system makes a copy of the data.

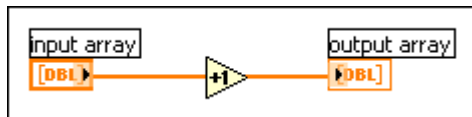
- Do not use complicated, hierarchical data types such as clusters or arrays of clusters containing large arrays or strings. You might end up using more memory. If possible, [use more efficient data types](#).
- Do not use transparent and overlapped front panel objects unless they are necessary. Such objects might use more memory.

Refer to the [LabVIEW Style Checklist](#) for more specific ideas on how to optimize the performance of a VI.

Memory Issues in Front Panels

When a front panel is open, controls and indicators keep their own, private copy of the data they display.

The following block diagram shows the increment function, with the addition of front panel controls and indicators.



When you run the VI, the data of the front panel control is passed to the block diagram. The increment function reuses the input buffer. The indicator then makes a copy of the data for display purposes. Thus, there are three copies of the buffer.

This data protection of the front panel control prevents the case in which you enter data into a control, run the associated VI, and see the data change in the control as it is passed in-place to subsequent nodes. Likewise, data is protected in the case of indicators so that they can reliably display the previous contents until they receive new data.

With subVIs, you can use controls and indicators as inputs and outputs. The execution system makes a copy of the control and indicator data of the subVI in the following conditions:

- The front panel is in memory. This can occur for any of the following reasons:
 - The front panel is open.
 - The VI has been changed but not saved (all components of the VI remain in memory until the VI is saved).
- The panel uses data printing.
- The block diagram uses Property Nodes.
- The VI uses local variables.
- The panel uses data logging.

For a Property Node to be able to read the chart history in subVIs with closed panels, the control or indicator needs to display the data passed to it. Because there are numerous other attributes like this, the execution system keeps subVI panels in memory if the subVI uses Property Nodes.

If a front panel uses front panel data logging or data printing, controls and indicators maintain copies of their data. In addition, panels are kept in memory for data printing so the panel can be printed.

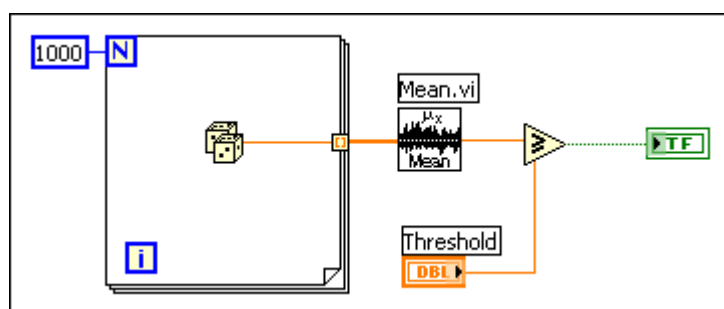
If you set a subVI to display its front panel when called using the VI Properties dialog box or the SubVI Node Setup dialog box, the front panel loads into memory when you call the subVI. If you set the **Close afterwards if originally closed** item, LabVIEW removes the front panel from memory when the subVI finishes execution.

SubVIs Can Reuse Data Memory

In general, a subVI can use data buffers from its caller as easily as if the block diagrams of the subVI were duplicated at the top level. In most cases, you do not use more memory if you convert a section of your block diagram into a subVI. For VIs with special display requirements, as described in the previous section, there might be some additional memory usage for front panels and controls.

Understanding When Memory Is Deallocated

Consider the following block diagram.



After the Mean VI executes, the array of data is no longer needed. Because determining when data is no longer needed can become very complicated in larger block diagrams, the execution does not deallocate the data buffers of a particular VI during its execution.

On macOS, if the execution system is low on memory, it deallocates data buffers used by any VI not currently executing. The execution system does not deallocate memory used by front panel controls, indicators, global variables, or uninitialized shift registers.

Now consider the same VI as a subVI of a larger VI. The array of data is created and used only in the subVI. On macOS, if the subVI is not executing and the system is low on memory, it might deallocate the data in the subVI. This is a case in which using subVIs can save on memory usage.

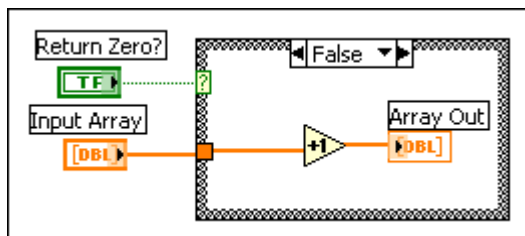
On Windows and Linux platforms, data buffers are not normally deallocated unless a VI is closed and removed from memory. Memory is allocated from the operating system as needed, and virtual memory works well on these platforms. Due to fragmentation, the application might appear to use more memory than it really does. As memory is allocated and freed, the application tries to consolidate memory usage so it can return unused blocks to the operating system.

You can use the Request Deallocation function to deallocate unused memory after the VI that contains this function runs. This function is only used for advanced performance optimizations. Deallocating unused memory can improve performance in some cases. However, aggressively deallocating memory can cause LabVIEW to reallocate space repeatedly rather than reusing an allocation. Use this function if your VI allocates a large amount of data but never reuses that allocation. When a top-level VI calls a subVI, LabVIEW allocates a data space of memory in which that subVI runs. When the subVI finishes running, LabVIEW does not deallocate the data space until the top-level VI finishes running or until the entire application stops, which can result in out-of-memory conditions and degradation of performance. Place the Request Deallocation function in the subVI you want to deallocate memory for. When you set the **flag** Boolean input to TRUE, LabVIEW reduces memory usage by deallocating the data space for the subVI.

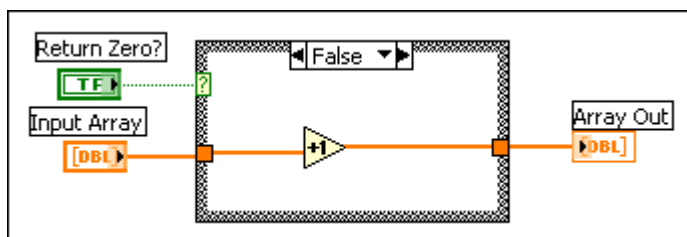
Determining When Outputs Can Reuse Input Buffers

If an output is the same size and data type as an input, and the input is not required elsewhere, the output can reuse the input buffer. As mentioned previously, in some cases even when an input is used elsewhere, the compiler and the execution system can order code execution in such a way that it can reuse the input for an output buffer. However, the rules for this are complex. Do not depend on them.

You can use the Show Buffer Allocations window to see if an output buffer reuses the input buffer. In the following block diagram, placing an indicator in each case of a Case structure breaks the flow of data because LabVIEW creates a copy of the data for each indicator. LabVIEW does not use the buffer it created for the input array but instead creates a copy of the data for the output array.



If you move the indicator outside the Case structure, the output buffer reuses the input buffer because LabVIEW does not need to make a copy of the data the indicator displays. Because LabVIEW does not require the value of the input array later in the VI, the increment function can directly modify the input array and pass it to the output array. In this situation, LabVIEW does not need to copy the data, so a buffer does not appear on the output array, as shown in the following block diagram.



Optimizing Memory Usage by Matching Data Types

If you use consistent data types, LabVIEW can reuse the memory buffers allocated for an input when generating an output. This can improve the memory usage and execution time of the VI. However, if an input has a different data type from an output, the output cannot reuse the memory LabVIEW allocates for that input.

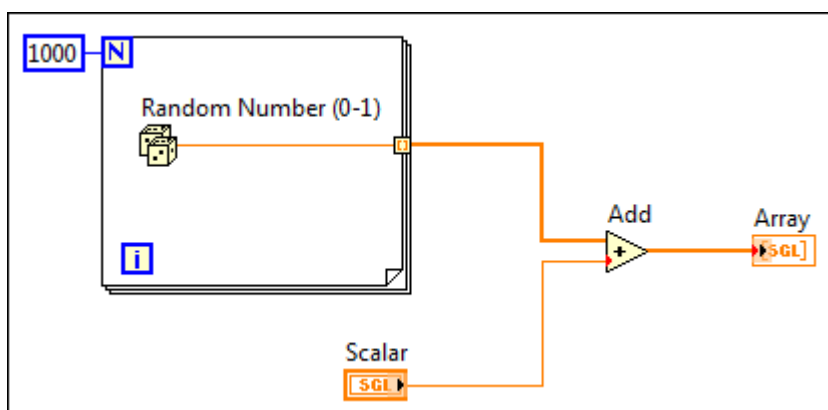
For example, if you add a 32-bit integer to a 16-bit integer, LabVIEW coerces the 16-bit integer to a 32-bit integer. The compiler creates a new buffer for the coerced data, and LabVIEW places a *coercion dot* on the Add function. In this example, LabVIEW can use the 32-bit integer input for the output buffer, assuming the input meets all the other requirements. However, because LabVIEW coerces the 16-bit integer, LabVIEW cannot reuse the memory allocated for that input.

To minimize memory usage, use consistent data types wherever possible. Using consistent data types produces fewer copies of data that result when LabVIEW coerces data to a larger size. In some applications, you also may consider using smaller data types to minimize data usage. For example, you may consider using four-byte, single-precision numbers instead of eight-byte, double-precision numbers. However, to avoid unnecessary coercions, match the data types of objects wired to a subVI with the data type the subVI expects.

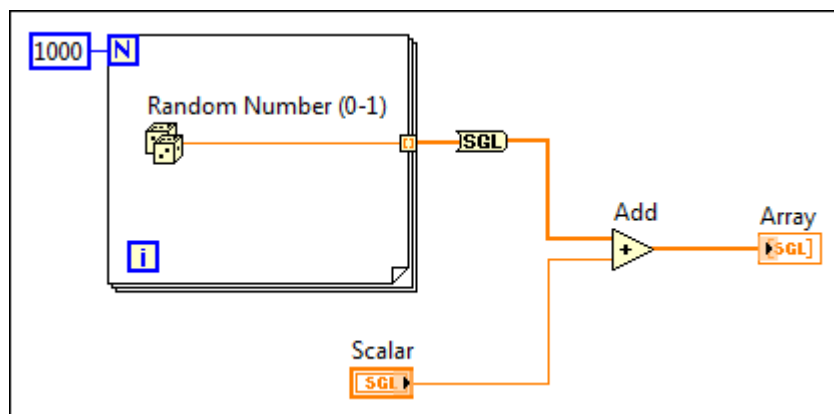
Optimizing Memory Usage When Generating Data of a Specific Type

When you need to generate data of a specific data type, you may have to convert data types on your block diagram. However, you can optimize memory usage by converting data types using Conversion functions before LabVIEW creates large arrays.

In the following example, the output data needs to be of the single-precision, floating-point data type to match the data type of an input of another VI. LabVIEW creates an array of 1,000 random values and adds the array to a scalar value. LabVIEW places a coercion dot on the Add function because LabVIEW coerces the single-precision, floating-point scalar data type to match the random double-precision, floating-point numbers.

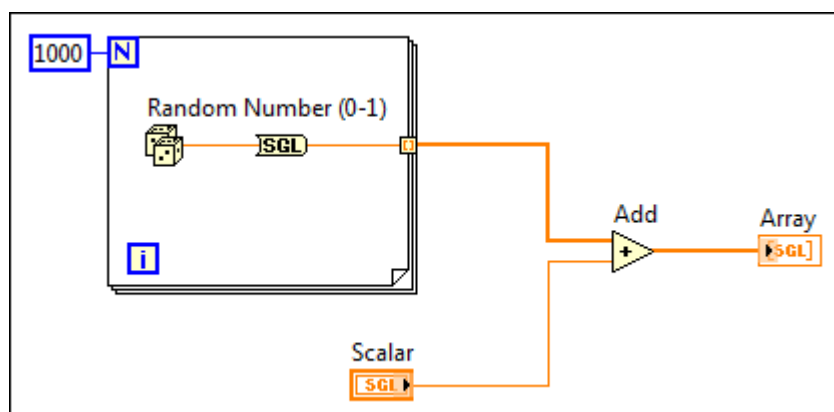


The following block diagram shows an attempt to correct this problem by converting the array of double-precision, floating-point numbers using a To Single Precision Float function.



Because the function still converts the data type of the large array after LabVIEW creates the array, the VI uses the same amount of memory as the example with the coercion dots.

The following block diagram illustrates how to optimize memory usage and execution speed by converting the random number before LabVIEW creates the array.



When you cannot avoid conversions, you can use a Conversion function before LabVIEW creates a large array to avoid converting the large data buffer allocated for the array from one data type to another. Converting data before LabVIEW creates a large array optimizes the memory usage of a VI.

Avoid Constantly Resizing Data

If the size of an output is different from the size of an input, the output does not reuse the input data buffer. This is the case for functions such as Build Array, Concatenate Strings, and Array Subset, which increase or decrease the size of an array or string. When working with arrays and strings, avoid constantly using these functions, because your program uses more data memory and executes more slowly because it is constantly copying data.

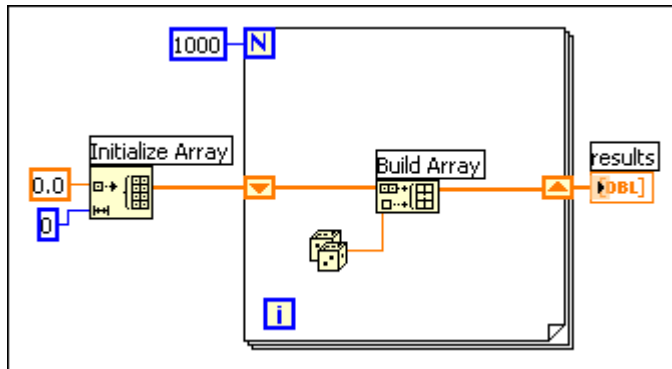
Example 1: Building Arrays

Consider the following block diagram, which is used to create an array of data. This block diagram creates an array in a loop by constantly calling Build Array to concatenate a new element. The input array is reused by Build Array. The VI continually resizes the buffer in

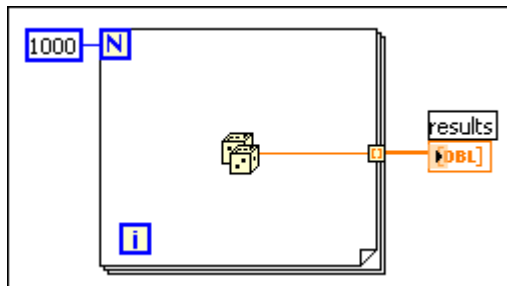
each iteration to make room for the new array and appends the new element. The resulting execution speed is slow, especially if the loop is executed many times.



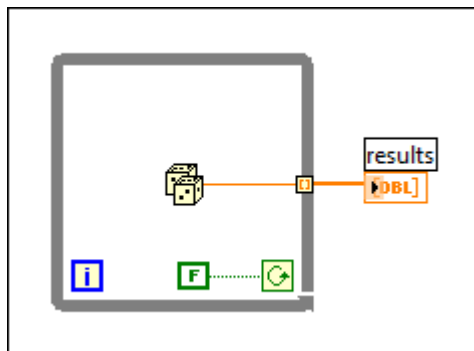
Note When you manipulate arrays, clusters, waveforms, and variants, you can use the In Place Element structure to improve memory usage in VIs.



If you want to add a value to the array with every iteration of the loop, you can see the best performance by using auto-indexing on the edge of a loop. With For Loops, the VI can predetermine the size of the array (based on the value wired to N), and resize the buffer only once.



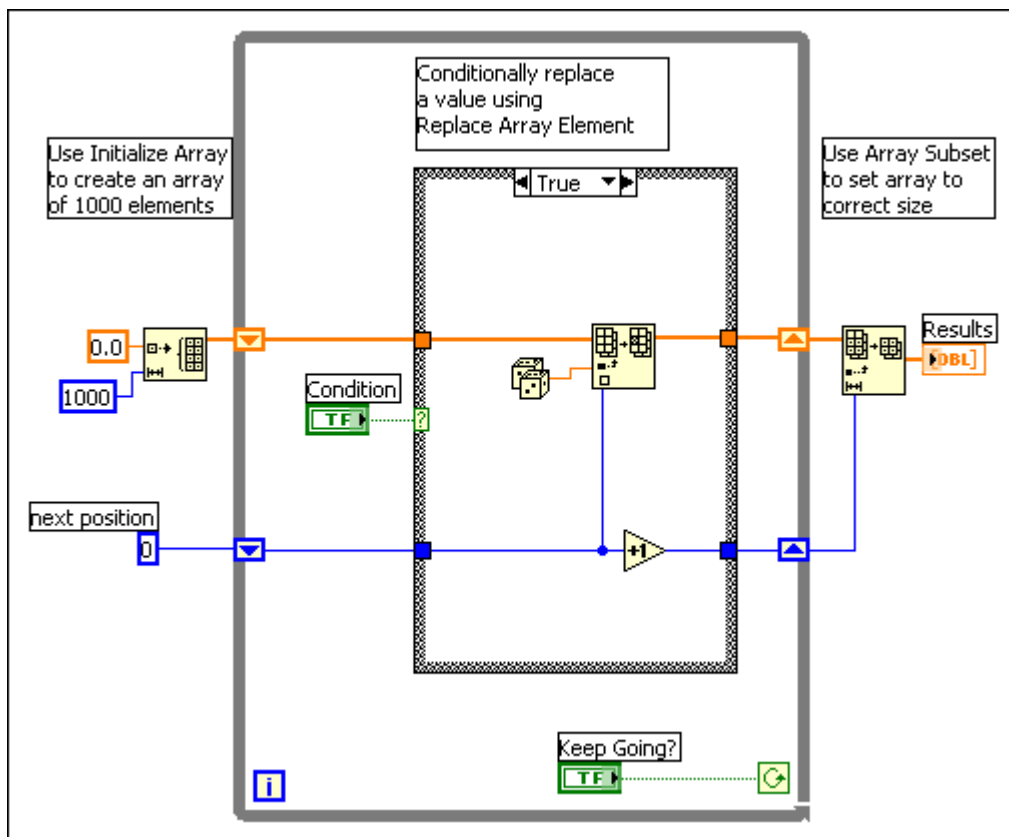
With While Loops, auto-indexing is not quite as efficient, because the end size of the array is not known. However, While Loop auto-indexing avoids resizing the output array with every iteration by increasing the output array size in large increments. When the loop is finished, the output array is resized to the correct size. The performance of While Loop auto-indexing is nearly identical to For Loop auto-indexing.



Auto-indexing assumes you are going to add a value to the resulting array with each iteration of the loop. If you must conditionally add values to an array but can determine an upper limit on the array size, you might consider preallocating the array and using Replace Array Subset to fill the array.

When you finish filling the array values, you can resize the array to the correct size. The array is created only once, and Replace Array Subset can reuse the input buffer for the output buffer. The performance of this is very similar to the performance of loops using auto-indexing. If you use this technique, be careful the array in which you are replacing values is large enough to hold the resulting data, because Replace Array Subset does not resize arrays for you.

An example of this process is shown in the following block diagram.



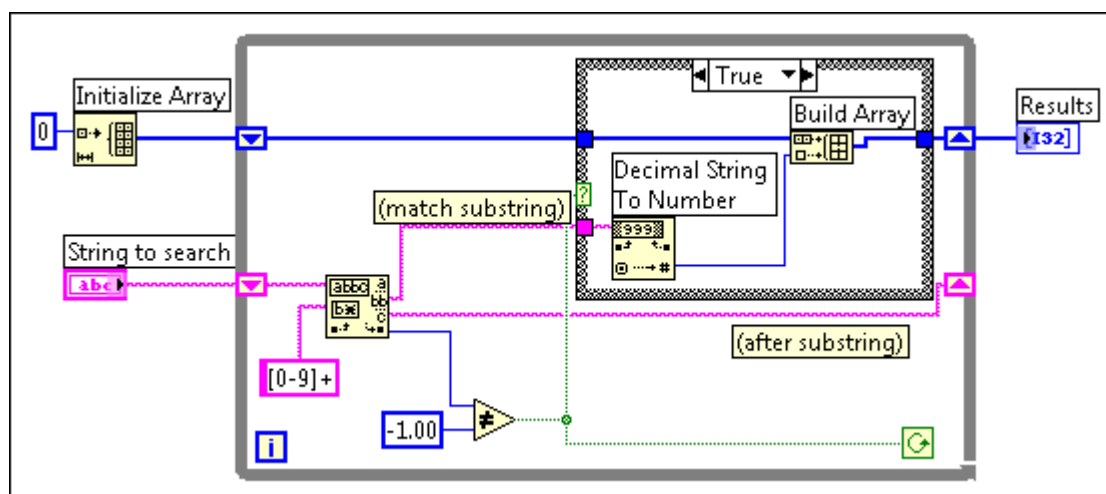
Example 2: Searching through Strings

You can use the Match Pattern function to search a string for a pattern. Depending on how you use it, you might slow down performance by unnecessarily creating string data buffers.

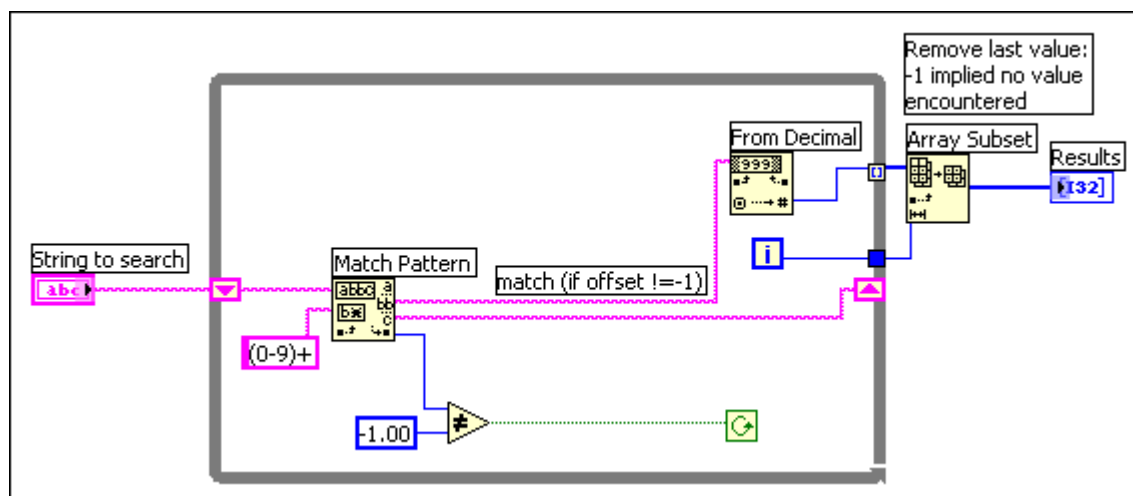
Assuming you want to match an integer in a string, you can use `[0-9]+` as the regular expression input to this function. To create an array of all integers in a string, use a loop and call Match Regular Expression repeatedly until the offset value returned is -1.

The following block diagram is one method for scanning for all occurrences of integers in a string. It creates an empty array and then searches the remaining string for the numeric pattern with each iteration of the loop. If the pattern is found (offset is not -1, this block diagram uses Build Array to add the number to a resulting array of numbers. When there

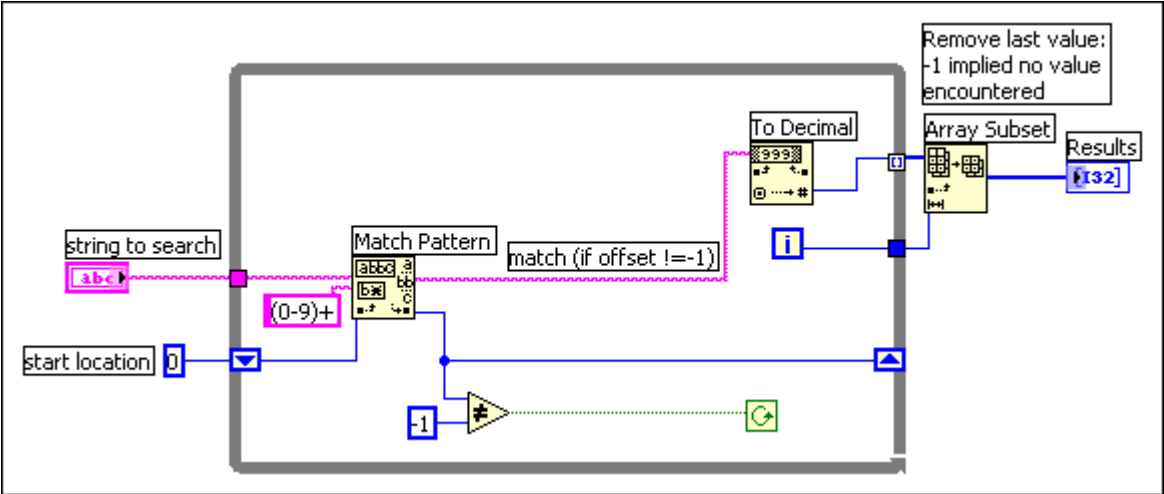
are no values left in the string, Match Regular Expression returns -1 and the block diagram completes execution.



One problem with this block diagram is that it uses Build Array in the loop to concatenate the new value to the previous value. Instead, you can use auto-indexing to accumulate values on the edge of the loop. Notice you end up seeing an extra unwanted value in the array from the last iteration of the loop where Match Regular Expression fails to find a match. A solution is to use Array Subset to remove the extra unwanted value. This is shown in the following block diagram.



The other problem with this block diagram is that you create an unnecessary copy of the remaining string every time through the loop. Match Regular Expression has an input you can use to indicate where to start searching. If you remember the offset from the previous iteration, you can use this number to indicate where to start searching on the next iteration. This technique is shown in the following block diagram.



Developing Efficient Data Structures

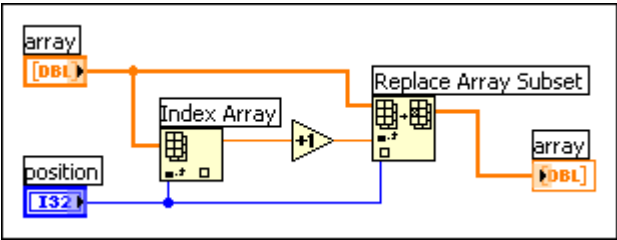
One of the points made in the previous example is that hierarchical data structures, such as clusters or arrays of clusters containing large arrays or strings, cannot be manipulated efficiently. This section explains why this is so and describes strategies for choosing more efficient data types.

The problem with complicated data structures is that it is difficult to access and change elements within a data structure without causing copies of the elements you are accessing to be generated. If these elements are large, as in the case where the element itself is an array or string, these extra copies use more memory and the time it takes to copy the memory.

You can generally manipulate scalar data types very efficiently. Likewise, you can efficiently manipulate small strings and arrays where the element is a scalar. In the case of an array of scalars, the following code shows what you do to increment a value in an array.



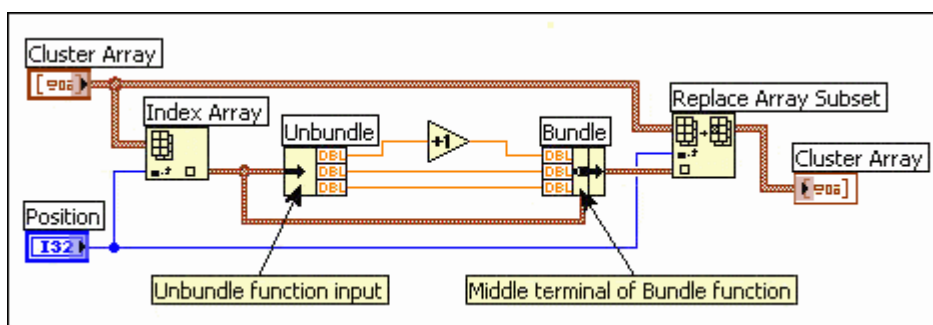
Note When you manipulate arrays, clusters, waveforms, and variants, you can use the In Place Element structure to improve memory usage in VIs. Many LabVIEW operations require LabVIEW to copy and maintain data values in memory, thereby decreasing execution speed and increasing memory usage. The In Place Element structure performs common LabVIEW operations without LabVIEW making multiple copies of the data values in memory. Instead, the In Place Element structure operates on data elements in the same memory location and returns those elements to the same location in the array, cluster, variant, or waveform. Because LabVIEW returns the data elements to the same location in memory, the LabVIEW compiler does not have to make extra copies of the data in memory.



This is quite efficient because it is not necessary to generate extra copies of the overall array. Also, the element produced by the Index Array function is a scalar, which can be created and manipulated efficiently.

The same is true of an array of clusters, assuming the cluster contains only scalars. In the following block diagram, manipulation of elements becomes a little more complicated, because you must use Unbundle and Bundle. However, because the cluster is probably small (scalars use very little memory), there is no significant overhead involved in accessing the cluster elements and replacing the elements back into the original cluster.

The following block diagram shows the efficient pattern for unbundling, operating, and bundling again. The wire from the data source should have only two destinations – the Unbundle function input, and the middle terminal on the Bundle function. LabVIEW recognizes this pattern and is able to generate better-performing code.



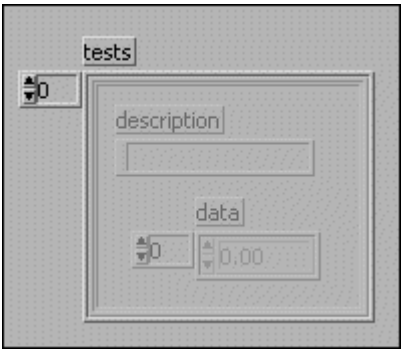
If you have an array of clusters where each cluster contains large sub-arrays or strings, indexing and changing the values of elements in the cluster can be more expensive in terms of memory and speed.

When you index an element in the overall array, a copy of that element is made. Thus, a copy of the cluster and its corresponding large subarray or string is made. Because strings and arrays are of variable size, the copy process can involve memory allocation calls to make a string or subarray of the appropriate size, in addition to the overhead actually copying the data of a string or subarray. This might not be significant if you only plan to do it a few times. However, if your application centers around accessing this data structure frequently, the memory and execution overhead might add up quickly.

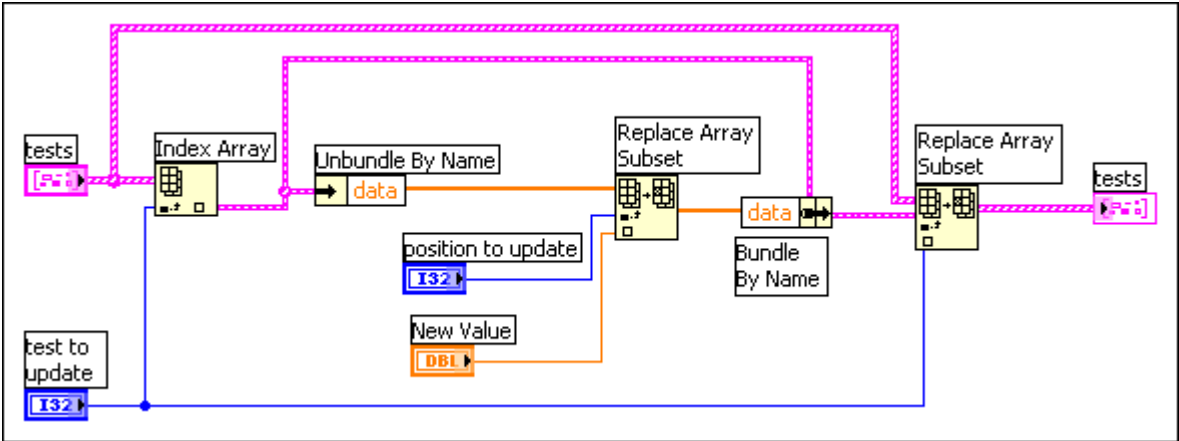
The solution is to look at alternative representations for your data. The following three case studies present three different applications, along with suggestions for the best data structures in each case.

Case Study 1: Avoiding Complicated Data Types

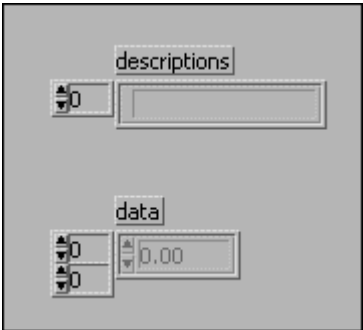
Consider an application in which you want to record the results of several tests. In the results, you want a string describing the test and an array of the test results. One data type you might consider using to store this information is shown in the following front panel.



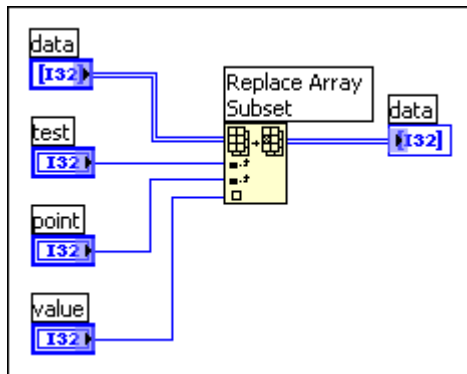
To change an element in the array, you must index an element of the overall array. Now, for that cluster you must unbundle the elements to reach the array. You then replace an element of the array and store the resulting array in the cluster. Finally, you store the resulting cluster into the original array. An example of this is shown in the following block diagram.



Each level of unbundling/indexing might result in a copy of that data being generated. Notice a copy is not necessarily generated. Copying data is costly in terms of both time and memory. The solution is to try to make the data structures as flat as possible. For example, in this case study break the data structure into two arrays. The first array is the array of strings. The second array is a 2D array, where each row is the results of a given test. This result is shown in the following front panel.



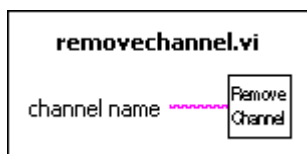
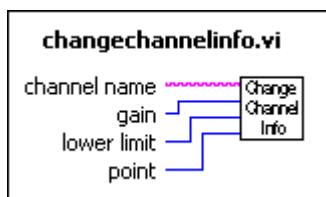
Given this data structure, you can replace an array element directly using the Replace Array Subset function, as shown in the following block diagram.



Case Study 2: Global Table of Mixed Data Types

Here is another application in which you want to maintain a table of information. In this application, you want the data to be globally accessible. This table might contain settings for an instrument, including gain, lower and upper voltage limits, and a name used to refer to the channel.

To make the data accessible throughout your application, you might consider creating a set of subVIs to access the data in the table, such as the following subVIs, the Change Channel Info VI and the Remove Channel Info VI.



The following sections present three different implementations for these VIs.

Obvious Implementation

With this set of functions, there are several data structures to consider for the underlying table. First, you might use a global variable containing an array of clusters, where each cluster contains the gain, lower limit, upper limit, and the channel name.

As described in the previous section, this data structure is difficult to manipulate efficiently, because generally you must go through several levels of indexing and unbundling to access your data. Also, because the data structure is a conglomeration of several pieces of information, you cannot use the Search 1D Array function to search for a channel. You can use Search 1D Array to search for a specific cluster in an array of clusters, but you cannot use it to search for elements that match on a single cluster element.

Alternative Implementation 1

As with the previous case study, choose to keep the data in two separate arrays. One contains the channel names. The other contains the channel data. The index of a given channel name in the array of names is used to find the corresponding channel data in the other array.

Notice that because the array of strings is separate from the data, you can use the Search 1D Array function to search for a channel.

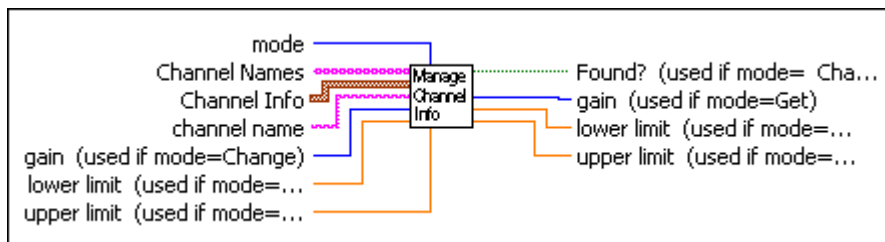
In practice, if you are creating an array of 1,000 channels using the Change Channel Info VI, this implementation is roughly twice as fast as the previous version. This change is not very significant because there is other overhead affecting performance.

When you read from a global variable, a copy of the data of the global variable is generated. Thus, a complete copy of the data of the array is being generated each time you access an element. The next method shows an even more efficient method that avoids this overhead.

Alternative Implementation 2

There is an alternative method for storing global data, and that is to use an uninitialized shift register. Essentially, if you do not wire an initial value, a shift register remembers its value from call to call.

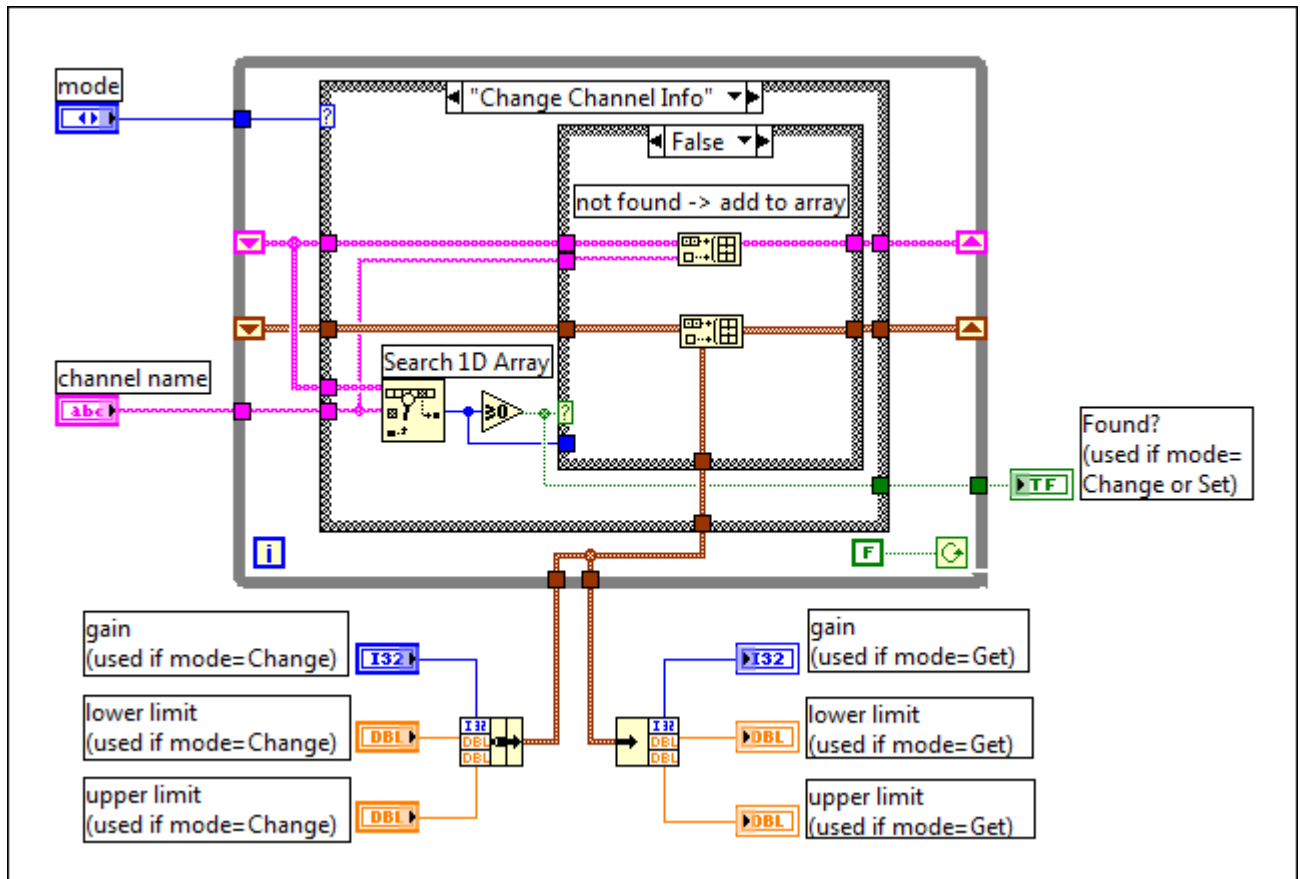
The LabVIEW compiler handles access to shift registers efficiently. Reading the value of a shift register does not necessarily generate a copy of the data. In fact, you can index an array stored in a shift register and even change and update its value without generating extra copies of the overall array. The problem with a shift register is only the VI that contains the shift register can access the shift register data. On the other hand, the shift register has the advantage of modularity.



You can make a single subVI with a **mode** input that specifies whether you want to read, change, or remove a channel, or whether you want to zero out the data for all channels.

The subVI contains a While Loop with two shift registers—the channel data, and one for the channel names. Neither of these shift registers is initialized. Then, inside the While Loop you place a Case structure connected to the mode input. Depending on the value of the mode, you might read and possibly change the data in the shift register.

Following is an outline of a subVI with an interface that handles these three different modes. Only the Change Channel Info code is shown.

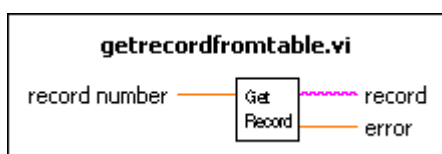
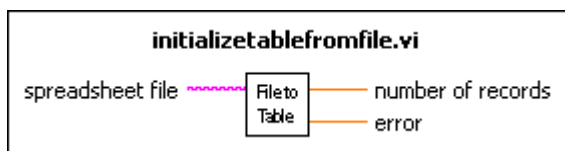


For 1,000 elements, this implementation is twice as fast as the previous implementation, and four times faster than the original implementation.

Case Study 3: A Static Global Table of Strings

The previous case study looked at an application in which the table contained mixed data types, and the table might change frequently. In many applications, you have a table of information that is fairly static once created. The table might be read from a spreadsheet file. Once read into memory, you mainly use it to look up information.

In this case, your implementation might consist of the following two functions, Initialize Table From File and Get Record From Table.



One way to implement the table is to use a two-dimensional array of strings. Notice the compiler stores each string in an array of strings in a separate block of memory. If there are a large number of strings (for example, more than 5,000 strings), you might put a load on the memory manager. This load can cause a noticeable loss in performance as the number of individual objects increases.

An alternative method for storing a large table is to read the table in as a single string. Then build a separate array containing the offsets of each record in the string. This changes the organization so that instead of having potentially thousands of relatively small blocks of memory, you instead have one large block of memory (the string) and a separate smaller block of memory (the array of offsets).

This method might be more complicated to implement, but it can be much faster for large tables.

VI Code Complexity

VI code complexity is a collective term that refers to several characteristics of a VI, including the number, type, and organization of front panel and block diagram nodes. LabVIEW presents code complexity in the following related but distinct ways:

Complexity Measure	When to Use	Explanation
VI Metrics window	• Measuring and estimating project development efforts based on quantifiable metrics • Identifying specific ways to reduce the complexity of a VI, thereby obtaining the following benefits for the VI: ◦ Improved compilation speed ◦ Decreased memory usage during compilation ◦ Improved readability and maintainability of the block diagram	This window reports several aspects of code complexity, such as the number of various kinds of nodes and the maximum depth of nested structures on the block diagram.

Complexity Measure	When to Use	Explanation
Compiled Code Complexity	<i>Adjusting the threshold at which LabVIEW limits compiler optimizations to improve editor responsiveness</i>	This value provides an overall representation of code complexity for a VI. The compiler uses this value to determine whether a VI requires too much memory during compilation to receive full optimizations. LabVIEW calculates this value based on internal compilation details related to the numbers provided by the VI Metrics window. However, these details are subject to change between versions of LabVIEW. Therefore, unlike the concrete statistics in the VI Metrics window, this code complexity value uses units that do not directly represent the implementation details of the compilation process. Instead, the units serve only as a point of reference by which to compare the code complexity of a VI to the environmental compiler optimizations threshold that LabVIEW uses to limit optimizations.

Reentrancy: Allowing Simultaneous Calls to the Same SubVI

By default, LabVIEW configures VIs for non-reentrant execution. In non-reentrant execution, when a VI is called as a subVI simultaneously by other VIs, only one of those calls can enter and execute the subVI at a time. LabVIEW ensures that callers of the non-reentrant subVI take turns using the subVI. A non-reentrant VI has only a single data space to store the parameters of its callers, any temporary values computed during execution, and any state that the non-reentrant VI maintains between calls. However, if you configure a VI for reentrant execution, LabVIEW can enter and execute the reentrant VI simultaneously from all callers. In reentrant execution, LabVIEW allocates multiple instances, or clones, of the data space so that each call site, or usage of the subVI on the block diagram of the caller, can execute simultaneously, in parallel, such that each call site uses a separate clone.



NOTE

(FPGA Module) FPGA VIs are reentrant by default.

Types of Reentrant Execution

LabVIEW provides the following types of reentrant VIs:

- **Non-reentrant execution** —LabVIEW allocates a single data space for use by all instances of the subVI.
- **Shared clone reentrant execution** —LabVIEW allocates a pool of clones, initially populated with one clone, to share amongst callers. When a caller calls a shared clone VI, it pulls a clone from the pool and returns the clone when the call finishes executing. If the pool is empty when a caller calls a shared clone VI, LabVIEW creates a new clone for that call and places the clone in the pool when execution finishes, thereby expanding the clone pool.
- **Preallocated clone reentrant execution** —LabVIEW allocates a separate, non-shareable, clone for each use of the subVI.

To configure the reentrancy settings for a VI, refer to the Execution page of the VI Properties dialog box.

Choosing Which Type of Reentrant Execution to Use

Usage Considerations	Non-reentrant	Shared Clone Reentrancy	Preallocated Clone Reentrancy
Ability to maintain state, such as in uninitialized shift registers	Maintains a single instance of the data space and uses its state across all call sites.	Does not maintain state—Each call site pulls the data space of a clone randomly from the pool of clones. Call sites may end up sharing state.	Maintains state for each call site—Each call site has its own separate and specific clone.
Determinism for multiple, simultaneous calling VIs	Call sites might wait in line behind other simultaneous callers—The time required for an instance of a subVI to begin executing varies with how many other instances are scheduled to execute before it.	Possible waiting—Call sites may need to wait for LabVIEW to create a new clone if the pool of clones is empty. When the pool is not empty, pulling a clone and returning the clone to the pool can introduce non-deterministic delays.	No waiting—LabVIEW allocates a clone for each call site prior to execution. This allows deterministic memory usage and execution speed.
Call overhead when no simultaneous calling VIs exist	Medium—LabVIEW must check for simultaneous callers.	Highest—LabVIEW must atomically pull clones from and return clones to the pool of clones. Calls may need to wait for LabVIEW to add a clone to the pool.	Lowest—LabVIEW already preallocates a clone to each call site.
Memory usage	Lowest—LabVIEW allocates only a single data space for the subVI.	Medium—LabVIEW must allocate clones only for the maximum number of instances running simultaneously.	Highest—LabVIEW must allocate a clone for each instance of the subVI.



Note (FPGA Module) Shared clone reentrant VIs and preallocated clone reentrant VIs behave the same when used in FPGA VIs.

Asynchronously Calling Multiple Instances of a Reentrant VI

Refer to the [Asynchronously Calling Multiple Instances of a VI for Parallel Execution](#) topic for information about asynchronously calling reentrant VIs.

Viewing the Front Panel of the Clone of a Reentrant VI

Each clone has an individual front panel. To view the front panel while the clone is not running, double-click the instance of the subVI whose panel you want to view. You can also set the reentrant VI to [open the front panel during execution](#). The title bar of the subVI contains (c1one) to indicate that it is a clone rather than the original VI. To access the original VI from the clone, select **View»Browse Relationships»Reentrant Items**. You can also use the Front Panel:Open method within the subVI to programmatically open the front panel of each clone when it runs. The title bar of the subVI contains (c1one) to indicate that it is a clone of the source VI.

You can interact with the front panel of a reentrant VI similarly to the front panel of a non-reentrant VI, particularly in the following ways:

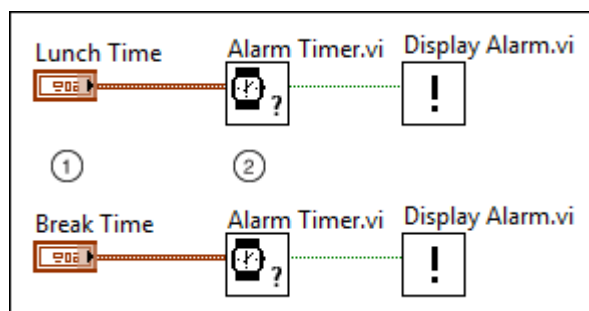
- Configuring an Event structure to handle events for front panel objects
- Using the front panel as a [subpanel](#)
- Using properties and methods to [configure controls and indicators](#)

Examples of Reentrant Execution

The following two sections describe examples of reentrant VIs that wait and do not share data.

Ensuring Simultaneous Execution of SubVI Instances

The following block diagram alerts the user when both his **Lunch Time** and his **Break Time** arrive. However, the reentrancy setting of the Alarm Timer VI is critical for producing accurate results.



①	The user specifies a Lunch Time and a Break Time . For the purposes of this example, assume Lunch Time is 12:00pm and Break Time is 10:00am.
②	The Alarm Timer VI waits until the specified input time. Because the instances of the Alarm Timer VI appear in parallel parts of the diagram, it is impossible to predict which instance executes first. For the purposes of this example, assume the Lunch Time timer executes first.

If the Alarm Timer VI is configured for non-reentrant execution, the **Break Time** timer cannot begin running until 12:00pm because it must wait for the **Lunch Time** timer to finish executing. This delay prevents the **Break Time** timer from triggering an alarm at 10:00am. Therefore, to ensure that both instances of the subVI can execute simultaneously and trigger their alarms at the appropriate time, you must configure the Alarm Timer VI to use one of the types of reentrant execution.

Storing Data between Multiple Calls to Each Instance of a SubVI

To maintain state between calls to a subVI from a specific call site, you must use preallocated clone reentrant execution. For example, you create a VI, ExpAvg, that calculates a running exponential average of the last four data points that have been passed to the VI. LabVIEW can maintain the most recent four data points as state in an uninitialized shift register in a single iteration While Loop.

Another VI uses the ExpAvg subVI to calculate the running average of two data acquisition channels. The VI monitors the voltages at two points in a process and displays the exponential running average on a strip chart. The block diagram of the VI contains two instances of the ExpAvg subVI. The calls alternate — one for Channel 0, and one for Channel 1. Assume Channel 0 runs first. If the ExpAvg subVI is not reentrant, the call for Channel 1 uses the average computed by the call for Channel 0, and the call for Channel 0 uses the average computed by the call for Channel 1. By making the ExpAvg subVI reentrant, each call can run independently without sharing the data.

Overriding Reentrancy Settings for SubVIs in Timed Structures

(Real-Time, Windows) You can configure LabVIEW to either preallocate or share clones of a shared-clone reentrant VI within a *Timed Loop or Timed Sequence* structure. For example, if you place a shared-clone reentrant VI inside of a Timed Loop or Timed Sequence structure, you can set the structure to preallocate clones for instances of the reentrant VI you call within the structure. Instances of the reentrant VI that you call outside the structure continue to share clones. To set the allocation of a VI within a Timed Loop or a Timed Sequence structure, right-click the structure, select **Shared Clone Allocation**, and select one of the following:

- **Automatic** —LabVIEW decides whether to set the structure as **Preallocate** or **Allocate As Needed** based on the context of the structure. LabVIEW will append the shortcut menu item with text that indicates the setting LabVIEW chooses. For example, if LabVIEW chooses **Preallocate**, the menu item becomes **Automatic (Preallocate)**.

- **Preallocate** —LabVIEW creates a clone VI for each call to the reentrant VI within the structure. The Timed Loop or Timed Sequence structure displays an  icon to indicate that LabVIEW is preallocating clones for any reentrant VI in the structure.
- **Allocate As Needed** —LabVIEW does not create a clone VI until you make a call to the reentrant VI within the structure. The Timed Loop or Timed Sequence structure displays an  icon to indicate that LabVIEW is sharing clones for any reentrant VI in the structure.

Debugging Reentrant VIs

To allow debugging on a reentrant VI, select **File»VI Properties** to display the VI Properties dialog box, select **Execution** from the pull-down menu, and place a checkmark in the **Allow debugging** checkbox.

To debug a specific instance of a VI with preallocated clones, open that instance of the subVI. Although you cannot edit the clone VI, you can use the block diagram of the clone for debugging purposes. For example, you can set *breakpoints*, use *probes*, enable *execution highlighting*, and *single-step* through execution.

To debug an instance of a VI that shares clones, apply breakpoints, probes, and execution highlighting within the source VI, not within the clone VI. Clone VIs do not maintain these settings across calls.



Note When you debug applications and shared libraries, you cannot debug reentrant panels that an Open VI Reference function creates. You also cannot debug reentrant panels that are entry points to LabVIEW-built shared libraries.

Using Matrices

Use the matrix data type instead of a 2D array to represent matrix data because the matrix data type stores rows or columns of real or complex scalar data for matrix operations, particularly some linear algebra operations. The Mathematics VIs that perform matrix operations accept the matrix data type and return matrix results, which enables subsequent polymorphic VIs and functions in the data flow to perform matrix-specific operations. If a Mathematics VI does not perform matrix operations but accepts a matrix data type, the VI automatically converts the matrix data type to a 2D array. If you wire a 2D array to a VI that performs matrix operations by default, the VI automatically converts the 2D array to a real or complex matrix, depending on the data type of the 2D array.

Most Numeric functions support the matrix data type and matrix operations. For example, you can use the Multiply function to multiply a matrix by another matrix or by a number. You can combine basic numeric data types and complex linear algebra functions to create numeric algorithms that perform accurate matrix operations.



NOTE

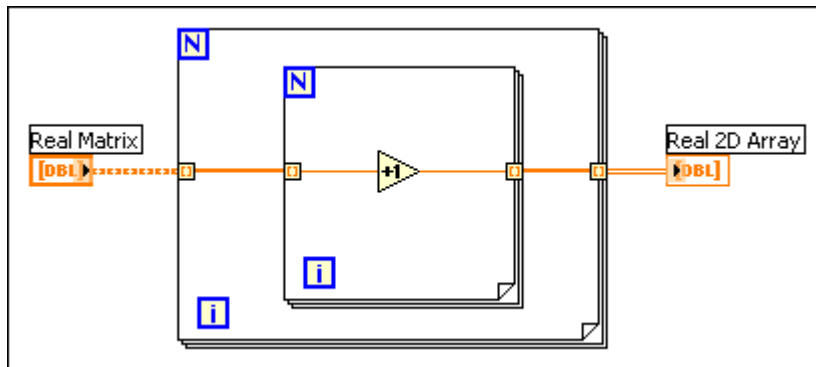
Coercion dots appear on VIs and functions when the VI or function converts data to or from a matrix or 2D array. This kind of data conversion does not affect performance because LabVIEW stores matrices the same way it stores 2D arrays.

A real matrix contains double-precision elements, and a complex matrix contains complex elements with double-precision components. A matrix can have only two dimensions. You cannot create an array of matrices. Use the Bundle function to combine two or more matrices and create a cluster. Matrices have the same restrictions as arrays.

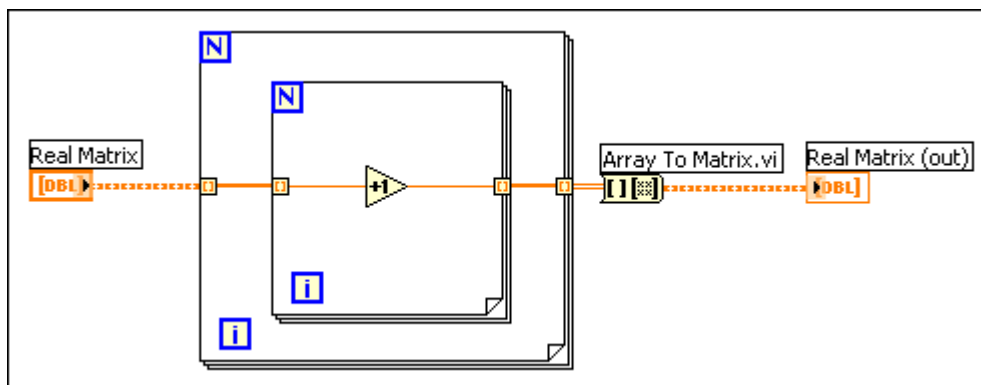
Use the Matrix functions to manipulate elements, rows, and columns in a matrix. The Matrix functions return matrix data types. For example, extract a row or column of a matrix with the Get Matrix Elements function to produce a matrix of one row or column instead of a 1D array of scalar values. Combine this matrix with other arrays with the Build Matrix function to produce a matrix rather than a 2D array of scalar values.

Any VI you build that reduces the dimension of a matrix requires you to convert the data to a 1D array or to a double-precision, floating-point or complex number. If you use the 1D array or number to rebuild a 2D structure, LabVIEW creates a 2D array rather than the original matrix.

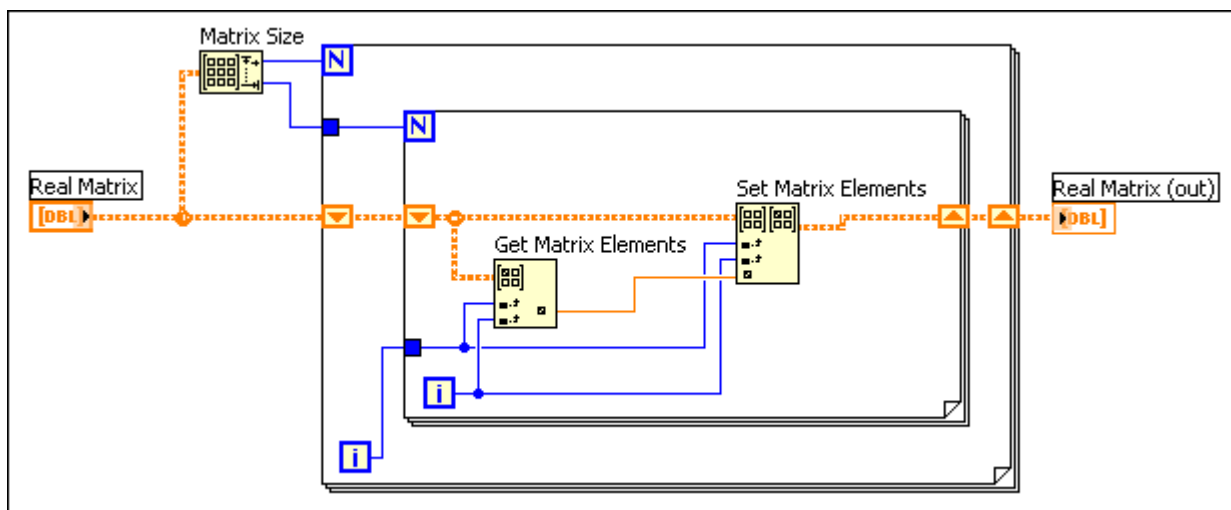
The following block diagram uses auto-indexing in a For Loop to extract data from a matrix and reconstruct the matrix in a 2D array.



With auto-indexing enabled, the For Loop treats a matrix as a 2D array. If, at some point in a VI, you need to use the matrix data type as an input, convert the 2D array to a matrix using the Array to Matrix function, as shown in the following block diagram.



To keep the data as a matrix data type, disable auto-indexing and use shift registers. For example, the following block diagram uses shift registers to maintain the two dimensions and data type of the matrix. The shift registers pass the matrix data type between iterations.



Many polymorphic functions that accept the matrix data type return the matrix data type even if the operation itself is an array-based operation. If a function or subVI in a block diagram converts a matrix data type to 2D array and subsequent operations in the data flow are array-based operations, allow the conversion to an array to occur, work with the array data, and then convert the array to a matrix with the Array to Matrix function when you need to, such as when you use the Linear Algebra VIs.



Note If a block diagram includes subVIs that accept matrix data types but return 2D arrays, you do not need to convert the resulting arrays back into matrices before you wire the arrays to polymorphic VIs or functions that accept the matrix data type by default. If you need to store the data as a matrix, use the Array To Matrix function to convert the data back to a matrix.

Creating Matrix Controls, Indicators, and Constants

By default, matrix controls and indicators show more than one element and display scrollbars for both dimensions. Because a matrix control is a type definition, the shortcut menus for matrix controls exclude the **Add Dimension** and **Remove Dimension** options and include type definition options. To create a matrix constant on the block diagram, select a matrix control on the Controls palette and place it on the block diagram or create a constant from any matrix terminal. You can use a matrix constant to store constant data or as a basis for comparison with another matrix.

On the block diagram, the matrix data type looks like a real 2D array or complex 2D array data type with a different wire pattern. The VIs and functions that accept the matrix data type automatically support matrix-specific operations when you wire a matrix data type as an input.

When you wire a matrix data type as an input to one of the following functions, a VI that includes subVIs that work with the matrix data type replaces the function:

- Equal?
- Not Equal?
- Absolute Value

- Add
- Multiply
- Square Root
- Subtract
- Exponential
- Natural Logarithm
- Power Of X
- Re/Im To Complex
- Complex To Re/Im
- Polar To Complex
- Complex To Polar

The resulting VI has the same icon but contains a matrix-specific algorithm. The node remains a VI if you disconnect the matrix from the input(s). Wire other data types as inputs to restore the original function. If you wire a data type to a function and that data type causes a basic math operation to fail, the function returns an empty matrix or NaN. For example, if you wire a matrix with a dimension of 2×3 to one input of the Add function, then wire a 3×2 matrix to the other input, the function returns an empty matrix.

Comparing Matrices

Use the Equal? and Not Equal? functions to compare matrices. These functions operate in Compare Aggregates mode for matrix inputs and follow the same rules as comparing arrays.

When a numeric function that accepts two or more inputs receives a matrix and a 2D array as inputs, in most cases, the function returns a matrix data type. The function performs the array operation if the output wired to the connector pane is not a 2D array. For example, if you wire a real matrix and a real 2D array to the Divide function, by default the function returns a real matrix output that contains the results of the function. If you wire a real matrix and a complex 2D array, the function returns a complex 2D array because the output requires complex elements. In the second case, a coercion dot appears on the input to which the real matrix is wired to indicate that the function converts the real matrix to a complex 2D array for the operation.

Default Sizes and Values of Matrices

You cannot limit the size of a matrix to a fixed number of elements. However, when you set the default values of a matrix control, you can set the default size.

Do not make the default size of a matrix larger than necessary. If you set a large default size for a matrix, the default data for each element of the matrix saves with the VI and increases the size of the VI on disk.

When you add a matrix control to the front panel, the control initially represents an empty matrix. Each dimension is 0, and the matrix elements are dimmed. A front panel matrix

control has two default values—a matrix default value, either a floating-point or complex default value, and a default value in each cell.

The matrix default value is like the default value of any other front panel control. It is the value of the matrix when the VI is loaded. It also is used for a matrix in a subVI when the calling VI does not wire a value.

The scalar default value is the value used to pad the matrix when the matrix expands. For example, if you set the matrix index to a value beyond the defined portion of the matrix and enter a value in an element greater than one row past the defined portion of the matrix, the elements between the previous end of the matrix and the element you added are set to the scalar default.

For the Matrix functions, values appear outside the original matrix or 2D array when the functions perform an invalid operation. In the case of an invalid operation, the values that LabVIEW returns or fills in at exterior locations depend on the data and element type of the original matrix or 2D array. The invalid operations values are different from the default value of 0 that appears when a matrix expands.



RELATED INFORMATION

[Coercion Dots](#) on page 360

[Grouping Data with Arrays and Clusters](#) on page 460

[Transferring Values between Loop Iterations](#) on page 415

[Displaying Scroll Bars of Controls and Indicators](#) on page 1050

[Creating Custom Controls, Indicators, and Type Definitions](#) on page 165

[Creating Custom Controls, Indicators, and Type Definitions](#) on page 165

[Creating and Editing User-Defined Constants](#) on page 888

[Comparing Arrays](#) on page 1183

[Creating Matrix Controls, Indicators, and Constants](#) on page 1116

Porting and Localizing VIs

This section describes porting VIs among platforms in LabVIEW. It also describes localizing VIs in LabVIEW.



RELATED INFORMATION

[Porting VIs among Platforms](#) on page 318

[Localizing VIs](#) on page 322

[Exporting and Importing VI Strings](#) on page 960

[Example of Exported VI Strings](#) on page 961

Porting VIs among Platforms

Portability issues include differences in filenames, separator characters, resolutions and fonts, possible overlapping of labels, and differences in picture formats.

Portable and Nonportable VIs

VIs are portable among all platforms on which LabVIEW runs as long as the versions of LabVIEW are the same. VIs that contain platform-specific features, such as .NET or ActiveX, are not portable. Porting in these cases breaks the VIs.

LabVIEW uses the same file format on all platforms.

When you open the VI on the new platform, LabVIEW detects the VI is from another platform and recompiles the VI to use the correct instructions for the current processor. If you transfer VIs on a disk formatted for a different platform, you may require a utility program to read the disk.

You cannot port the following VIs:

- VIs distributed in the `vi.lib` directory — Each distribution of LabVIEW contains its own `vi.lib`. Do not move VIs in `vi.lib` across platforms.
- Compatibility VIs — LabVIEW uses compatibility VIs if you created VIs in earlier versions of LabVIEW and open them in a newer version of LabVIEW. Compatibility VIs emulate functionality in previous versions of LabVIEW. Porting breaks these VIs because they call shared libraries that are platform-specific.
- VIs that contain the Call Library Function Node — Porting breaks these VIs unless they can find a library of the same name as the Call Library Function references.
- Platform-specific communication VIs, such as .NET or ActiveX functions on Windows and AppleEvents VIs on macOS.
- The VISA Peek and VISA Poke functions.

Filename Differences

To ease porting VIs among platforms, consider the VI filename.

- Avoid filenames longer than 255 characters.
- Avoid using characters in a filename that are invalid on another platform. The colon `:` is invalid on macOS, the slash `/` is invalid on Linux, and the following characters are invalid on Windows: `\`, `/`, `:`, `*`, `?`, `"`, `<`, `>`, and `|`.
- Filenames are case sensitive on Linux but not elsewhere. If you refer to a VI by name, capitalize the name consistently wherever you use it.

Separator Character Differences

If you develop a VI for use on other platforms, do not use platform-specific path separator characters \, /, : in filenames. Avoid any special characters in filenames because different file systems interpret them differently. For example, hidden files on Linux begin with a period.

Resolution and Font Differences

Fonts can vary from platform to platform. After you port a VI, select new fonts to obtain an appealing display. When you design VIs, keep in mind the three fonts that map best among platforms: the Application font, the System font, and the Dialog font.

The Application font is the LabVIEW default font used in the Controls palette, the Functions palette, the context help, and tip strips.

The System font is the LabVIEW default font used for menus.

The Dialog font is the LabVIEW default font used for text in dialog boxes and system controls.

The default font is the same for all three fonts depending on the platform.

- (Windows) LabVIEW uses Segoe UI by default.
- (macOS) LabVIEW uses Lucida Grande by default.
- (Linux) LabVIEW uses Helvetica by default.

When you port a VI that contains one of these fonts to another platform, LabVIEW ensures the font maps to something similar on that platform.

If you do not use the defined fonts, the font can change size on the new platform because of the differences in the fonts available and differences in the resolution of the display on the other platform. For example, if you select Geneva or New York on macOS and port the VI to Windows, LabVIEW cannot find that font name on Windows and substitutes it with Arial. The substituted font may not match the original, and objects may overlap. If you port a VI to Windows and Windows does not recognize the fonts, the new font may display differently.

Similarly, a font may not be recognized on another language system. For example, a Japanese system may not recognize the Windows font Arial. Use Application, Dialog, and System fonts to avoid resetting the fonts after localizing an application.

If you use a defined font on a section of text, avoid changing the size of that text. If you change the font to something other than the default size and then port the VI to a different platform, LabVIEW tries to match the font with the new size, which might be inappropriate depending on the resolution of the screen.

Character Code Issues

A computer represents data as bytes. It stores letters and other characters by assigning each character a code.

In the simplest case, one byte corresponds to one character according to a mapping table, or encoding. Most operating systems support many different encodings. One type of encoding is single-byte encoding, which can represent up to 256 characters. To represent additional characters beyond English characters, operating systems also support multi-byte encodings. Multi-byte encodings can represent thousands of characters but require multiple bytes for each character. Japanese and Chinese systems use multi-byte encodings to represent several thousands of characters. Almost all operating systems also now support variable-byte encodings that use fewer bytes for more common characters. A variable-byte encoding, such as UTF-8, uses a single-byte for the first 128 characters and adds an additional byte as needed to expand the number of possible character representations.



NOTE

In applications on multi-byte systems, do not attempt to parse a string by converting the string into an array of bytes because certain characters may use two bytes. Some String functions do not handle multi-byte characters.

The most common 8-bit encoding in the United States and Western European countries is ISO Latin I, which is an extension of ASCII.

ASCII encoding is a 7-bit code (from 0 to 127). It contains upper and lowercase English, American English punctuation, base 10 numbers, and a few control codes. Control codes are a set of non-displayable characters used for text formatting. You can print most of the ASCII characters in the \0x21 to \0x7F range.

To represent international characters, operating systems support 8-bit encodings such as ISO Latin 1 as extensions of ASCII. The lower 128 character codes are identical to ASCII, and the upper 128 are different for each encoding. The various international characters reside in the upper 128 character codes. LabVIEW for Windows and Linux use 8-bit character sets but different encodings, which can cause portability problems.

For the United States and Western European countries, LabVIEW for Windows and Linux use the ISO Latin 1 encoding, also known as a code page. For operating systems localized to Japanese, Chinese, or Korean, LabVIEW uses different multi-byte encodings to represent characters in each of these languages. LabVIEW determines which encoding to use based on the OS system locale and language settings. These encodings are the same on Windows and Linux, so portability is less of an issue. However, LabVIEW for macOS (64-bit) uses a variable-length extension of ASCII called UTF-8, which can represent any language.



Note All operating systems support the UTF-8 encoding. However, LabVIEW does not use UTF-8 on Windows to maintain compatibility with existing user applications.

When you port a VI from one platform to another, such as from macOS to Windows or from Windows to macOS, you might encounter problems because of the differences between platform code pages, especially if you use non-ASCII characters in the upper 128 range. For example, the word **café** in French uses the extended character **é**, whose hexadecimal value

is `\0xE9` on Windows. To display this same character **é** on 64-bit macOS, you use the two-byte sequence `\0xC3\0xA9`.

LabVIEW performs a code page conversion for elements such as captions, VI and parameter tips and descriptions, VI titles, and other private data. Private data includes listbox item names, table row and column headers, font used for table cells, graph plot names, graph annotation names, and cursor names. However, LabVIEW does not map labels or string content. Labels and string content might not appear as you expect after you port them to a new platform, and you may need to modify them manually. Therefore, if you plan to port your VIs from Windows or Linux to macOS and vice versa, avoid using international characters in labels or string content.

LabVIEW does not map string content because strings can be interpreted as binary information. If you communicate with an instrument using GPIB or serial communication and if your instrument expects the binary string `\0x63\0x61\0x66\0xC3\0xA9`, this string appears as **café** on the 64-bit macOS platform, and it appears as **cafÃ©** on the Windows platform. In both cases, you should send the same binary string to the instrument. To avoid this type of communication error, LabVIEW does not perform character mapping on string values.

LabVIEW does not map labels because remapping labels to a new encoding can break a VI. For example, if you use the **Unbundle by Name** function and LabVIEW remaps the labels of cluster contents, the VI can break because references to the control by name no longer match.

Overlapping Labels

When you move a VI to a new platform, controls and labels might change size, depending on whether the fonts are smaller or larger on the new platform. LabVIEW tries to keep labels from overlapping their owners by moving them away from the owning control. Also, every label and constant has a default **Size to Text** attribute. When you first create a label or constant, this attribute is set so the bounds of the object resize as necessary to display all the enclosed text.

If you manually resize the label, LabVIEW turns off this attribute, and the item in the shortcut menu no longer shows a checkmark. With **Size to Text** turned off, the bounds of the object stay constant, and LabVIEW clips or crops the enclosed text as necessary. If you do not want LabVIEW to clip text when you move among platforms, leave this attribute on for labels and constants. After you resize the label, you can manually turn on **Size to Text**.

Not all computers have high-resolution monitors. However, if you have a monitor with a high resolution, do not make your front panels very large if you want them to port well. For best portability results, leave extra space between controls and avoid overlapping controls. When you port a label that overlaps another label, it might overlap the whole control when you enlarge the font on the new platform.

Picture Differences

The most basic type of picture contains a bitmap, which is a series of values that specify the color of each pixel in the picture. More complex pictures can contain commands the computer system executes every time it displays the picture.

Use the paint layer of a graphics application to create bitmap-based pictures. Bitmaps are common storage formats for pictures on all platforms. If you use pictures that contain bitmaps on front panels, the pictures usually look the same when you load the VIs on another platform. However, pictures that contain drawing commands such as filling might include commands that other platforms do not support. Check how VIs look on another platform if you expect to use them there.

Use the draw layer of a graphics application to create pictures that contain drawing commands. To make the pictures more portable, paste the final picture into a paint layer of a graphics application before you import the picture into the VI.

(Windows and macOS) With some graphic applications on Windows and many graphics applications on macOS, you can cut or copy a picture with a nonrectangular shape. For example, on macOS, you can use a Lasso tool to select the outline of a circle, triangle, or a more complicated shape such as a musical note.

Other platforms might draw these irregular shapes on a rectangular white background. On Windows, if you want pictures with irregular shapes and pictures that scale well, use applications that support enhanced metafiles.



RELATED INFORMATION

[Text Characteristics](#) on page 113

Localizing VIs

You can localize a VI by [exporting the VI strings](#) into a tagged file, translating the file, and importing it back into the VI. Localized versions of LabVIEW use the same executable as the U.S. version. Therefore, localized versions are compatible with the U.S. version.

The tagged text file, also known as a VI string file, contains information to localize the VI title, description, control captions, control descriptions, and other private data of controls. Because you cannot localize control labels, a control also has a caption. By exporting the VI strings into a text file, LabVIEW creates a caption for each control and substitutes the label with a caption on the front panel.

Besides translating strings on the front panel, you can use localized decimal separators when converting numbers to strings. Use the Format Date/Time String function to specify how to display the date and time.

You can localize the following VI strings:

- VI window titles and descriptions
- Object caption and descriptions
- Free labels
- Default data (string, table, path, and array default data)
- Private data (listbox item names, table row and column headers, graph plot names, graph cursor names, and graph annotation names)

- Block diagram strings (free labels and string constants), if you specify it

**NOTE**

Localizing block diagram string constants might cause VIs to break. Breaking VIs is especially likely when block diagram string constants contain multibyte characters.

Syntax of the VI String File

The format of the VI string file is much like an HTML file. The system marks every element with a start tag and an end tag. A start tag begins with < and ends with >. An end tag starts with </ and ends with >. LabVIEW ignores white space characters except within text. Because the character < indicates the beginning of a tag, LabVIEW uses << for the less than character in text. For the > character, LabVIEW uses >> for the greater than character. LabVIEW replaces the double quotation mark with ". Also, LabVIEW denotes end-of-line characters as <CR>, <CRLF>, or <LF>, which are treated as a carriage return, a carriage return followed by a line feed, and a line feed, respectively. The format of the VI string file is intended to be machine readable. Do not be concerned if you find it difficult to read. Each line of an export file contains no more than two tags for private or default data. LabVIEW also indents items once for each nesting level. If you change or delete tags, LabVIEW issues errors when it imports the file into the VI. Refer to the [Example of Exported VI Strings](#) topic for an example text file containing exported VI strings.

The following table lists the VI tag types and the corresponding VI tag syntax.

VI Tag Type	VI Tag Syntax
[VI string file].	<VI [vi attributes] > [vi info] </VI>
[VI attributes].	syntaxVersion=5 LVversion=nnn revision=nnn name="text"
[VI info].	[vi title] [description] [content] [bdcontent]
[VI title].	<TITLE>text</TITLE> . <TITLE><NO_TITLE></TITLE>.
[description].	<DESC>text</DESC>.
[content].	<CONTENT>[grouper] [objects]</CONTENT>
[bd content].	<BDCONTENT>[bd objects] </BDCONTENT>

A space separates VI attributes. No space is permitted between the attribute name and the following equal sign and between the equal sign and the attribute value.

For example:

```
<VI syntaxVersion=5 LVversion=4502007 revision=10 name="A0
Generate Waveform.vi">
```

```
<TITLE>A0 Generate Waveform.vi</TITLE>
```



```
<DESC>This VI generates a timed, simple-buffered waveform for
the given output channel at the specified update rate.</DESC>

<CONTENT>

.....

</CONTENT>

</VI>
```

The following table lists the tags that describe the content of the front panel (free and object-owned labels, caption labels, attributes) and the corresponding tag syntax.

Content Tag Type	Content Tag Syntax
[content]	<CONTENT>[grouper] [objects]</CONTENT>
[grouper]	<GROUPER>[parts]</GROUPER>
[objects]	([control] [label]) *
[control]	<CONTROL [control attributes]> [control info] </CONTROL>
[label]	<LABEL>[style text] </LABEL>
[style text]	<STEXT>text with font information </STEXT>

Between <STEXT> and </STEXT>, you can type font specifications. LabVIEW encodes font information using the following format: . You can list the font attributes in any order. Font specification is different from other elements because it has no end tag. For example, you can describe a caption with the text " **Bold**label " as follows:

```
<LABEL><STEXT><FONT name="times new roman" size=12
style='B'>Bold <FONT style='I'>label</STEXT></LABEL>
```

You can define fonts as predef to specify one of the predefined fonts—Application font, System font, or Dialog font.

The <GROUPER></GROUPER> tags enclose front panel objects that you grouped together. These tags also enclose each pane you configure with splitter bars. Because the front panel always contains at least one pane, the VI string file always contains at least one set of <GROUPER></GROUPER> tags. Clusters also contain panes.

The following table lists the tags that describe the objects that contain data and the corresponding tag syntax.

Content Tag Type	Content Tag Syntax
[control]	<CONTROL [control attributes]> [control info] </CONTROL>

Content Tag Type	Content Tag Syntax
[control attributes]	ID=xxx type="Boolean" name="switch"
[control info]	[description] [tip strip] [parts] [privData section] [defData section] [content]
[tip strip]	<TIP>text</TIP>
[parts]	<PARTS> [part]*</PARTS>
[part]	<PART [part attributes]> [part info] </PART>
[part attributes]	partID=nnn partOrder=nnn
[part info]	[control] [label] [multiLabel]

The following is an example of a ring control description with a caption of "Ring" and options **Load**, **Unload**, **Open**, and **Close**.

```
<CONTROL ID=87 type="Ring" name="RINGcontrol">
<DESC>ring control</DESC>
<PARTS>
<PART ID=12 order=0
type="Ring Text"><MLABEL><STRINGS><STRING>Load<
STRING><STRING>Unload<STRING><STRING>Open<
STRING><STRING>Close<STRING></STRINGS></MLABEL></PART>

<PART ID=82 order=0 type="Caption"><LABEL><STEXT><FONT
color=FF0033 size=12>Ring<STEXT></LABEL></PART>

</PARTS>
</CONTROL>
```

LabVIEW uses the MLABEL (multilabel) tag above to designate the option string on a ring control or the strings on Boolean buttons, a string for each of the four states. The following is a generic description of the MLABEL tag syntax:

```
[multiLabel]<MLABEL> [mlabel info] </MLABEL>

[mlabel info] [font] [strings]
```

The following table lists the tags that describe the default data for strings, tables, arrays, and paths and the corresponding tag syntax.

Content Tag Type	Content Tag Syntax
[defData section]	<DEFAULT> [defData] </DEFAULT>
[defData]	[str def] [table def] [arr data] [path data]
[str def]	[string] <SAME_AS_TEXT>
[table def]	[strings]
[arr data]	<ARRAY nElems=n> [arr element data] </ARRAY>
[arr element data]	[clust data] [str data] [non-str data]
[str data]	[string]
[non-str data]	<NON_STRING>
[clust data]	<CLUSTER nElems=n> [clust element data] </CLUSTER>
[clust element data]	[clust data] [str data] [non-str data] [arr data] [path data]
[path data]	<PATH type="absolute"> a<SEP> system </PATH>

For [arr data], n [arr element data] must follow the <ARRAY> tag. Likewise, for [clust data], n [clust element data] must follow the <CLUSTER> tag.

For the string control default data [str def], use a special tag, <SAME_AS_LABEL>, which indicates the string default data is the same as the text label on the string parts list. When you use <SAME_AS_LABEL>, it is unnecessary to type the same text for both the text label and the string default data.

For path control default data [path data], the <PATH> start tag can have an attribute that specifies the path type. The possible attribute values are "absolute", "relative", "not-a-path", and "unc". A <SEP> tag separates the path segments that come between the <PATH> and </PATH> tags. For example, on the Windows platform, an absolute path c:\windows\temp\temp.txt is written as follows:

```
<PATH type="absolute">c<SEP>windows<SEP>temp<SEP>temp.txt</PATH>
```

The following tables list the tags that describe private data and the corresponding tag syntax.

Content Tag Type	Content Tag Syntax
[privData section]	<PRIV> [privData]* </PRIV>
[privData]	([strings] [col header] [row header] [cell fonts] [plots] [cursors] [path privData] [tab control privData])
[strings]	<STRINGS> [string]* </STRINGS>

Content Tag Type	Content Tag Syntax
[string]	<STRING> text </STRING>
[col header]	<COL_HEADER> [string]* </COL_HEADER>
[row header]	<ROW_HEADER> [string]* </ROW_HEADER>
[cell fonts]	<CELL_FONTS> [cell font]* </CELL_FONTS>
[plots]	<PLOTS> [string]* </PLOTS>
[cursors]	<CURSORS> [string]* </CURSORS>
[cell font]	[row# col#][font]
[font]	
[path privData]	<PROMPT>text</PROMPT>
	<MTCH_PTN>TEXT</MTCH_PTN> <STRT_PTH>[path data]</STRT_PTH>
[tab control privData]	<PAGE_CAPTIONS>[string]*</PAGE_CAPTIONS>
[tab control page]	<PAGE> [description] [tip strip] [objects] </PAGE>

Content Tag Type	Content Tag Syntax
[bd content]	<BDCONTENT> [bdobjects]</BDCONTENT>
[bd objects]	(([control] [label] [node]))*
[node]	<NODE [node attributes]>[node info]</NODE>
[node attributes]	ID=xxx type="Sequence"
[node info]	[description] [bd content]

Editing VI Window Titles

You can customize the VI window title to make it more descriptive than the VI filename, which is important for localized VIs. You can localize the VI window title, which is not governed by file-system naming constraints and is still recognized by the VIs that might call it because the calling VI references the subVI by its filename, not its title. Use the [export/import tools](#) to localize the window title, or use the Window Appearance page of the VI Properties dialog box to manually change the VI window title while you edit a VI. You also can use the VI Server to change the VI window title programmatically.

Period and Comma Decimal Separators

By default, LabVIEW adapts to use the appropriate decimal separator on your operating system preferences, which affects the display of numbers within controls and the operation of functions that convert between numbers and strings. This causes problems when passing numbers as strings to instruments or parsing instrument output strings.

The following functions use the operating system settings by default. To use a period, you must wire a False value to the Boolean input called **Use System Decimal Point (+)**.

- Fract/Exp String To Number
- Number To Fractional String
- Number to Exponential String
- Number To Engineering String

For the functions that take a format string, the %x; format code, where x represents the separator character, can be used to specify the decimal separator for that specific call. These functions include:

- Format Into String
- Scan From String
- Format Value
- Scan Value
- Array To Spreadsheet String
- Spreadsheet String To Array
- Read Delimited Spreadsheet

Select **Tools»Options**, select Front Panel from the **Category** list, and place a checkmark in the **Use localized decimal point** checkbox to specify whether to use the system settings (locale) for the decimal separator or a period.

Displaying Date and Time Values as Strings

You can set the display format of date and time for strings by using the Format Date/Time String function by default. LabVIEW reads the operating system settings and displays the date/time representation appropriate for the current locale. LabVIEW returns abbreviated weekday and month names as numeric values for systems that do not support abbreviated names, such as Chinese and Korean. You can customize the display by selecting **Display Format** from the shortcut menu.

Time and Date Format for Controls

Use the *time stamp* control to use, view, and store absolute time with high precision. You can select a custom-fixed format or use a system (locale-specific) format by selecting **Display Format** from the shortcut menu and configuring the Display Format page. You also

can use a numeric control to display time stamp values, but the numeric control holds a relative quantity. The time stamp control holds an absolute quantity.

Debugging Techniques

If a VI is not broken, but you get unexpected data, you can use several techniques to identify and correct problems with the VI or the block diagram data flow. If you cannot debug the VI using the following techniques, the VI might have a *race condition*.



NOTE

To *debug a stand-alone application or shared library*, start by using these debugging techniques on the block diagram of the startup VI.

Error Clusters

Most built-in VIs and functions have *error in and error out parameters*. These parameters detect errors encountered in each node on the block diagram and indicate if and where an error occurred. You also can use these parameters in the VIs you build.

When you wire the error parameters of VIs and functions, the **error in** and **error out** clusters provide the following components of information:

- **status** is a Boolean value that reports TRUE if an error occurred.
- **code** is a 32-bit signed integer that identifies the error numerically. A nonzero error code coupled with a **status** of FALSE signals a warning rather than a error.
- **source** is a string that identifies where the error occurred.

Warnings

Although warnings do not prevent you from running a VI, they can cause unexpected behavior. When you debug a VI, ensure that you *display warnings*. If the VI causes a warning, use the Error List window to determine the cause and correct the VI.

Execution Highlighting

View an animation of the execution of the block diagram by clicking the **Highlight Execution** button, shown as follows.



Execution highlighting shows the movement of data on the block diagram from one node to another using bubbles that move along the wires. Use execution highlighting in conjunction with single-stepping to see how data values move from node to node through a VI.



Note Execution highlighting greatly reduces the speed at which the VI runs. If a VI runs more slowly than expected, confirm that you turned off execution highlighting in subVIs.

If the **error out** cluster reports an error, the error value appears next to **error out** with a red border. If no error occurs, OK appears next to **error out** with a green border.

Single-Stepping

Single-step through a VI to view each action of the VI on the block diagram as the VI runs. The single-stepping buttons, shown as follows, affect execution only in a VI or subVI in single-step mode.



Enter single-step mode by clicking the **Step Into** or **Step Over** button on the block diagram toolbar. Move the cursor over the **Step Into**, **Step Over**, or **Step Out** button to view a tip strip that describes the next step if you click that button. You can single-step through subVIs or run them normally.

When you single-step through a VI, nodes blink to indicate they are ready to execute. If you single-step through a VI with execution highlighting on, an execution glyph, shown as follows, appears on the icons of the subVIs that are currently running.



Probe Tool

Use the Probe tool, shown as follows, to check intermediate values on a wire as a VI runs.



Use the Probe tool if you have a complicated block diagram with a series of operations, any one of which might return incorrect data. Use the Probe tool with execution highlighting, single-stepping, and breakpoints to determine if and where data is incorrect. If data is available, the probe immediately updates and displays the data in the **Debug** window during execution highlighting, single-stepping, or when you pause at a breakpoint. When execution pauses at a node because of single-stepping or a breakpoint, you also can probe the wire that just executed to see the value that flowed through that wire.

Types of Probes

You can check intermediate values on a wire when a VI runs by using a generic probe, by using an indicator on the Controls palette to view the data, by using a supplied probe, by using a customized supplied probe, or by creating a new probe.

Generic

Use the generic probe to view the data that passes through a wire. Right-click a wire and select **Custom Probe»Generic Probe** from the shortcut menu to use the generic probe.

The generic probe displays the data. You cannot configure the generic probe to respond to the data.

LabVIEW displays the generic probe when you right-click a wire and select **Probe**, unless you already specified a custom or supplied probe for the data type.

You can debug a custom probe similar to a VI. However, a probe cannot probe its own block diagram, nor the block diagram of any of its subVIs. When debugging probes, use the generic probe.

Using Indicators to View Data

You also can use an indicator to view the data that passes through a wire. For example, if you view numeric data, you can use a chart within the probe to view the data. Right-click a wire, select **Custom Probe»Controls** from the shortcut menu, and select the indicator you want to use. You also can click the **Select a Control** icon on the Controls palette and select any custom control or type definition saved on the computer or in a shared directory on a server. LabVIEW treats type definitions as custom controls when you use them to view probed data.

If the data type of the indicator you select does not match the data type of the wire you right-clicked, LabVIEW does not place the indicator on the wire.

Supplied

Supplied probes are VIs that display comprehensive information about the data that passes through a wire. For example, the VI Refnum Probe returns information about the VI name, the VI path, and the hex value of the reference. You also can use a supplied probe to respond based on the data that flows through the wire. For example, use an Error probe on an error cluster to receive the status, code, source, and description of the error and specify if you want to set a conditional breakpoint if an error or warning occurs.

The supplied probes appear at the top of the **Custom Probe** shortcut menu. Right-click a wire and select **Custom Probe** from the shortcut menu to select a supplied probe. Only probes that match the data type of the wire you right-click appear on the shortcut menu.

Custom

Use the Create New Probe dialog box to create a probe based on an existing probe or to create a new probe. Right-click a wire and select **Custom Probe»New** from the shortcut menu to display the **Create New Probe** dialog box. Create a probe when you want to have more control over how LabVIEW probes the data that flows through a wire. When you create a new probe, the data type of the probe matches the data type of the wire you right-clicked. If you want to edit the probe you created, you must open it from the directory where you saved it.

After you select a probe from the **Custom Probe** shortcut menu, navigate to it using the **Select a Control** palette option, or create a new probe using the Create New Probe dialog box, that probe becomes the default probe for that data type, and LabVIEW loads that probe when you right-click a wire and select **Probe** from the shortcut menu. LabVIEW only loads probes that exactly match the data type of the wire you right-click. That is, a double precision floating-point numeric probe cannot probe a 32-bit unsigned integer wire even though LabVIEW can convert the data.



NOTE

If you want a custom probe to be the default probe for a particular data type, save the probe in the `user.lib\_probes\default` directory. Do not save probes in the `vi.lib\_probes` directory because LabVIEW overwrites those files when you upgrade or reinstall.

Sampling Probes

(FPGA Module) Use the Sampling probes in host VIs or FPGA VIs to check intermediate values on a wire as a VI runs and to view changes in signal data over time, such as when you need to debug signals from a single-cycle Timed Loop.

When you use a Sampling probe in a host VI, you must first specify a sampling source for the probe. A sampling source determines when LabVIEW reads, or samples, data from associated probes. You can use While Loops, For Loops, or FPGA simulated time as sampling sources depending on whether you use Sampling probes in host VIs or FPGA VIs.

Breakpoints

Use the Breakpoint tool, shown as follows, to place a breakpoint on a VI, node, or wire and pause execution at that location.



When you set a breakpoint on a wire, execution pauses after data passes through the wire and the **Pause** button appears red. Place a breakpoint on the block diagram to pause execution after all nodes on the block diagram execute. The block diagram border appears red and blinks to reflect the placement of a breakpoint.

When a VI pauses at a breakpoint, LabVIEW brings the block diagram to the front and uses a marquee to highlight the node, wire, or line of script that contains the breakpoint. When

you move the cursor over an existing breakpoint, the black area of the Breakpoint tool cursor appears white.

When you reach a breakpoint during execution, the VI pauses and the **Pause** button appears red. Also, the VI background and border begin blinking. You can take the following actions:

- Single-step through execution using the single-stepping buttons.
- Check intermediate values on *probes* that you placed on wires prior to running the VI.
- Check intermediate values on probes that you place after running the VI if you have enabled the *Retain Wire Values* option.
- Change values of front panel controls.
- Check the *Call list* pull-down menu to see the list of VIs that call the VI that is stopped at a breakpoint.
- Click the **Pause** button to continue running to the next breakpoint or until the VI finishes running.

LabVIEW saves breakpoints with a VI, but they are active only when you run the VI. It can be useful to save a breakpoint for repeated use, instead of removing it and recreating it. However, you might not want all breakpoints to be active each time you run the VI. Disable breakpoints so execution does not pause at that breakpoint when you run the VI. You can use the **Debug** window to disable, enable, clear, or locate existing breakpoints. Open the **Debug** window by selecting **View»Debug Window** or by right-clicking an object on the block diagram and selecting **Breakpoint** from the shortcut menu.

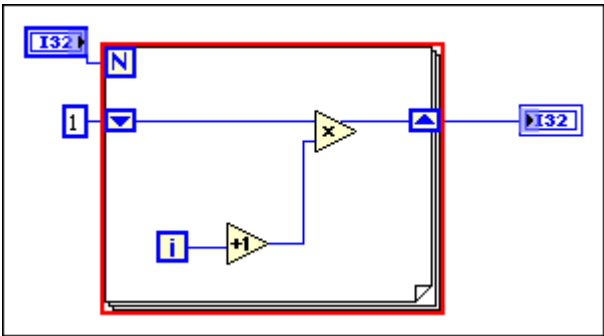
You can *remove breakpoints* individually or throughout the entire VI hierarchy.

How LabVIEW Highlights Breakpoints

LabVIEW behaves differently in terms of how breakpoints are highlighted and when execution pauses dependent on where you have placed breakpoints on the block diagram. You can place breakpoints on structures, nodes, and wires.

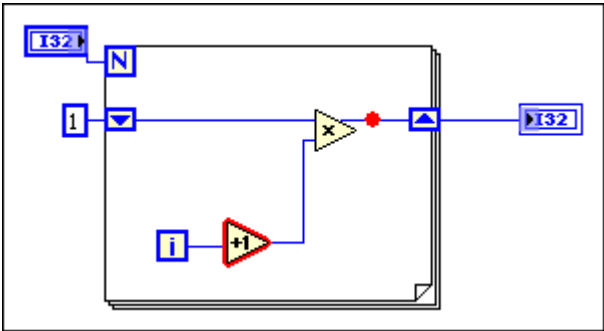
Structure Breakpoint

The following illustration shows a For Loop with a breakpoint. The VI pauses before the entire For Loop finishes executing. A red border around the block diagram highlights the breakpoint. If the block diagram is inside a structure, the structure includes the red border. The pause in execution occurs when all nodes on the block diagram finish running. If a breakpoint is in a loop structure, the pause occurs after each loop iteration.



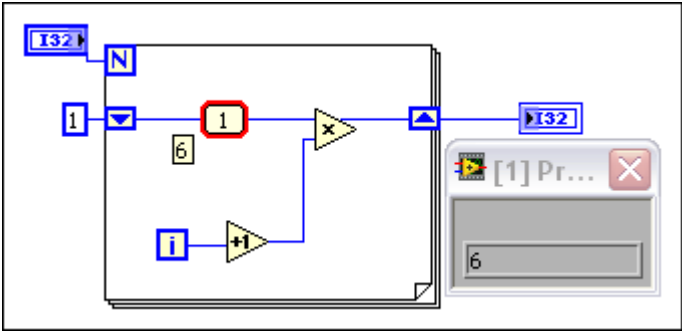
Node Breakpoint

The following illustration shows a breakpoint on the Increment node. The VI pauses before the Increment function runs. One wire also has a breakpoint. The VI pauses again after the Multiply function executes. A red border around the node highlights the breakpoint. The pause in execution occurs just before the node executes. You can probe all the input signals into the node with the Probe tool.



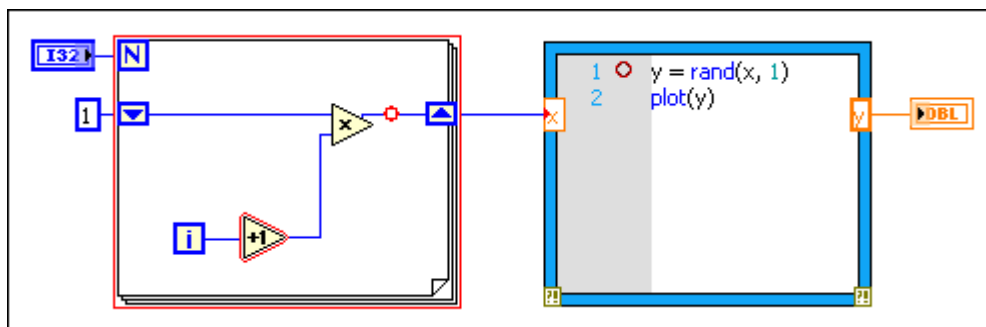
Wire Breakpoint

The following illustration shows a breakpoint on a wire with an attached probe. When you have selected *Retain Wire Values* and you place the cursor over a wire, a tip strip appears, as shown, that includes the value of the data in the wire. The pause in execution occurs after data pass through the wire and the probe displays the data from the current iteration. A red bullet in the middle of the wire highlights the breakpoint. If you attach a probe to the wire, the probe also has a red border.



Disabled Breakpoint

The following illustration shows breakpoints on the For Loop and an Increment node. All breakpoints are disabled. The VI does not pause at any of the breakpoints. The red border around the block diagram objects is thinner than normal to indicate that you have disabled the breakpoint. If you enable the breakpoints, the red border returns to a thicker width, as shown in the previous illustrations.



Suspending Execution

Suspend execution of a subVI to edit values of controls and indicators, to control the number of times the subVI runs before returning to the caller, or to go back to the beginning of the execution of the subVI. You can cause all calls to a subVI to start with execution suspended, or you can suspend a specific call to a subVI.

To suspend all calls to a subVI, open the subVI and select **Operate»Suspend when Called**. The subVI automatically suspends when another VI calls it. If you select this menu item when single-stepping, the subVI does not suspend immediately. The subVI suspends when it is called.

To suspend a specific subVI call, right-click the subVI node on the block diagram and select **SubVI Node Setup** from the shortcut menu. Place a checkmark in the **Suspend when called** checkbox to suspend execution only at that instance of the subVI.

The VI Hierarchy window, which you display by selecting **View»VI Hierarchy**, indicates whether a VI is paused or suspended. An arrow glyph, shown as follows, indicates a VI that is running regularly or single-stepping.



A pause glyph, shown as follows, indicates a paused or suspended VI.



A green pause glyph, or a hollow glyph in black and white, indicates a VI that pauses when called. A red pause glyph, or a solid glyph in black and white, indicates a VI that is currently paused. An exclamation point glyph, shown as follows, indicates that the subVI is suspended.



A VI can be suspended and paused at the same time.

Determining the Current Instance of a SubVI

When you pause a subVI, the **Call list** pull-down menu on the toolbar lists the chain of callers from the top-level VI down to the subVI. This list is not the same list you see when you select **Browse»This VI's Callers**, which lists all calling VIs regardless of whether they are currently running. Use the **Call list** menu to determine the current instance of the subVI if the block diagram contains more than one instance. When you select a VI from the **Call list** menu, its block diagram opens and LabVIEW highlights the current instance of the subVI.

You also can use the Call Chain function to view the chain of callers from the current VI to the top-level VI.

Debugging Sections of Block Diagrams

You can run a VI with a section of the block diagram disabled, similar to commenting out a section of code in a text-based programming language. Disable a section of the block diagram to determine if the VI performs better without it. Place the section you want to disable inside a Diagram Disable structure.

You also can debug a section of code to isolate the problem. To debug a section of code, place the code you want to debug inside a Conditional Disable structure.

Instrumentation

If a probe, breakpoint, or execution highlighting takes a long time to debug the code of a VI, create an instrumentation VI. Use an instrumentation VI to observe the execution of a VI and debug the problem.

Program the instrumentation VI to read data from a VI you want to debug and to generate a report with the results. In the instrumentation VI, specify the section of code you want to debug, point to a path to the VI you want to debug, and include a time step.

An instrumentation VI works similarly to the Desktop Execution Trace Toolkit because both tools provide you with the execution order of data in a block diagram.

Verify Object Behavior

In addition to using debugging tools, you also can use the following techniques to identify and correct problems with the VI or the block diagram data flow:

- Check the representation of controls and indicators to see if you are receiving overflow because you converted a floating-point number to an integer or an integer to a smaller integer. For example, you might wire a 16-bit integer to a function that only accepts 8-bit integers. This causes the function to convert the 16-bit integer to

an 8-bit representation, potentially causing a loss of data. Refer to [Preventing Undefined Data](#) for more information.

- Determine if the data that one function or subVI passes is [undefined](#). This often happens with numbers. For example, at one point in the VI an operation could have divided a number by zero, thus returning Inf (infinity), whereas subsequent functions or subVIs were expecting numbers.
- Determine if any For Loops inadvertently execute zero iterations and produce empty arrays. Refer to [Execution Highlighting](#) and [Preventing Undefined Data](#) for more information.
- Verify you initialized [shift registers](#) properly unless you intend them to save data from one execution of the loop to another.
- Check the [cluster element order](#) at the source and destination points. LabVIEW detects data type and cluster size mismatches at edit time, but it does not detect mismatches of elements of the same type.
- Check that the VI does not contain hidden subVIs. You inadvertently might have hidden a subVI by placing one directly on top of another node or by decreasing the size of a structure without keeping the subVI in view. Check the inventory of subVIs the VI uses against the results of **View»Browse Relationships»This VI's SubVIs** and **View»Browse Relationships»Unopened SubVIs** to determine if any extra subVIs exist.
- Select **View»VI Hierarchy** to find unwired subVIs. Unlike unwired functions, unwired VIs do not always generate errors unless you configure an input to be [required](#). If you mistakenly place an unwired subVI on the block diagram, it executes when the block diagram does. Consequently, the VI might perform extra actions.
- Use the Context Help window to check the default values for each function and subVI on the block diagram. VIs and functions pass default values if recommended or optional inputs are unwired. For example, a Boolean input might be set to TRUE if unwired.



RELATED INFORMATION

[Debug Window - LabVIEW Programming Reference Manual](#)

Correcting Broken VIs

If a VI does not run, it is a broken, or nonexecutable, VI. The **Run** button appears broken, shown as follows, when the VI you are creating or editing contains errors.



If the button still appears broken when you finish wiring the block diagram, the VI is broken and cannot run.

Finding Causes for Broken VIs

Warnings do not prevent you from running a VI. They are designed to help you avoid potential problems in VIs. Errors, however, can break a VI. You must resolve any errors before you can run the VI.

Click the broken **Run** button or select **View»Error List** to find out why a VI is broken. The Error list window lists all the errors. The **Items with errors** section lists the names of all items in memory, such as VIs and project libraries that have errors. If two or more items have the same name, this section shows the specific application instance for each item. The **errors and warnings** section lists the errors and warnings for the VI you select in the **Items with errors** section. The **Details** section describes the errors and in some cases recommends how to correct the errors. Click the **Help** button to display a topic in the LabVIEW Help that describes the error in detail and includes step-by-step instructions for correcting the error.

Click the **Show Error** button or double-click the error description to highlight the area on the block diagram or front panel that contains the error.

The toolbar includes the **Warning** button, shown as follows, if a VI includes a warning and you placed a checkmark in the **Show Warnings** checkbox in the Error list window.



Configure LabVIEW to always show warnings in the Error list window by selecting **Tools»Options**, selecting **Environment** from the **Category** list, and placing a checkmark in the **Show warnings in Error List dialog by default** checkbox in the **Debugging** section. You can make this change with the **Error list** window open and see the change immediately.

Common Causes of Broken VIs

The following list contains common reasons why a VI is broken while you edit it:

- The block diagram contains a broken wire because of a mismatch of data types or a loose, unconnected end.
- A *required block diagram terminal* is unwired.
- A *subVI* is broken or you edited its connector pane after you placed its icon on the block diagram of the VI.

Handling Errors

Users can encounter errors in even the most reliable VIs. Without a mechanism to check for errors, you can only know that a VI does not work properly. Error checking identifies why and where errors occur in your VI.

For example, one common situation in which you can encounter errors occurs when you perform any kind of input and output (I/O). Almost all I/O functions return error

information. Include error checking in VIs, especially for I/O operations (file, serial, instrumentation, data acquisition, and communication), and provide a mechanism to handle errors effectively.

Automatic Error Handling

By default, LabVIEW automatically handles any error when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box.



NOTE

The LabVIEW Run-Time Engine does not support automatic error handling.

Disabling Automatic Error Handling

You also can disable automatic error handling in preference of other error handling methods. For example, if an I/O VI on the block diagram times out, you may want the VI to retry for a certain period of time instead of allowing automatic error handling to stop the application and display an error dialog box. You can implement custom error handling on the block diagram of the VI.

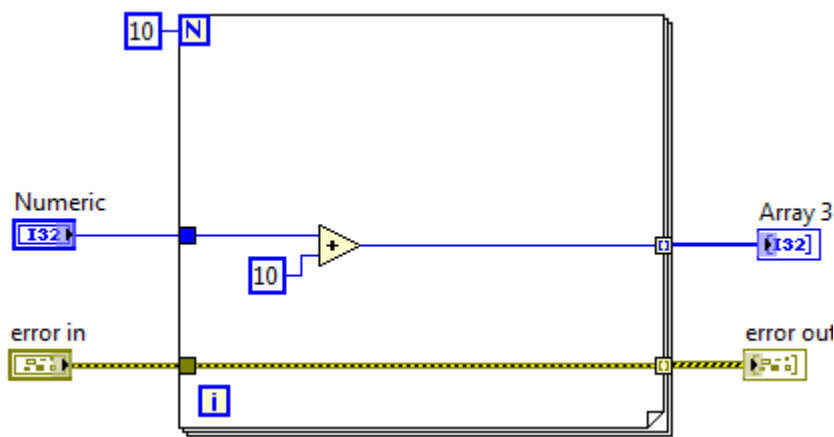
Refer to the following table for information about how to disable automatic error handling for different types of VIs.

Type of VI	Disabling Automatic Error Handling
Current VI	Select File»VI Properties . Then select Execution from the Category pull-down menu and remove the checkmark from the Enable automatic error handling checkbox.
New, blank VIs	Select Tools»Options . Then select Block Diagram from the Category list and remove the checkmark from the Enable automatic error handling in new VIs checkbox.
SubVI or function within a VI	Create an error out indicator in the subVI or wire the error out parameter of that subVI to the error in input of another subVI or function.

Other Methods of Error Handling

Error handling in LabVIEW follows the dataflow model, similar to how data values flow through a VI. Wire the error information from the beginning of the VI to the end. Include an error handler VI at the end of the VI to determine if the VI ran without errors. Use the **error in** and **error out** clusters in each VI to pass the error information through the VI. Error clusters typically provide the same [standard error in](#) and [standard error out](#) functionality. The error clusters are [flow-through parameters](#).

For example, the following image displays a VI with standard error cluster.



As the VI runs, LabVIEW tests for errors at each execution node. If LabVIEW does not find any errors, the node executes normally. If LabVIEW detects an error, the node passes the error to the next node without executing that part of the code. The next node does the same thing, and so on. At the end of the execution flow, LabVIEW reports the error.

Refer to the following table for information about the elements of an **error in** and **error out** cluster.

Cluster Element Name	Cluster Element Description
status	A Boolean value that reports TRUE if an error occurred
code	A 32-bit signed integer that identifies the error numerically. A nonzero error code coupled with a status of FALSE signals a warning rather than an error
source	A string that identifies where the error occurred



Note Some VIs, functions, and structures that accept Boolean data also recognize an error cluster. For example, you can wire an error cluster to a Boolean function or to the Boolean inputs of the Select, Quit LabVIEW, or Stop functions to handle errors using logical operations.

You can develop and implement your own error handling in LabVIEW. Use the LabVIEW error handling VIs and functions on the Dialog & User Interface palette and the **error in** and **error out** parameters of most VIs and functions to manage errors. For example, if LabVIEW encounters an error, you can display the error message in different kinds of dialog boxes.

The VIs and functions native to LabVIEW return errors in one of two ways—with numeric error codes or with an error cluster. Typically, functions use numeric error codes, and VIs use an error cluster, usually with error inputs and outputs.

You can use debugging tools in conjunction with error handling to help identify and manage errors.

You also can refer to the following table for examples of common methods for using error clusters.

Method	Description
Stopping a loop with an error cluster	To stop the iteration of a loop, you can wire an error cluster to either the conditional terminal of a While Loop or to a For Loop with a conditional terminal. If you wire an error cluster to a conditional terminal, only the TRUE or FALSE value of the status parameter of the error cluster passes to the terminal. If an error occurs, the loop stops. Also, the shortcut menu items Stop if True and Continue if True change to Stop on Error and Continue while Error . In a For Loop with a conditional terminal, you also must wire a value to the count terminal or auto-index an input array to set a maximum number of iterations. The For Loop executes until an error occurs or until the number of set iterations completes.
Executing code conditionally according to error status	When you wire an error cluster to the selector terminal of a Case structure , the case selector label displays two cases: Error and No Error . If an error occurs, the Case structure executes the Error subdiagram. To create a new VI with a Case structure for error handling, use the SubVI with Error Handling template VI . The border of the Case structure also changes color, red for Error and green for No Error .

Using the Standard Functionality for error in Parameters

Many LabVIEW nodes such as VIs and functions contain an **error in** parameter you can use to [set up error I/O](#) in VIs in order to [manage errors](#) in them. These parameters typically provide the same, standard functionality. When a node exhibits different parameter functionality, the exceptions are documented in the reference material for that node.



NOTE

Some nodes, such as error handling VIs, contain an **error in** parameter that does not provide standard **error in** functionality but that does contain standard **error in** cluster elements.

Standard **error in** functionality is as follows:

	error in describes error conditions that occur before this node runs. The default is no error. If an error occurred before this node runs, the node passes the error in value to error out . This node runs normally only if no error occurred before this node runs. If an error occurs while this node runs, it runs normally and sets its own error status in error out . Use the Simple Error Handler or General Error Handler VIs to display the description of the error code. Use exception control to treat what is normally an error as no error or to treat a warning as an error. Use error in and error out to check errors and to specify execution order by wiring error out from one node to error in of the next node.

error in contains the following cluster elements:

	status is TRUE (X) if an error occurred before this node ran or FALSE (checkmark) to indicate a warning or that no error occurred before this node ran. The default is FALSE.
	code is the error or warning code. The default is 0.If status is TRUE, code is an error code. If status is FALSE, code is 0 or a warning code.
	source specifies the origin of the error or warning and is, in most cases, the name of the node that produced the error or warning. The default is an empty string.

Using the Standard Functionality for error out Parameters

Many LabVIEW nodes such as VIs and functions contain an **error out** parameter you can use to [set up error I/O](#) in VIs in order to [manage errors](#) in them. These parameters typically provide the same, standard functionality. When a node exhibits different parameter functionality, the exceptions are documented in the reference material for that node.

Standard **error out** functionality is as follows:

	error out contains error information. If error in indicates that an error occurred before this VI or function ran, error out contains the same error information. Otherwise, it describes the error status that this VI or function produces. Right-click the error out front panel indicator and select Explain Error from the shortcut menu for more information about the error.

error out contains the following cluster elements:

	status is TRUE (X) if an error occurred or FALSE (checkmark) to indicate a warning or that no error occurred.
	code is the error or warning code. If status is TRUE, code is an error code. If status is FALSE, code is 0 or a warning code.
	source specifies the origin of the error or warning and is, in most cases, the name of the node that produced the error or warning.

Predefined Command Line Operations

The command line interface for LabVIEW supports the following operations:

- `MassCompile` — Mass compiles the files in the specified directory.
- `ExecuteBuildSpec` — Builds an application or a library using the settings in the specified build specification(s) and returns the path of the output files.
- `RunVI` — Runs a VI with the predefined connector pane interface and returns the output or error information.

- `CloseLabVIEW` — Closes LabVIEW without any prompts.
- `CreateComparisonReport` — Generates a VI comparison report for two specified VIs.
- (VI Analyzer Toolkit) `RunVIAAnalyzer` — Runs the specified VI analyzer task in the LabVIEW VI Analyzer Toolkit and saves the test report to the specified location.
- (Unit Test Framework Toolkit) `RunUnitTests` — Runs tests on the specified files in the LabVIEW Unit Test Framework Toolkit and saves the JUnit file to the specified location.

You can also [create custom operations](#) to run in LabVIEW.

MassCompile

[Mass compiles](#) the files in the specified directory.

The following table lists the available arguments of this operation. Required arguments are in bold.



NOTE

Caching VIs can significantly decrease the execution time of a mass compile because LabVIEW does not have to load cached VIs when you compile a top-level VI. However, VI caching consumes more system memory. If you mass compile a folder that contains VIs with the same name in different subfolders and the VIs do not belong to a library, VI caching may cause [cross-linking](#).

To run this operation in the CLI for LabVIEW, execute a command using the following syntax:

```
LabVIEWCLI -OperationName MassCompile -DirectoryToCompile <
directory of file(s) > -MassCompileLogFile < path to log file
> -AppendToMassCompileLog < true or false > -NumOfVIsToCache
< number of VIs to cache > -ReloadLVSBs < true or false >
```

For example, `LabVIEWCLI -OperationName MassCompile -DirectoryToCompile "C:\temp" -MassCompileLogFile "C:\temp\log.txt" -AppendToMassCompileLog true -NumOfVIsToCache 0 -ReloadLVSBs false`

ExecuteBuildSpec

[Builds](#) an application or a library using the settings in the specified build specification(s) and returns the path of the output files.

The following table lists the available arguments of this operation. Required arguments are in bold.

Argument	Description	Default
-ProjectPath	Path to the LabVIEW project .lvproj file that contains the <i>build specification</i> .	—
-TargetName	Target that contains the build specification.	My Computer
-BuildSpecName	Name of the build specification that appears under Build Specifications in the Project Explorer window.	Empty string—The CLI for LabVIEW builds all build specifications under the specified target.

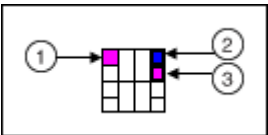
To run this operation in the CLI for LabVIEW, execute a command with the following syntax:

```
LabVIEWCLI -OperationName ExecuteBuildSpec -ProjectPath <
path to project > -TargetName < name of target > -
BuildSpecName < name of build specification >
```

```
For example, LabVIEWCLI -OperationName ExecuteBuildSpec -
ProjectPath "C:\temp\test.lvproj" -TargetName "My Computer" -
BuildSpecName "My DLL"
```

RunVI

Runs a VI with the following connector pane interface and returns the output or error information.



Terminal	Data Type	Description
①		Command line arguments for the VI.
②		CLI for LabVIEW exit code that the VI returns.
③		Output or error information to include in the CLI for LabVIEW console or log file.

The following table lists the available arguments of this operation. Required arguments are in bold.

Argument	Description	Default
-VIPath	Path to the VI to run in LabVIEW. If the connector pane of the VI does not match the predefined interface, the CLI for LabVIEW returns an error.	—
< Command line arguments for the VI >	Elements of the input array of the VI. Separate elements with spaces.	—

To run this operation in the CLI for LabVIEW, execute a command with the following syntax:

```
LabVIEWCLI -OperationName RunVI -VIPath < VI path > < first element of array > < second element of array >...
```

For example, `LabVIEWCLI -OperationName RunVI -VIPath "C:\Users\inlabview.APAC\Documents\DemoProject\RunVI.vi" 101 202`

CloseLabVIEW

Closes LabVIEW without any prompts.



NOTE

Any unsaved changes will be lost if a VI is open when you run this operation. Make sure you save all your changes before running this operation.

This operation does not have any arguments.

To run this operation in the CLI for LabVIEW, execute the following command:

```
LabVIEWCLI -OperationName CloseLabVIEW
```

CreateComparisonReport

Compares two specified VIs and outputs an HTML, XML, Word, or text report containing the results of the comparison operation.

The following table lists the available arguments of this operation. Required arguments are in bold.

Argument	Description	Default
-vi1	Name or path of the first VI you are comparing.	—
-vi2	Name or path of the second VI you are comparing.	—

Argument	Description	Default
-reportType	Desired file type of the output report. This argument supports the following values: - HTML – produces an HTML format report with external images and style file. - HTMLSingleFile – produces an HTML form report with embedded images and style definition. - MicrosoftWord – produces a report in Microsoft Word format.  NOTE Microsoft Word must be installed on your machine for this option to function. - PlainText – produces a report in plain text. - XML – produces a report in XML format with external images. If this argument is not specified, it is inferred from the value provided for -reportPath.	HTMLSingleFile
-reportPath	Name or path of the output report.	—
-o	Overwrites existing reports and supporting files in the specified report directory if they exist.	—
-c	Creates a new directory for the specified report path if it does not exist.	—
-nofp	Excludes front panel differences from the comparison report.	—
-nofppos	Excludes front panel position differences from the comparison report.	—
-nobd	Excludes block diagram differences from the comparison report.	—
-nobdcosm	Excludes block diagram cosmetic differences from the comparison report.	—
-noattr	Excludes VI attribute differences from the comparison report.	—

Argument	Description	Default
-d	Compares the specified VIs without including dependencies.	—

To run this operation in the CLI for LabVIEW, execute a command with the following syntax:

```
LabVIEWCLI -OperationName CreateComparisonReport -vi1 < name or path of first VI > -vi2 < name or path of second VI > -reportType < file type of report > -reportPath < name or path of output report >
```

For example, `LabVIEWCLI -OperationName CreateComparisonReport -vi1 "firstVI.vi" -vi2 "secondVI.vi" -reportType "HTMLSingleFile" -reportPath "report.html"`

RunVIAalyzer

Runs the specified VI analyzer task in the LabVIEW VI Analyzer Toolkit and saves the test report to the specified location.

The following table lists the available arguments of this operation. Required arguments are in bold.

Argument	Description	Default
-ConfigPath	Path to the configuration file that contains VI Analyzer task settings to use in the analysis. You can use a configuration file you saved through the VI Analyzer or the VI Analyzer VIs. Alternatively, you can specify a VI, folder, or LLB to analyze. If you specify an item other than a configuration file, the VI runs all VI Analyzer tests on the specified item.	—
-ReportPath	Path to the report file.	—
-ConfigPassword	Password of the configuration file, if any.	Empty string
-ReportSaveType	Format of the report or results file. The value for this argument must be one of the following: - ASCII - HTML - RSL File	ASCII
-ResultsPath	Path to the results file. If the path is an empty string, LabVIEW ignores this argument.  NOTE This argument is case sensitive.	—

To run this operation in the CLI for LabVIEW, execute a command with the following syntax:

```
LabVIEWCLI -OperationName RunVIAAnalyzer -ConfigPath < path to configuration file > -ReportPath < path to report > -ReportSaveType < file type of report > -ConfigPassword < password of configuration file >
```

```
For example, LabVIEWCLI -OperationName RunVIAAnalyzer -ConfigPath "C:\temp\test.viancfg" -ReportPath "C:\temp\output.html" -ReportSaveType "HTML" -ConfigPassword "abc"
```

RunUnitTests

Runs tests on the specified files in the LabVIEW Unit Test Framework Toolkit and saves the JUnit file to the specified location.



NOTE

To run this operation, you must install the UTF JUnit Report library using the JKI VI Package Manager (VIPM) software.

The following table lists the available arguments of this operation. Required arguments are in bold.

Argument	Description	Default
-ProjectPath	Path to the project file.	—
-JUnitReportPath	Path to the output JUnit file.	—

To run this operation in the CLI for LabVIEW, execute a command with the following syntax:

```
LabVIEWCLI -OperationName RunUnitTests -ProjectPath < path to project file > -JUnitReportPath < path to output JUnit file >
```

```
For example, LabVIEWCLI -OperationName RunUnitTests -ProjectPath "C:\temp\test.lvproj" -JUnitReportPath "C:\temp\test.xml"
```



RELATED INFORMATION

[Comparing VIs using the LabVIEW Command Line Interface](#) on page 967

You can use the LabVIEW Command Line Interface (CLI) to programmatically compare two VIs and generate a comparison report listing the differences between the files.

Windows Embedded Standard Targets

Working with Windows Embedded Standard Targets

You develop LabVIEW VIs that run on Windows Embedded Standard targets by adding a target to a LabVIEW project. When you run the VI from the host computer, or host VI, the VI compiles and runs on the target. The host VI accesses the data on the target and displays the data on the front panel.

Unsupported LabVIEW Features

During remote development, certain LabVIEW features are not supported for VIs that run on Windows Embedded Standard targets. If you attempt to run on a Windows Embedded Standard target a VI that has unsupported functionality, the VI might still execute. However, the unsupported functions do not work and may return standard LabVIEW error codes.

The following features are not supported on Windows Embedded Standard targets during remote development.

- Subpanel controls
- Debugging reentrant VIs
- XControls

Troubleshooting Guide

Use the following table to find solutions when using LabVIEW Remote Development Target Support and developing applications on a Windows Embedded Standard target.

Issue	Solution
I cannot discover my target in Measurement & Automation Explorer (MAX) on the host computer.	On the host computer, MAX does not support discovering Windows Embedded Standard targets.
I cannot discover the target in the LabVIEW Development System using the Add Targets and Devices dialog box.	Ensure that the target meets the requirements necessary to discover a device for remote development: • A user with administrator privileges is logged into the target. • LabVIEW Remote Development is enabled on the target. • (Touch Panel devices) A device type is selected on the target.

Issue	Solution
I cannot connect to the target in a LabVIEW project, although I can discover the target using the Add Targets and Devices dialog box.	Ensure that the version of the LabVIEW Run-Time Engine on the target matches the version of the LabVIEW Development System on the host computer.
Can I configure remote development settings for my target from the host computer?	From the host computer, you can configure remote development settings by completing the following steps: 1. On the host computer, open a Web browser and enter the following URL to access the NI Web-based Configuration & Monitoring application: <code>http://localhost:3582</code>. 2. Click the Network Browser button on the left and double-click the row that contains your target. A separate NI Web-based Configuration & Monitoring application specific to your target opens in a Web browser. 3. Click Login in the top-right corner of the page. 4. Enter Admin in the User name field. 5. Leave the Password field blank. The default password is blank. Set the password to establish security. 6. Click OK. 7. On the System Configuration page, configure the remote development settings for your target. 8. Click Save to save changes. 9. Click Restart in the top-right corner of the page to restart the target. You must log into the target for changes to take effect.

Troubleshooting Windows Embedded Standard Target Connection Errors

If you encounter difficulties while connecting to the Windows Embedded Standard target, resolve the following possible issues when troubleshooting. These solutions require you to log into the target to make changes. You must restart and log into the target for changes to take effect.

Issue	Solution
Windows Firewall blocked the LabVIEW Remote Development application.	You must authorize the LabVIEW Remote Development application to allow access through the Windows Firewall.
LabVIEW Remote Development is not enabled on the target.	Launch MAX and select My System in the configuration tree. In the Remote Development Settings section of the System Configuration page, ensure that LabVIEW Remote Development contains a checkmark.

Issue	Solution
The version and bitness of the LabVIEW Run-Time Engine enabled for remote development on the target does not match the version and bitness of the LabVIEW Development System on the host computer.	Launch MAX and select My System in the configuration tree. In the Remote Development Settings section of the System Configuration page, refer to the Run-Time Engine Version .
The LabVIEW Remote Development application is not running on the target.	Launch the Windows Task Manager. In the Processes page, verify that the list of processes contains <code>niLVRemoteDev.exe</code> . If the list does not contain <code>niLVRemoteDev.exe</code> , you must restart the LabVIEW Remote Development application.

Related Information

[Adding Targets to a LabVIEW Project](#)

[Running VIs on Windows Embedded Standard Targets](#)

[Deploying Installers to Windows Embedded Standard Targets](#)

Running VIs on Windows Embedded Standard Targets

You can add VIs and dependent files under a Windows Embedded Standard target in the Project Explorer window. When you run a VI under a target, LabVIEW deploys the VI and the target settings to the Windows Embedded Standard target.

You can view the front panel of a VI running on the Windows Embedded Standard target if you connect the target to a monitor. The front panel on the target mirrors the front panel on the host computer. However, control and appearance of the front panel on the host and target have the following caveats.

- When the host connects to the target, the target does not have control of the front panel. For example, you cannot resize, minimize, maximize, or move the front panel on the target. If you attempt to gain control of the user interface by other protocols, such as remote front panels, the attempt is denied. Ensure the location of the front panel does not truncate the front panel when you connect to the target.
- When the host connects to the target, the front panel of the target displays **Remotely Controlled** in the window title and as a watermark in the bottom-right corner.
- If the host disconnects from the target, the target gains control of the user interface on the target and the VI operates like a local VI. When the host reconnects to the target, the host regains control of the user interface and updates its control and indicator values with values made by the target. The host does not update its user

interface with changes, such as front panel window size and position, made by the target.

- When the host connects to the target and the VI stops running, the front panel on the target remains open. If the front panel on the host computer closes, the front panel on the target also closes. If the VI is running and you disconnect from the target on the host computer, the VI continues to run on the target.
- With a few exceptions, all dialog boxes appear on only the host computer or only the target depending on which device has control of the user interface. For VI-based dialog boxes, the dialog boxes appear on both the host computer and target. When both the host computer and target display dialog boxes, only the host computer has control of the user interface.
- For file dialog boxes and dialog boxes invoked by executing a call library node, such as a Windows or custom DLL, the dialog boxes always appear on the target. For these dialog boxes, the host computer displays a message to notify you of the dialog box activity on the target. The user interface on the host is locked until you interact with the dialog box on the target.

Related Information

[Working with Windows Embedded Standard Targets](#)

[Deploying Installers to Windows Embedded Standard Targets](#)

Deploying Installers to Windows Embedded Standard Targets

After you develop a VI on the host computer, you can build the VI into an executable application. You also can build an installer that includes the executable application or other files and deploy or run the installer on a Windows Embedded Standard target. If you build an installer, include any components users need to run your application.



NOTE

To build a stand-alone application or installer, you must have the Application Builder installed. The LabVIEW Professional Development System includes the Application Builder. If you use another LabVIEW development system, you can purchase the Application Builder separately by visiting the National Instruments website.

To build an application or installer, you first must create a build specification. A build specification contains all the settings for the build, such as files to include, directories to create, and settings for VIs. You can create a build specification at any time by right-clicking **Build Specifications** under the Windows Embedded Standard target in the [Project Explorer](#) window and selecting **New»Application (EXE)** or **New»Installer** from the shortcut menu.

After creating the build specification, right-click the build specification in the [Project Explorer](#) window and select one of the following options from the shortcut menu:

- **Build** —Builds the build specification, such as an application or installer. This option does not deploy the installer to the target automatically.
- **Deploy** —Builds the installer and deploys the installer to the target. This option does not run the installer on the target automatically.
- **Install** —Builds, deploys, and runs the installer on the target to the location specified in the build specification.

Related Information

Developing and Distributing an Application

Creating Build Specifications

Using Build Specifications

Block Diagram within LabVIEW

Designing the Block Diagram

Use the following guidelines to design block diagrams:

- Use a left-to-right and top-to-bottom layout. Although the positions of block diagram elements do not determine execution order, avoid wiring from right to left to keep the block diagram organized and easy to understand. Only wires and structures determine execution order.
- Avoid creating a block diagram that occupies more than one or two screens. If a block diagram becomes large and complex, it can be difficult to understand or debug.
- Decide if you can reuse some components of the block diagram in other VIs or if a section of the block diagram works together as a logical component. If so, divide the block diagram into subVIs that perform specific tasks. Using subVIs helps you manage changes and debug the block diagrams quickly.
- Use the error handling VIs, functions, and parameters to manage errors on the block diagram.
- Improve the appearance of the block diagram by wiring efficiently. Poor wire organization might not produce errors, but it can make the block diagram difficult to read and debug or make the VI appear to do things it does not do.
- Avoid wiring under a structure border or between overlapped objects, because LabVIEW might hide some segments of the resulting wire.
- Avoid placing objects on top of wires. Placing a terminal or icon on top of a wire makes it appear as if a connection exists when it does not.
- Use labels to document code on the block diagram.
- To increase the space between tightly grouped objects, press <Ctrl> and drag in the direction you want to add space. (macOS) Press <Option>. To reduce space between scattered objects, press <Ctrl-Alt> and drag in the direction you want to decrease

the space. (macOS) Press <Option-Ctrl>. The objects move in real time as you drag the mouse. If the direction you drag is primarily vertical or horizontal, the operation snaps to the dominant direction.

Refer to the LabVIEW Style Checklist for more tips on how to make block diagrams easy to read and manage.



NOTE

You can reroute all existing wires and rearrange all existing objects on the block diagram automatically.



RELATED INFORMATION

[Automatically Cleaning Up the Block Diagram](#) on page 886

[Creating Modular Code](#) on page 203

[Handling Errors](#) on page 338

[Labeling](#) on page 1363

[LabVIEW Wiki Style Guide](#)

Using Wires to Link Block Diagram Objects

You transfer data among block diagram objects through wires. Each wire has a single data source, but you can wire it to many VIs and functions that read the data, similar to passing required parameters in text-based programming languages. You must wire all required block diagram terminals. Otherwise, the VI is *broken* and will not run. Display the Context Help window to see which terminals a block diagram node requires. The labels of *required terminals* appear bold in the **Context Help** window.

Wire Appearance and Structure

Wires are different colors, styles, and thicknesses depending on their data types, similar to how the color and symbol of a terminal indicate the data type of the corresponding control or indicator. A broken wire appears as a dashed black line with a red X in the middle. Broken wires occur for a variety of reasons, such as when you try to wire two objects with incompatible data types. The arrows on either side of the red X on the broken wire indicate the direction of the data flow, and the color of the arrows indicate the data type of the data flowing through the wire.

Refer to the [control and indicator data types table](#) for more information about data types.

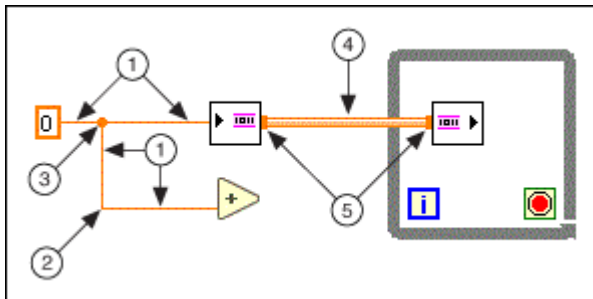
Wire stubs are the truncated wires that appear next to unwired terminals when you move the Wiring tool over a VI or function. They indicate the data type of each terminal. A tip strip also appears, listing the name of the terminal. After you wire a terminal, the wire stub for that terminal does not appear when you move the Wiring tool over its node.

A wire segment is a single horizontal or vertical piece of wire. A bend in a wire is where two segments join. The point at which two or more wire segments join is a junction. A wire

branch contains all the wire segments from junction to junction, terminal to junction, or terminal to terminal if there are no junctions in between.

Most wires represent synchronous communication from one node to another. Channel wires are a kind of wire that represents asynchronous communication. Channel wires have an off-center stripe of the data color faded out on one side. They do not draw tunnels as they enter or leave structures. Channel wires draw small blocks called flanges on the tips of node terminals where the wire connects to the terminal.

The following illustration shows a wire segment, bend, junction, a channel wire, and flange.



1	Segment
2	Bend
3	Junction
4	Channel Wire
5	Flange

Wiring Objects

Use the Wiring tool to manually connect the terminals on one block diagram node to the terminals on another block diagram node. When you move the Wiring tool over a terminal, the terminal blinks. When you move the Wiring tool over a VI or function terminal, a tip strip also appears, listing the name of the terminal. Wiring to the terminal might create a broken wire. You must correct the *broken wire* before you can run the VI.

Use the Context Help window to determine exactly where to connect wires. When you move the cursor over a VI or function, the **Context Help** window lists each terminal of the VI or function.



NOTE

Click the **Show Optional Terminals and Full Path** button in the Context Help window to display the optional terminals of the VI or function.

You also can use the Context Help window to determine which types of data to wire to the terminals of objects. When you move the cursor over the terminal of a VI or function, the **Terminal Data Type** field of the **Context Help** window describes which data type the

terminal accepts. If the terminal accepts multiple data types, the **Terminal Data Type** field describes the default data type for the terminal.



Note If you wire two objects of different data types, and the types are similar enough for LabVIEW to coerce the data types to match, LabVIEW places a **coercion dot** on a block diagram node to alert you that LabVIEW coerced the value passed into the node to a different representation. In **certain coercions**, coercion dots may lead to an increase in memory usage and execution time and a decrease in accuracy of results.

Bending Wires

While you are wiring a terminal, bend the wire at a 90 degree angle once by moving the cursor in either a vertical or horizontal direction. To bend a wire in multiple directions, click the mouse button to set the wire and then move the cursor in the new direction. You can repeatedly set the wire and move it in new directions.

Undoing Wires

To undo the last point where you set the wire, press the <Shift> key and click anywhere on the block diagram. (macOS) Press the <Option> key and click. (Linux) Click the middle mouse button. To abort the entire wiring operation, right-click anywhere on the block diagram or press the <Esc> key.

To swap wire positions on a function with two inputs without manually deleting and replacing the wires, press the <Ctrl> key while clicking one of the inputs.



Caution Crossing **wires** can clutter a block diagram and make the block diagram difficult to debug.

Automatically Wiring Objects

As you move a selected object close to other objects on the block diagram, LabVIEW draws temporary wires to show you valid connections. When you release the mouse button to place the object on the block diagram, LabVIEW automatically connects the wires. You also can automatically wire objects already on the block diagram. LabVIEW connects the terminals that best match and does not connect the terminals that do not match.

Toggle automatic wiring by pressing the spacebar while you move an object using the Positioning tool.

By default, automatic wiring is enabled when you select an object from the Functions palette or when you copy an object already on the block diagram by pressing the <Ctrl> key and dragging the object. Automatic wiring is disabled by default when you use the Positioning tool to move an object already on the block diagram.

(macOS) Press the <Option> key. (Linux) Press the <Alt> key.

You can disable automatic wiring by selecting **Tools»Options** and selecting **Block Diagram** from the **Category** list.

Routing Wires

LabVIEW automatically finds a route for a wire as you wire it. LabVIEW routes a wire around existing objects on the block diagram, such as loops and structures. LabVIEW also routes a wire to decrease the number of bends in the wire. When possible, automatically routed wires from control terminals exit the right side of the terminal, and automatically routed wires to indicator terminals enter the left side of the terminal.

To automatically route an existing wire, right-click the wire and select **Clean Up Wire** from the shortcut menu. You can *reroute all existing wires and rearrange all existing objects* on the block diagram automatically.

Press the <A> key after you start a wire to temporarily disable automatic wire routing and route a wire manually. Press the <A> key again to enable automatic wire routing for the wire. After you end the wire, LabVIEW enables automatic wire routing again. You also can temporarily disable automatic routing after you click to start or set a wire by holding down the mouse button while you wire to another terminal or set point and then releasing the mouse button. After you release the mouse button, LabVIEW enables automatic wire routing again.

You can disable automatic wire routing for all new wires by selecting **Tools»Options**, selecting **Block Diagram** from the **Category** list, and removing the checkmark from the **Enable automatic wire routing** checkbox.

If you disable automatic wire routing, you can wire terminals vertically or horizontally depending on the direction in which you first move the Wiring tool. The wire connects to the center of the terminal, regardless of where you click the terminal. After you click the terminal, press the spacebar to switch between the horizontal and vertical direction.

You also can press the spacebar to switch between the horizontal and vertical direction if automatic wire routing is enabled. If LabVIEW finds a route for the wire in the new direction, the wire switches to that direction.

Selecting Wires

Select wires by using the Positioning tool to single-click, double-click, or triple-click them. Single-clicking a wire selects one segment of the wire. Double-clicking a wire selects a wire branch. Triple-clicking a wire selects the entire wire.

Labeling Wires

You can use an *owned label* to label a long wire and identify its use. To create an owned label on a wire, right-click the wire and select **Visible Items»Label** from the shortcut menu. Labeling wires is useful for wires coming from shift registers and for long wires that span the entire block diagram. You can move an owned label anywhere along the wire. You cannot *lock* an owned label to a wire.

Correcting Broken Wires

A broken wire appears as a dashed black line with a red X in the middle -  -. Broken wires occur for a variety of reasons, such as when you try to wire two objects with incompatible data types. Move the Wiring tool over a broken wire to display a tip strip that describes why the wire is broken. This information also appears in the Context Help window when you move the Wiring tool over a broken wire. Right-click the wire and select **List Errors** from the shortcut menu to display the Error list window. Click the **Help** button to display more information about why the wire is broken.

Triple-click the wire with the Positioning tool and press the <Delete> key to remove a broken wire. You also can right-click the wire and select from shortcut menu options such as **Delete Wire Branch**, **Create Wire Branch**, **Remove Loose Ends**, **Clean Up Wire**, **Change to Control**, **Change to Indicator**, **Enable Indexing at Source**, and **Disable Indexing at Source**. These options change depending on the reason for the broken wire.

To remove all broken wires, select **Edit»Remove Broken Wires** or press the <Ctrl-B> keys. (macOS) Press the <Command-B> keys. (Linux) Press the <Meta-B> keys. You also can select a structure or area of the block diagram from which you want to remove broken wires and press the <Ctrl-B> keys to remove broken wires from only the selected area.



Caution Use caution when removing all broken wires. Sometimes a wire appears broken because you are not finished wiring the block diagram.

You also can break a wire if you delete a segment of the wire. Reconnect the wire segment to fix the broken wire.



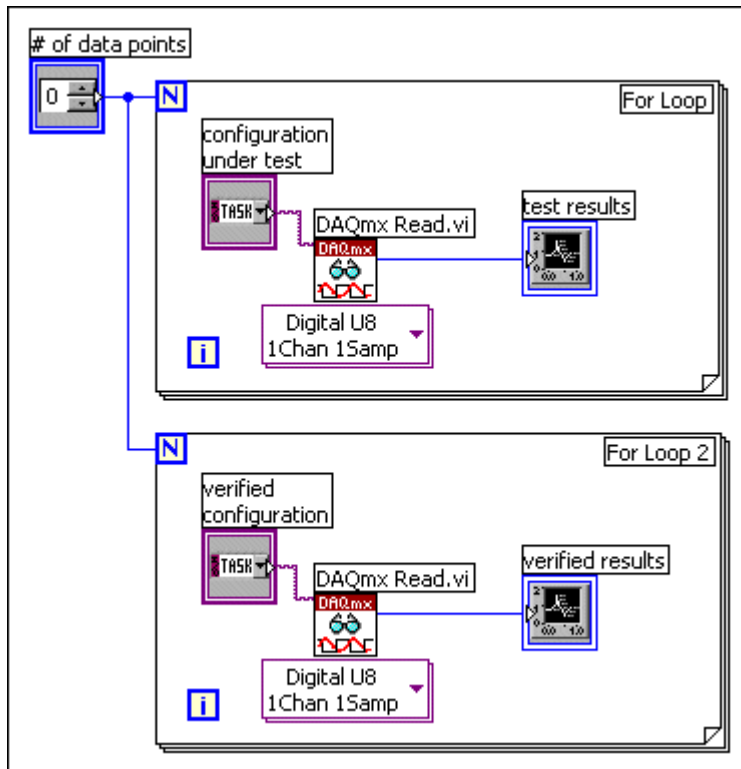
Note If you delete a segment of a wire with an *owned label*, the broken wire retains the label. If you connect two broken wires that have owned labels, the label of the wire connected to the source terminal becomes the label of the new wire.

Block Diagram Data Flow

LabVIEW follows a dataflow model for running VIs. A block diagram node executes when it receives all required inputs. When a node executes, it produces output data and passes the data to the next node in the dataflow path. The movement of data through the nodes determines the execution order of the VIs and functions on the block diagram.

Visual Basic, C++, JAVA, and most other text-based programming languages follow a control flow model of program execution. In control flow, the sequential order of program elements determines the execution order of a program.

In LabVIEW, the flow of data rather than the sequential order of commands determines the execution order of block diagram elements. Therefore, you can create block diagrams that have simultaneous operations. For example, you can run two For Loops simultaneously and display the results on the front panel, as shown in the following block diagram.



LabVIEW is a **multitasking and multithreaded** system, running multiple execution threads and multiple VIs simultaneously.

Data Dependency and Artificial Data Dependency

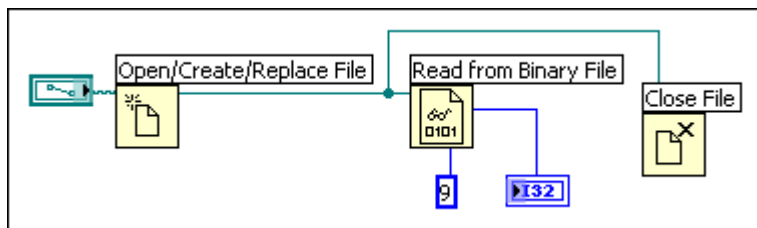
The control flow model of execution is instruction driven. Dataflow execution is data driven, or data dependent. A node that receives data from another node always executes after the other node completes execution.

Block diagram nodes not connected by wires can execute in any order. You can use **flow-through parameters** to control execution order when natural data dependency does not exist. You can use a **sequence structure** to control execution order when flow-through parameters are not available.

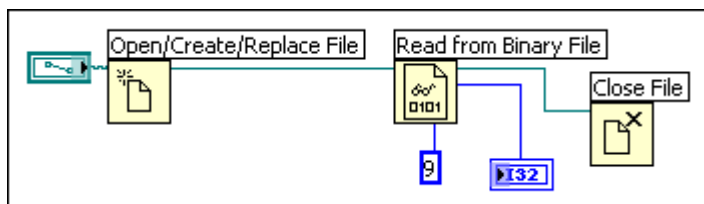
Missing Data Dependencies

Do not assume left-to-right or top-to-bottom execution when no data dependency exists. Make sure you explicitly define the sequence of events when necessary by wiring the dataflow.

In the following block diagram, no dependency exists between the Read from Binary File function and the Close File function because the Read from Binary File function is not wired to the Close File function. This example might not work as expected because there is no way to determine which function runs first. If the Close File function runs first, the Read from Binary File function does not work.



The following block diagram establishes a dependency by wiring an output of the Read from Binary File function to the Close File function. The Close File function does not run until it receives the output of the Read from Binary File function.



Flow-Through Parameters

Flow-through parameters, typically a refnum or [error cluster](#), return the same value as the corresponding input parameter. Use these parameters to control execution order when natural data dependency does not exist. By wiring the flow-through output of the first node you want to execute to the corresponding input of the next node you want to execute, you create an artificial data dependency. Without these flow-through parameters, you must use sequence structures to ensure that data operations take place in the order you want.

Data Flow and Managing Memory

Dataflow execution makes managing memory easier than the control flow model of execution. In LabVIEW, you do not allocate memory for variables or assign values to them. Instead, you create a block diagram with wires that represent the transition of data.

VIs and functions that generate data automatically allocate the memory for that data. When the VI or function no longer uses the data, LabVIEW deallocates the associated memory. When you add new data to an array or a string, LabVIEW allocates enough memory to manage the new data.

Because LabVIEW automatically handles memory management, you have less control over when memory is allocated or deallocated. If a VI works with large sets of data, you need to understand when [memory allocation](#) takes place. Understanding the principles involved can help you write VIs with significantly smaller memory requirements. [Minimizing memory usage](#) as you develop can help you increase the speed at which VIs run.

Coercion Dots

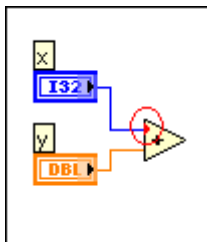
Coercion dots appear on block diagram nodes to alert you that LabVIEW converted the value passed into the node to a different representation. LabVIEW places a coercion dot on

the border of the terminal where the conversion takes place. Because VIs and functions can have many terminals, a coercion dot can appear inside an icon if you wire through one terminal to another terminal.

Red Coercion Dots

LabVIEW places red coercion dots on block diagram nodes when you connect a wire of one numeric type to a terminal of a different numeric type or when you connect any non-variant wire to a *variant* terminal.

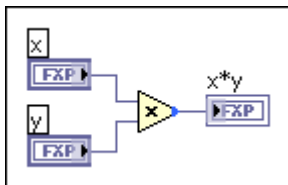
For example, LabVIEW places a red coercion dot on the Add function in the following illustration because the function requires two inputs of matching numeric data types, but receives a double-precision, floating-point number and an integer. The data types are similar enough that LabVIEW can coerce the integer and still perform the add operation.



Blue Coercion Dots

LabVIEW places blue coercion dots on the output terminals of Numeric functions when you manually *configure fixed-point output settings* for a function. The dot indicates that LabVIEW does not adjust the output data type automatically to avoid *overflow and rounding* errors.

The following block diagram shows a blue coercion dot.



Eliminating Coercion Dots to Maintain Accuracy and Optimize Memory Usage

Coercion dots can affect *memory usage* or overall performance of a VI in the following ways:

- **Type definition coercions** —If you wire an instance of a type definition to an object that does not link to the same type definition, the VI may generate inaccurate results.

- **Scalar numeric coercions** —If you manipulate different numeric types, LabVIEW coerces the data types to match. When this coercion occurs, the VI may generate inaccurate results.
- **Large array coercions** —If you coerce the data type of a very large array, the VI may use a large amount of memory and take longer to execute.

If you match data types, you can avoid unnecessary coercions.



TIP

When you move the cursor over a terminal with a coercion dot, the **Connected Wire Data Type** field of the Context Help window displays the data type of the wire connected to the VI or function. To eliminate coercion dots, match the **Connected Wire Data Type** to the **Terminal Data Type**.

Type Definition Coercions

When you wire an instance of a *type definition* to an object that does not link to that same type definition, LabVIEW places a coercion dot on the terminal receiving the data to warn you that if you edit the type definition, the VI may break or generate inaccurate data.

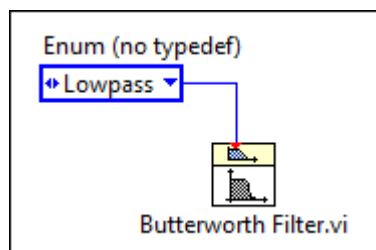


NOTE

If you wire a type definition to an object of the same data type, LabVIEW also coerces the type definition unless the object requires the same type definition. You can avoid the coercion by deleting the type definition and wiring the same parameter without a type definition to the object.

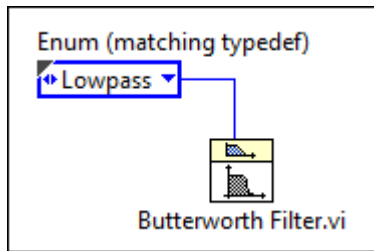
If you edit the type definition of an object wired to a cluster or an enum, the wire connecting to the cluster or enum breaks.

The following block diagram illustrates a type definition coercion that can affect the accuracy of the results and break a wire if you edit the type definition. In this example, the input on the subVI links to a type definition that defines the filter type.



LabVIEW places a coercion dot on the wired terminal of the subVI to indicate that the enum constant does not link to the same type definition.

The following block diagram illustrates a use of type definitions that maintains the accuracy of the results and does not break the wire if you edit the type definition.

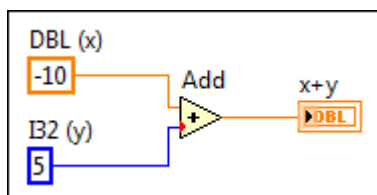


You can match the type definition of the input on the subVI by **copying** the object that links to the type definition or by **replacing** the object that does not link to the type definition with an instance of the type definition. You also can match the type definition of the input of the subVI by right-clicking the input terminal and selecting **Create»Constant** from the shortcut menu. The constant links to the type definition automatically.

Scalar Numeric Coercions

When manipulating simple scalar **numeric data types**, LabVIEW coerces to the data type that does not result in fractional data loss. If both data types are integers, LabVIEW coerces the data type with the lower maximum value to the data type with the higher maximum value.

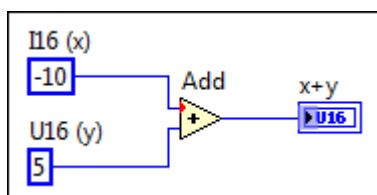
The following block diagram illustrates a coercion that does not affect the accuracy of the results.



In this example, when adding a double-precision, floating-point numeric to a long integer, LabVIEW coerces the long integer to a double-precision, floating-point numeric and adds the two values together. The coercion dot that LabVIEW places on the Add function does not change the result of the operation because LabVIEW coerces the data type with the lower maximum value to a more precise data type. However, you should match data types when possible.

If you want to make sure LabVIEW coerces the data to the type you want, change the data type of a control, constant, or indicator by right-clicking the object and selecting **Representation** from the shortcut menu.

The following block diagram illustrates a coercion that causes inaccurate results.



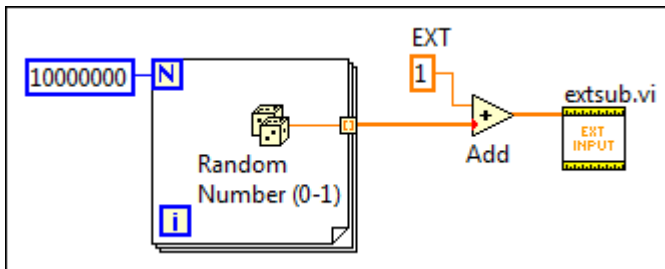
In this example, LabVIEW coerces the signed integer to an unsigned integer. Because LabVIEW attempts to coerce a negative value to a data type that does not support negative values, LabVIEW generates a value of 65,531 instead of -5. To avoid generating inaccurate data, *change the representation* of block diagram objects to matching data types.



Tip You can enable the Adapt to Source property on the output indicator to automatically change the indicator data type to the data type of the numeric input values. To enable the Adapt to Source property, right-click the indicator and select **Adapt to Source** from the shortcut menu.

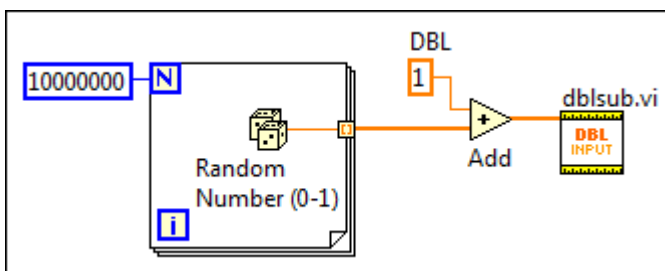
Large Array Coercions

When LabVIEW coerces data stored in very large arrays, the VI can use up a large amount of memory and take longer to execute. The following block diagram illustrates a VI that uses a large amount of memory because LabVIEW coerces the data type of the large array to the data type of the input of the subVI.



In this example, LabVIEW coerces a very large array to extended-precision, floating-point numbers to match the data type of the subVI. To minimize memory usage, match data types by right-clicking the control, constant, or indicator on the block diagram and selecting **Representation** from the shortcut menu. In the previous block diagram, changing the data type of the constant and the control on the block diagram of the subVI eliminates the need for LabVIEW to coerce any data.

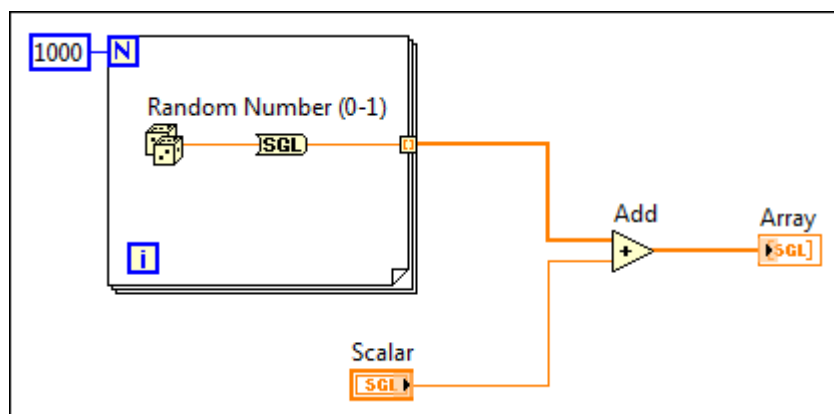
The following block diagram illustrates a VI that uses a more efficient amount of memory.



Tip To track and monitor how much memory a VI uses, you can use the Profile Performance and Memory window, which acquires and displays data about the execution time and memory usage of VIs.

Conversion Functions

You can use Conversion functions to eliminate coercion dots. In general, Conversion functions have the same effect as coercion dots but take up more space on the block diagram. However, when you use a Conversion function inside a loop that generates an array, you can avoid the conversion of a large data buffer because the conversion takes place before LabVIEW creates the array, as the following block diagram illustrates.



Changing the Color of Coercion Dots

To change the color of red coercion dots, use the Environment page of the Options dialog box. Remove the checkmark from the **Use default colors** checkbox in the **Colors** section and click the **Coercion Dots** color box to select a different color.

You cannot change the color of blue coercion dots.

Creating a Project in LabVIEW

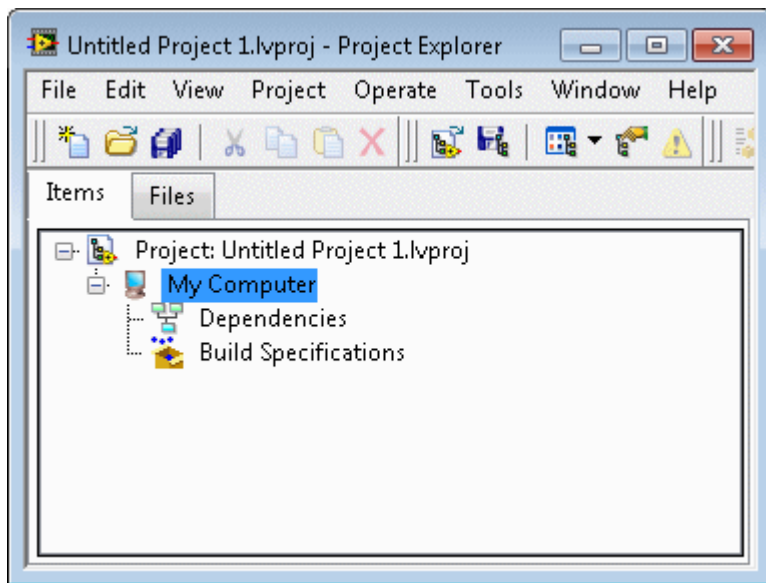
Managing a Project in LabVIEW

Projects in LabVIEW consist of VIs, files necessary for those VIs to run properly, and supplemental files such as documentation or related links. Use the Project Explorer window to manage projects in LabVIEW. In the **Project Explorer** window, you can use **folders** and **libraries** to group together items, and you can use a list of VI hierarchies called **Dependencies** to keep track of items a VI depends on.

Using the Project Explorer Window

Use the Project Explorer window to edit projects. To display the **Project Explorer** window, [create a new project](#). You also can select **File»Open Project** to open an existing project.

The Project Explorer window, shown as follows, includes two pages, the **Items** page and the **Files** page.



The **Items** page displays the project items as they exist in the project tree. The **Files** page displays the project items that have a corresponding file on disk. You can organize filenames and folders on this page. Project operations on the **Files** page both reflect and update the contents on disk. You can switch from one page to the other by right-clicking a folder or item under a target and selecting **Show in Items View** or **Show in Files View** from the shortcut menu.

The Project Explorer window includes the following items by default:

- **Project root** —Contains all other items in the Project Explorer window. This label on the project root includes the filename for the project.
 - **My Computer** —Represents the local computer as a target in the project.
 - **Dependencies** —Includes items that *VIs under a target require*, such as other VIs, shared libraries, and LabVIEW project libraries.
 - **Build Specifications** —Includes build configurations for source distributions and other types of builds available in LabVIEW toolkits and modules. If you have the LabVIEW Professional Development System or Application Builder installed, you can use **Build Specifications** to *configure the following* :
 - Stand-alone applications
 - Installers
 - .NET Framework assemblies
 - Packed libraries
 - Shared libraries
 - Source distributions
 - Web services
 - Zip files

You can [hide](#) the **Dependencies** and **Build Specifications** items in the Project Explorer window. If you hide either item you must display it again to access the item; for example, to build an application or shared library.

When you [add another target](#) to the project, LabVIEW creates an additional item in the Project Explorer window to represent the target. Each target also includes **Dependencies** and **Build Specifications**. You can [add files](#) under each target.

You can place a VI from the Project Explorer window on the block diagram of another open VI. Select the VI you want to use as a subVI from the **Project Explorer** window, and drag it to the block diagram of the other VI.

You also can use the Project properties and methods to configure and modify projects and the Project Explorer window programmatically.

Organizing Items in a Project

The following list describes some of the caveats and recommendations to consider when you organize items in a project:

- You can use a [sort option](#) to sort items within a project. Sort options apply automatically to items within the project and do not alter the organization of a project on disk. Use sort options to better organize and manage items within a project.
- Create a separate directory of files for each project you create. Organizing project files into separate directories makes it easier to identify files related to specific projects on disk.
- Directories on disk do not match the project structure of [virtual folders](#). After you add a directory on disk to a project in a virtual folder, LabVIEW does not update the folder in the project if you make changes to the directory on disk. Add a directory on disk to the project in an [auto-populated folder](#) to monitor and update changes on disk in the project.
- (Windows) If you are [building an installer](#), make sure you save the files that are in a project on the same drive as the project file `lvproj`. If some files are on a different drive, such as a network drive, links can break if you include the project in an installer.
- The structure of files in a source distribution does not have to match the structure in the Project Explorer window. You can specify a different structure when you [build a source distribution](#).
- [Dependencies](#) updates automatically when you add, remove, or save an item in the project. You cannot directly add files to or remove files from **Dependencies**. LabVIEW automatically tracks the hierarchy of every item you include in the project to ensure **Dependencies** includes files on which the item depends.
- When you open a project, the items that a VI calls dynamically do not appear under **Dependencies**. When you run the callers, the dynamically loaded items appear in the **Items in Memory** folder under **Dependencies**. You can add these items manually under a target to [manage them in a project](#).
- When you build an application, you can apply settings to an entire folder. Consider grouping all dynamic items in a folder under the target.

- The project can contain conflicts when two or more items with the same qualified name from different paths exist in the project. A yellow warning triangle appears on any conflicting items. Click the **Resolve Conflicts** button to open the Resolve Project Conflicts dialog box, in which you can view a summary of the project conflicts and [resolve the project conflicts](#).

Loading Project Items into Memory

When you open a project, LabVIEW searches the locations of project items on disk to populate the project tree. LabVIEW then loads the following libraries, including the libraries under **Dependencies**, into memory:

- [Project libraries](#) .lvlib
- [Packed project libraries](#) .lvlibp
- [Class libraries](#) .lvclass
- [XControls libraries](#) .xctl
- Statechart libraries .lvsc

When loading libraries, LabVIEW loads the members of a library into memory as follows.



NOTE

When loading the member libraries, LabVIEW loads all the member VIs of the member libraries.

Use the VI Hierarchy window to view VIs in memory. If a library loads a VI or type definition into memory, the library also loads the entire VI hierarchy, which may include VIs that are otherwise not loaded.

Working with Projects and Targets

Use projects to group together LabVIEW files and files not specific to LabVIEW, deploy or download files to targets, create build specifications, and manage large applications. When you create and save a project, LabVIEW creates a project file .lvproj, which includes references to files in the project, configuration information, deployment information, build information, and so on. Use source control in LabVIEW to share files and projects among multiple users.

You must use a project to build stand-alone applications and shared libraries. You also must use a project to work with a Windows Embedded Standard, RT, FPGA, or Touch Panel target. Refer to the specific module documentation for more information about using projects with these targets.

If you are using a project with an NI device driver, refer to the specific driver documentation for more information about using projects with drivers.

Refer to the `labview\examples\project` directory for examples of managing a LabVIEW project.

Find related examples



RELATED INFORMATION

[Managing a Project in LabVIEW](#) on page 365

[Building and Distributing Applications](#) on page 389

[Saving a Project](#) on page 814

[Using Source Control in LabVIEW](#) on page 384

[Using Folders in LabVIEW Projects](#) on page 369

[Using Libraries in LabVIEW Projects](#) on page 370

[Using Packed Project Libraries in LabVIEW Projects](#) on page 374

[Managing Dependencies in LabVIEW Projects](#) on page 376

[Resolving Project Conflicts](#) on page 378

[Adding Folders to a Project](#) on page 806

[Adding Items to a Project](#) on page 809

[Adding Targets to a Project \(Windows\)](#) on page 811

[Showing and Hiding Items in a Project](#) on page 816

[Viewing Files in a Project](#) on page 816

Using Folders in LabVIEW Projects

You can add two types of folders to a LabVIEW project, virtual folders and auto-populating folders. Virtual folders organize project items. Right-click a target and select **Add»Folder (Snapshot)** from the shortcut menu to [add a virtual folder](#) to the project. Auto-populating folders update in real time to reflect the contents of folders on disk. Contents of an auto-populating folder do not always match disk contents exactly, as in the case of project libraries. The **Items** page of the project displays project library `.lvlib` contents by library hierarchy and not by disk organization. For example, if a VI exists in a library file, the VI appears in the project under the library file. The VI does not appear in the project in the location on disk.

Auto-populating folders are visible only on the **Items** page of the Project Explorer window. You can view the disk contents of an auto-populating folder but you cannot perform disk operations such as renaming, reorganizing, and removing project items. To perform disk operations of items in an auto-populating folder, use the **Files** page of the **Project Explorer** window. The **Files** page displays the location of project folders on disk. Project operations on the **Files** page both update and reflect the contents of the folder on disk. Likewise, LabVIEW automatically updates the auto-populating folder in the project if you make changes to the folder on disk outside of LabVIEW.

The advantage of performing file operations on the **Files** page instead of using the file system outside of LabVIEW is that LabVIEW can update referencing files. When you move,

remove, or rename items on the **Files** page, LabVIEW updates all referencing items to reflect the change. Also, if a project is open and you rename a file on disk that is in an auto-populating folder, LabVIEW detects the change and the Undo File Rename? dialog box appears. You can either revert the rename operation or continue with the operation. However, items in the project that reference that file do not reference the new name and might result in conflicts because the rename occurred outside of LabVIEW.

**NOTE**

(macOS and Linux) LabVIEW does not support the use of symbolic links for files in auto-populating folders on macOS or Linux operating systems.

An auto-populating folder can contain a virtual folder only if the virtual folder is within a library hierarchy. A virtual folder does not represent files on disk. You can [convert](#) an existing virtual folder to an auto-populating folder by right-clicking a virtual folder on the **Items** page and selecting **Convert to Auto-populating Folder** from the shortcut menu. A file dialog appears prompting you to select a folder on disk. LabVIEW automatically renames the virtual folder to match the folder on disk and adds all contents of the folder on disk to the project. Items in the virtual folder that do not exist in the directory on disk move to the target.

To disable auto-population, right-click an auto-populating folder and select **Stop Auto-populating** from the shortcut menu. The auto-populating folder becomes a virtual folder.



Note Auto-populating folders can contain virtual items such as variables and hyperlinks because these items do not have a corresponding item on disk. You can rearrange virtual items such as variables and hyperlinks within an auto-populating folder because the movement does not result in a movement on disk. You also can move virtual items into and out of auto-populating folders.

Using Libraries in LabVIEW Projects

LabVIEW project libraries are collections of VIs, type definitions, [shared variables](#), palette files, and other files, including other project libraries. When you [create and save a new project library](#), LabVIEW creates a project library file `.lvlib`, which includes the properties of the project library and the references to files that the project library owns.

**NOTE**

[Packed project libraries](#) are project libraries that you build with the Application Builder into a single file. You cannot modify the files in a packed library unless you make changes to the files and [rebuild the packed library](#).

Project libraries are useful if you want to organize files into a single hierarchy of items, avoid potential VI name duplication, limit public access to certain files, limit editing permission for a collection of files, and [set a default palette file](#) for a group of VIs. You can drag items that a project library owns from the [Project Explorer](#) window to the block diagram or front panel.

You can view the structure of a project library from the Project Explorer window or in a stand-alone project library window. If you are not in the **Project Explorer** window, right-

click a project library file and select **Open** from the shortcut menu to open it in the project library window.



Note If the project library file you select is not the top-level project library file, the stand-alone window that opens is the project library window of the top-level project library. The project library file you select is in the contents tree of the top-level project library window.

Use project libraries to organize a virtual, logical hierarchy of items. A project library file does not contain the actual files it owns, unlike an LLB, which is a physical directory that contains VIs. Files that a project library owns still appear individually on disk in the directories where you saved them.

Use project libraries to qualify the names of VIs and other LabVIEW files. LabVIEW identifies VIs by filename, so LabVIEW unintentionally might load and reference a VI because the VI has the same filename as another VI, a problem known as cross-linking. When a VI is part of a project library, LabVIEW qualifies the VI name with the project library name to avoid cross-linking. A qualified name includes the filename and the qualified name of the owning project library.

For example, if you build a VI named `caller.vi` that includes a subVI named `init.vi` that `library1.lvlib` owns, you also can include a different subVI named `init.vi` that `library2.lvlib` owns and avoid cross-linking problems. The qualified names that LabVIEW records when you save `caller.vi` are `library1.lvlib:init.vi` and `library2.lvlib:init.vi` respectively.



Note

- Only one project library can own a specific VI. However, you can associate a file not specific to LabVIEW, such as a text file or HTML file, with multiple project libraries.
- Avoid adding a file to a new library if that file already belongs to a library. If you need to add a file that is already associated with a library to a new library, you must also add the original library for that file to the new library. Otherwise, the build output will place the file in the correct destination folder without actually adding the file to the new library.

You can specify version numbers in a project library to distinguish changes to the collection of files over time. Set version numbers from the General Settings page of the Project Library Properties dialog box and update the numbers periodically. The version number does not affect the project library name.



Caution You must right-click the project library and select **Save As or Rename** from the shortcut menu to display the Save As dialog box and rename project libraries. If you rename a project library outside LabVIEW, you might break the project library.

Use project libraries to limit access to certain types of files. You can [configure access](#) to items and folders in a project library as public or private to prevent users from accessing

certain items. When you set access for a folder as private, all VIs in that folder also have private access.

You can [limit editing permission](#) by locking or password-protecting project libraries. When you lock a project library, users cannot add or remove items and cannot view items that you [set as private](#). When you assign a password to a project library, users cannot add or remove items or edit project library properties without a password. Users can open the Project Library Properties dialog box, but all dialog box components except protection options are disabled. Users must unlock the project library or enter a password to enable the dialog box components. To unlock the project library, right-click the project library and select **Enter Password** from the shortcut menu.



Note When you enter a password for a library in a LabVIEW project that does not have a valid license specific to the version of LabVIEW you purchased, you cannot drag and drop items into or out of the library.



Note [Adding password protection](#) to a project library does not add password protection to the VIs it owns. You must [assign password protection to individual VIs](#) if you want to limit edits to the block diagrams and front panels. Consider using the same password for the project library and for the VIs the project library owns to avoid confusion.

You can [create project libraries from project folders](#). You also can [convert LLBs to project libraries](#). LLBs have different features and advantages than project libraries, so consider the ways in which you might use an LLB before you decide whether to convert it to a project library. You can include project library files in an LLB.

If you include a palette file .mnu in a project library, you can [set it as the default palette file](#) for all VIs that the project library owns. The default palette file for a project library is the palette available in the shortcut menu when you right-click a sub-VI call to any VI that the project library owns, just as source palettes are available in the shortcut menus for many VIs and functions placed on the block diagram from the Functions palette. However, unlike source palettes, the default palette file for a project library does not have to contain any VIs from the project library it belongs to. From the General Settings page of the Project Library Properties dialog box, select the palette file in the **Default Palette** ring control. You also can set the default palette file from the Item Settings page. Select the .mnu file in the **Contents** tree and place a checkmark in the **Default Palette** checkbox.

Project libraries that LabVIEW deploys to the [Shared Variable Engine](#) (SVE) are called processes. Making changes to a process or to a shared variable within a process without affecting the corresponding library is known as online configuration. Making changes to a library or to a shared variable in a library without affecting the corresponding process is offline configuration. If you use offline configuration to change a shared variable, you must redeploy the library to the SVE before you run any VI within the library so the SVE can update the shared variable in the process.

Organizing Project Libraries

You can create an organizational structure for files that a LabVIEW project library owns. A well-organized structure for project library items can make it easier for you to use [source](#)

control, avoid filename conflicts, and divide the project library into public and private access areas.

The following list describes some of the caveats and recommendations to consider when you organize project libraries and the files that the project libraries own.

- **Create each project library** within a separate LabVIEW project that contains only files related to that project library, including example files and the files you use to create and test the project library. Give the project and project library similar filenames. If a project library includes several separate areas of functionality, consider using **project sublibraries** for each area.
- Create a separate directory of files for each project library you create. You can include the files that the project library owns in the directory. If you include files for more than one project library in the same directory, conflicts might occur if you try to include VIs of the same name in different libraries. Organizing project library files into separate directories makes it easier to identify files related to specific project libraries on disk.
- You also can use a **sort option** to organize items within an unlocked project library.
- If you move files on disk that a project library owns, reopen and save the project library again to ensure that the project library links correctly to the moved items.
- (Windows) If you are **building an installer** that includes a project library, make sure you save the files that the project library owns on the same drive as the project library. If some files are on a different drive, such as a network drive, project library links will break if you include the project library in an installer.
- You can **resolve conflicts** between a library and the items claimed by a library by right-clicking a project root and selecting **Find Items Incorrectly Claimed by a Library** from the shortcut menu. Use this feature to find items that do not recognize a claim from a library.
- Determine which items in a project library you want to **set as private and which as public**. Users cannot use private VIs as subVIs in other VIs or applications. Public items provide the interface to the project library functionality and might include palette VIs, XControls, instrument drivers, and tools you want users to find and use. Private items might include support VIs, copyrighted files, or items you might want to edit later without taking the risk of breaking users' code. Consider the following recommendations.
 - Create a folder in the project library named `private`. From the Item Settings page of the Project Library Properties dialog box, configure the access settings as private for the folder. LabVIEW automatically sets as private any project library files you add to the `private` folder, so you do not have to configure access settings for individual VIs.
 - Assume that all project library files that are not in the `private` folder are public. You do not need to create a folder for public files.
 - You also can organize public and private items in a project library by creating folders for each functionality group within a project library and adding a private subfolder within each functionality group folder.

Project Sublibraries

Project sublibraries are project libraries that another project library owns. The settings in the owning project library do not affect access settings and editing permission for items within the project sublibrary. You can set the access of a project sublibrary file `.lvlib` as private within the owning project library, but when you edit the project sublibrary itself, items the sublibrary owns retain public or private access settings.

Project sublibraries are useful if you want to create a project library that includes separate areas of functionality. For example, if you are creating a project library of graphics tools, you might divide the two-dimensional and three-dimensional drawing tools into separate sublibraries.

Using Packed Project Libraries in LabVIEW Projects

LabVIEW packed project libraries are [project libraries](#) that package multiple files into a single file with a `.lvlibp` file extension. The top-level file of a packed library is a project library. By default, the packed library has the same name as the top-level project library.

You should [build a packed library from VIs in a LabVIEW project](#) for the following reasons:

- When you build a stand-alone application, build time is shorter if portions of the stand-alone application are in packed libraries. Because a packed library is a precompiled file, it does not need to compile when you build the stand-alone application, shortening build time.
- Because a packed library packages multiple files in a single file, fewer files deploy when you deploy the VIs in a packed library.
- VIs that call exported VIs in a packed library can [adapt to memory allocation changes](#) so that you do not have to recompile the caller VIs.

From the [Project Explorer](#) window, right-click **Build Specifications** and select **New»Packed Library** from the shortcut menu to display the Packed Library Properties dialog box and configure settings to [build a packed library](#).

A packed library contains only LabVIEW files. By default, LabVIEW saves non-LabVIEW files to the same destination directory as the packed library. Select **Support Directory** in the **Destinations** list and change the path in the **Destination path** text box to change where LabVIEW saves non-LabVIEW files.

To [replace a project library](#) with a packed library in a project, right-click the project library and select **Replace with** from the shortcut menu. When you select **Replace with**, all caller VIs of the project library in the LabVIEW project update with the qualified names of the packed library.

Exported VIs in a Packed Project Library

When you open a packed library, you see only the exported LabVIEW VIs. Exported VIs are VIs in project libraries with a public access scope or VIs in LabVIEW classes with a public or protected [access scope](#).



NOTE

To create a packed library from a LabVIEW class, add the LabVIEW class to a project library and [build a packed library](#) from the project library.

To find the access scope of a file in a packed library, right-click the file from the project library and select **Access Scope** from the shortcut menu. You cannot change the access scope of files in a packed library unless you make changes from the LabVIEW project and then rebuild the packed library.

Types of Packed Project Library Builds

When you build a packed library, you can create it as a release build or a debug build.

A release build is the default build for a packed library. Create a packed library as a release build when you do not want to include block diagrams of the VIs in the library. If a VI is not a dialog box, you also can remove front panels on the Source File Settings page of the Packed Library Properties dialog box.



Note You cannot debug VIs in a packed library release build because the VIs do not have block diagrams.

Create a packed library as a debug build when you want to include block diagrams of the VIs. Use a debug build to debug issues in the packed library. Because you cannot save changes to VIs in an existing packed library, save the changes to the original VIs and then rebuild the packed library.

To create a debug build, place a checkmark in the **Enable Debugging** checkbox on the Advanced page of the Packed Library Properties dialog box.

Compatibility of Packed Project Libraries and Caller VIs

The following changes to a VI in a packed library require caller VIs to recompile:

- Changing the connector pane pattern.
- Changing the dynamic dispatch terminal on the connector pane.
- Changing an input terminal to an output terminal or an output terminal to an input terminal on the connector pane.
- Changing an optional input to required on the connector pane.
- Changing the data types of inputs or outputs on the connector pane.
- Adding new terminals to the connector pane.

If a VI calls a packed project library compiled for one target and you open the VI on another target that has a different operating system, the packed project library fails to load.

If a stand-alone application, a shared library, or a packed library calls a VI in a packed library, include the called packed library in the same directory as the other [build specifications](#). When a build specification calls a VI in a packed library, you can replace the library with an updated version only if the connector panes of the updated packed library are compatible with the build specification.

If the connector panes of exported VIs in the packed library are not compatible with a VI in the build specification, rebuild the build specification with the caller VI after you rebuild the library.

When a change to a VI in a packed library impacts the compatibility of the connector panes, by default the caller VI adapts to the changes, which means that you do not need to recompile the VI that calls the library. If you want the caller VI to recompile, remove the checkmark from the **Callers adapt at run time to Exported VI connector pane state** checkbox on the Connector Pane State page of the Packed Library Properties dialog box.

(Real-Time Module) When a change to a VI in a packed library impacts the compatibility of the connector panes, by default you need to recompile the VI that calls the library. If you want the caller VI to run without recompiling when you change a VI in a packed library, place a checkmark in the **Callers adapt at run time to Exported VI connector pane state** checkbox on the Connector Pane State page of the Packed Library Properties dialog box.

Packed Project Library Properties

The packed library has the same properties as the top-level project library. Because you cannot modify a packed library without rebuilding, you cannot make changes to properties in the Project Library Properties dialog box. To display the **Project Library Properties** dialog box, open a packed library, right-click the `.lvlibp` file, and select **Properties** from the shortcut menu.

To update the Project Library Properties dialog box, make changes to the properties of the top-level project library and rebuild the packed library.



Note The Project Library Properties dialog box has different properties from the Packed Library Properties dialog box, with the exception of the **Version Number**. When you create a packed library, the **Version Number** on the Version Information page of the **Packed Library Properties** dialog box overrides the **Version Number** on the General Settings page of the **Project Library Properties** dialog box.

Managing Dependencies in LabVIEW Projects

Use **Dependencies** to view items that VIs under a target require, such as:

- Other VIs
- Shared libraries (DLLs)
- [LabVIEW project libraries](#)

Each target in a **LabVIEW project** includes **Dependencies**. LabVIEW automatically identifies the files required by each item in the project and includes these files in **Dependencies**. LabVIEW organizes **Dependencies** into three folders: `vi.lib`, `user.lib`, and `Items in Memory`.

Using the Dependencies List

You cannot add items directly to **Dependencies**. **Dependencies** updates automatically when you add, remove, or save an item in the project. For example, if you add a VI that includes a subVI to a target, LabVIEW adds the subVI to **Dependencies**. However, if you manually add the subVI under the target, the item does not appear under **Dependencies**. When you name or move dependent files, take precautions to *avoid incorrect dependencies*.

To view the current dependencies of a project, right-click **Dependencies** and select **Refresh** from the shortcut menu. You also can use the Refresh Dependencies method to refresh **Dependencies** programmatically.



Note If you previously hid **Dependencies** in the Project Explorer window, you must *display the item* again to access it in the **Project Explorer** window.

LabVIEW tracks subVIs recursively. LabVIEW does not track shared libraries (DLLs) recursively. For example, if `a.vi` calls `b.dll` statically and `b.dll` calls `c.dll` statically, LabVIEW only considers `b.dll` as a dependent item. To manage `c.dll` in the project, you must explicitly add `c.dll` under the target.

If a dependency is part of a project library, LabVIEW includes the entire project library under **Dependencies**.

You cannot create new items under **Dependencies**. You cannot drag items from other places in the Project Explorer window to **Dependencies**.

Including Dynamic Dependencies

When you open a project, the items that a VI calls dynamically do not appear under **Dependencies**. When you run the callers, the dynamically loaded items appear in the `Items in Memory` folder under **Dependencies**. You can add these items manually under a target to *manage them in a project*.

Because dynamically loaded files are not statically linked by any callers in the project, any action that changes the path to a dynamically loaded file can prevent the project from loading the file. To ensure that dynamically loaded files are in the correct location, follow these guidelines:

- Group the files in a separate folder.
- Refer to the files using relative paths.
- If you need to move a project or distribute an application, remember to include the folder in which you grouped dynamic dependencies.

Managing Shared Dependencies

If you create applications that share code, changing the code can inadvertently affect other callers. Observe the following guidelines when managing dependencies with shared code:

- Minimize changes to code shared by applications.
- If you need to change shared code, make local copies for testing and integration.
- Dedicate time to managing and integrating changes to shared code.
- Use [source code control](#).

Replacing Project Items

You can replace a VI, class, or library in a LabVIEW project with another item of the same type. Right-click a project item in the Project Explorer window and select **Replace with** from the shortcut menu to select a file to replace the original item. LabVIEW treats libraries and packaged project libraries as the same type when you replace these project items.

Replacing project items can affect callers in the project. After replacing a project item, LabVIEW updates callers to refer to the new project item instead of the original project item. LabVIEW does not change the original and replacement project items in addition to updating the callers.

Replacing Libraries

LabVIEW returns a warning when you replace a library in which at least one member is referenced by any callers out of the original library and the referenced member is missing from the replacement library. You can click **Accept** in the warning dialog box to continue replacing the library. However, the callers will break after you replace the library because LabVIEW removes the original library including the referenced item from the project.

The replacement operation does not affect callers that reference members in its own library.

Resolving Project Conflicts

Projects can contain items that conflict with other items in the project. A conflict is a potential cross-link that occurs when a LabVIEW project contains two or more items with the same qualified name under a single target. For example, if a VI calls a subVI of the same qualified name as an item already in the project from a different path, a conflict occurs because of the cross-link. Most conflicts exist because other items in the project [reference a conflicting item](#).

Finding Conflicting Items

The best way to determine if cross-linking exists is to view the full path to the item. Right-click the project root and select **View»Full Paths** from the shortcut menu to display the

Paths column and view the file paths that correspond to the project items. You either must rename or remove all but one item with the same qualified name from the project.

To view detailed information about existing conflicts, click the **Resolve Conflicts** button, shown as follows, to display the Resolve Project Conflicts dialog box.



You also can select **Project»Resolve Conflicts** from the project menu to display this dialog box or right-click a conflicting item and select **Resolve Conflicts** from the shortcut menu.

Right-click a specific conflicting item on the **Items** page and select **Find»Conflicts** to view all conflicting items in the Find Conflicts dialog box. If the item only conflicts with one other item, LabVIEW highlights the item in the Project Explorer window. You also can use the Find:Conflicts property to find conflicts in the project programmatically.



NOTE

A yellow warning triangle appears on any conflicting items in the Project Explorer window.

The Hierarchy Conflicts with Project dialog box appears when you try to load a file that conflicts with items already in the LabVIEW project. LabVIEW cannot open the file because items in its hierarchy have the same qualified name as items in the project or **Dependencies**. If a dependent item conflicts with another project item, removing the items that reference it removes the item from **Dependencies**.

LabVIEW scans the VI hierarchy when you open the block diagram of a VI and add a new subVI. The subVI does not load if any members of its hierarchy have the same qualified name but different paths as an item already in the project. The Add to Project and Update Dependencies dialog box appears. You can add the hierarchy or cancel the load and choose another subVI.

Removing Conflicting Items

You can remove caller VIs from the project to resolve conflicts when a hierarchy of VIs conflicts with the hierarchy of other contents in the project. Right-click a VI or type definition and select **Find»Callers** or **Find»SubVIs** from the shortcut menu to highlight the caller or subVI the item references in the Project Explorer window. If more than one caller or subVI exists in the project, the Find Callers or Find SubVIs dialog box appears. Right-click a project root or target and select **Find Items with No Callers** from the shortcut menu to display the Find Items with No Callers dialog box and find all top-level items. If no callers reference a conflicting subVI, remove the subVI from the project. You also can use the Find:Callers, Find:SubVIs, or Find:Items with No Callers properties to find callers, subVIs, and items with no callers programmatically.



Note Removing a conflicting subVI from the project might not resolve the conflict because other VIs in the project may still reference the conflicting subVI. The item is a conflicting item until you remove all callers that call the conflicting item from the project. Deleting an item that has callers from the project moves the item to **Dependencies**.

Renaming Conflicting Items

If you do not want to remove the conflicting item and you detect it has the same qualified name as another item in the project, you can rename the item or add the item to a project library.

Renaming the item loads the callers that reference the incorrect item path, renames the item, and saves the item and all callers. The callers reference the new name. If you do not save the callers, the original item appears under **Dependencies** because callers still reference the item.

When a VI is part of a [project library](#), LabVIEW qualifies the VI name with the project library name to avoid cross-linking. A qualified name includes the filename and the qualified name of the owning project library filename. The qualified name changes without changing the path or filename.

If a LabVIEW project library in memory conflicts with another project item, you must rename at least one conflicting library before loading. Right-click the library and select **Unload** from the shortcut menu. After LabVIEW unloads the library, you can reload the library and the VIs from the correct paths. If a library conflicts with another project item and is not in memory, you can right-click the library and select **Load** from the shortcut menu. After LabVIEW loads the library, you can edit the library or its contents and load VIs from the correct paths.

Redirecting Conflicting Items

When two or more items have the same qualified name, and only one item exists on disk, you can right-click a conflicting item and select **Replace with Item Found by Project** from the shortcut menu. LabVIEW updates the callers of the incorrect item to reference the item found on disk.

If you detect that one or more VIs refers to the wrong subVI, redirect all callers to reference a subVI with a different path. Right-click a conflicting VI in the Project Explorer window and select **Replace with** from the shortcut menu to choose the correct subVI on disk. Select a replacement file from the file dialog that appears. LabVIEW automatically updates all items that reference the incorrect path to reference the replacement. You also can load each VI that refers to a conflicting item. The Resolve Load Conflict dialog box appears. You can choose a specific caller VI to load.



Note LabVIEW dims **Replace with Item Found by Project** and **Replace with** if the item is a project library or a member of a project library.

Finding Missing Items

If you think that one or more VIs incorrectly refers to an item that LabVIEW cannot find, right-click the project root and select **Find Missing Items** from the shortcut menu. The Find Missing Items dialog box appears. This dialog box lists all items in the project that reference an item on disk that LabVIEW cannot find. When an item is not in the project but another item in the project is dependent on the missing item, the missing item appears in **Dependencies**.

Finding Incorrectly Claimed Items

You can locate and resolve conflicts between a library and the items a library claims in LabVIEW. To determine whether or not an item reciprocates a claim from a library, right-click the project root and select **Find Items Incorrectly Claimed by a Library** from the shortcut menu to display a list of the incorrectly claimed items in the Find Project Items dialog box. Use this dialog box to navigate to a specific item and remove it from the library or add it to the library.



Tip You also can access this feature by right-clicking a folder or library and selecting **Items Incorrectly Claimed by a Library** from the shortcut menu.

Adding Targets to a LabVIEW Project (Windows)

Use the Add Targets and Devices dialog box to [add a target or device](#) to a LabVIEW project. You must use a project to work with a Windows Embedded Standard, RT, FPGA, or Touch Panel target. Refer to the specific module documentation for more information about using projects with these targets.



NOTE

You must have a module or driver that supports targets installed to display the Add Targets and Devices dialog box.

Right-click the project root and select **New»Targets and Devices** from the shortcut menu to display the Add Targets and Devices dialog box. If a target in the project supports other targets, you also can right-click the target and select **New»Targets and Devices** from the shortcut menu to add a target under the existing target. For example, if you have an NI PCI device installed on a computer, you can add the device under **My Computer**.

You can [add items](#) under a target. You also can use a target folder to organize one or more targets in your LabVIEW project. To add a target folder to the project, right-click the project root in the Project Explorer window and select **New»Target Folder** from the shortcut menu. A target folder appears in the **Project Explorer** window.

Deploying Files to a Target

In the [Project Explorer](#) window, open and run a VI under a target to deploy files to the target. LabVIEW creates and deploys file types appropriate for the target, including source files, build files, object code, and hardware configuration files.



Note If you are [working with a Windows Embedded Standard target](#), you can deploy only a built installer to the target.

Using Palettes with Multiple Targets

LabVIEW loads a separate palette set for each target. If you have VIs open on multiple targets in a LabVIEW project, multiple palette sets exist in memory. When you switch

between targets, LabVIEW tries to map each pinned palette to the corresponding palette in the active target. If a pinned palette does not have a corresponding palette, LabVIEW displays the top-level palette for the active target.

To edit the palette set for a specific target, select **Tools»Advanced»Edit Palette Set** from a VI on the target whose palette set you want to modify.

Working with Application Instances

LabVIEW creates an application instance for each target in a [LabVIEW project](#). When you open a VI from the Project Explorer window, the VI opens in the application instance for the target. LabVIEW also creates a main application instance, which contains open VIs that are not part of a project and VIs that you did not open from a project. Additionally, LabVIEW loads [shared libraries](#) in a unique application instance. Opening a shared library in a unique application instance prevents naming conflicts with VIs in the shared library, and VIs outside of the shared library.

Use the application instance name that appears in the application instance shortcut menu on the bottom left corner of the front panel and block diagram windows to identify which application instance a VI belongs to.

If you [remove a VI](#) from the Project Explorer window when the VI is open, the open VI remains in the same application instance. You can [select an application instance](#) for a VI by using the application instance shortcut menu in the bottom left corner of the front panel and block diagram windows.

If you add an open VI to a project by selecting the VI icon in the upper right corner of a front panel or block diagram window and dragging the icon to a target in the Project Explorer window, or if you drag an item from one target to another in the **Project Explorer** window, the open VI is not part of the application instance for the target until you select an application instance.

You also can use the Application:Default:Application property to return the default application reference programmatically. Use the Application property to open the target application instance programmatically.

Editing VIs in Multiple Application Instances

You can open the same VI on disk in multiple application instances at the same time. For example, you can open the same VI in two different projects or from two different targets within a project.

If you edit a VI that is open in only one application instance and then open the VI in another application instance, the VI you just opened contains the latest changes from the edited instance of the VI. However, if you edit a VI that is already open in more than one application instance, LabVIEW does not automatically apply the changes to the VI in the other application instances. You cannot edit, run, or save the VI in the other application instances until the VI is the same in all application instances.

Use one of the following methods to make the VI the same in all application instances.

- Click the **Synchronize with Other Application Instances** button on the VI toolbar to apply changes to the VI in all application instances.
- Save the VI that contains the changes. When you save the VI, LabVIEW automatically applies the changes to the VI in all other application instances.
- Undo edits to the VI in the application instance that contains the changes.

**NOTE**

You cannot undo edits to the VI after you synchronize the VI.

You must use an application reference when you are using VI server properties and methods in one application instance, and you want to interact with a different application instance. It may not be obvious that VIs are running in multiple application instances. For example, each LabVIEW project has its own application instance. VIs running in `project1.lvproj` are running in a different application instance than VIs running in `project2.lvproj`. VIs not running in a project run in the main application instance. In addition, LabVIEW opens user VIs you place in the `labview\help`, `labview\project`, and `labview\wizard` directories in a private application instance. For example, if you place a VI in the `labview\project` directory and then launch this VI from the **Tools** menu, LabVIEW runs the VI in a private application instance (NI.LV.Dialog) that is separate from the main application instance.

You cannot edit a VI while it is running or reserved for execution in another application instance. If a VI is part of a *project library*, LabVIEW also temporarily locks the project library in all application instances when the VI runs. You cannot edit the project library while the VI is running. After the VI stops, LabVIEW unlocks the project library.

Planning and Designing a Project

Before you design a project, you might find it useful to review common development pitfalls and software engineering life cycle models.

Before you develop your own VIs, create a list of tasks your users will need to perform. Determine the user interface components and the number and type of controls and indicators you need for data analysis, displaying analysis results, and so on. Think about and discuss with prospective users or other team members how and when the user will need to access functions and features. Create sample front panels to show to prospective users or team members and determine if the front panel helps your users accomplish their tasks. Use this interactive process to refine the user interface as necessary.

Divide your application into logical pieces of manageable size. Begin with a high-level block diagram that includes the main components of your application. For example, the block diagram could include different sections for configuration, acquisition, analysis, data display, data logging, and error handling.

After you design the high-level block diagram, define the inputs and outputs. Then, design the *subVIs* that make up the main components of the high-level block diagram. Test subVIs as you create them. You can create higher level test routines, but catching errors in a small module is easier than testing a hierarchy of several VIs. You might find that the initial design of the high-level block diagram is incomplete. Using subVIs to accomplish low-level tasks makes it easier to modify or reorganize your application.

Select **Help»Find Examples** to launch the NI Example Finder. You can browse the example VIs to examine the block diagrams of VIs and to see examples of subVIs.

Select **File»Create Project** to launch the Create Project dialog box, which lists customizable templates and sample projects for common applications.

Use the Project Explorer window to manage [LabVIEW projects](#).

Designing Modular Components

Using modular programming helps you manage changes and debug the block diagram quickly. Create subVIs for common or frequent operations that you can reuse. Using subVIs makes the high-level block diagram easy to read, debug, understand, and maintain. You might find it useful to identify a design technique when you work on a project.

Designing Projects with Multiple Developers

If multiple developers work on the same project, define programming responsibilities, interfaces, and coding standards in the beginning to ensure the development process and the application work well together. Refer to the [LabVIEW Style Checklist](#) for more information about coding standards.

Keep master copies of the VIs on a single computer and institute a source control policy. You can use the LabVIEW Professional Development System with third-party source control providers so you can check out files, track changes, and [merge changes](#) from within LabVIEW.

Before editing VIs, recompile any VIs that were last edited on a different platform to improve load time during editing sessions. After you open the VIs, press the <Ctrl-Shift> keys while clicking the **Run** button to recompile all VIs in memory. (macOS) Press the <Option-Shift> keys while clicking the **Run** button. (Linux) Press the <Meta-Shift> keys while clicking the **Run** button. Then select **File»Save All** to save all VIs in memory.

Source Control in LabVIEW

Using Source Control in LabVIEW

You can use a source control provider to share files among multiple users, improve security and quality, and track changes to shared projects. Use LabVIEW with third-party source control providers so you can check out files and track changes from within LabVIEW. Select **Tools»Source Control** to access the common source control operations within LabVIEW. You can use source control with files in a LabVIEW project or with individual VIs.



NOTE

Source control is available only with the Professional Development System.

A source control provider helps you manage software projects and facilitates file sharing among multiple users and multiple software projects. The provider maintains a centralized master copy of project files. As you make changes, you update the master copy to reflect those changes, which makes it easy for any user to access the latest version of the project files. A source control provider also encourages code reuse by making all code easily accessible.

Source control, also known as source code control, helps improve security and quality. When you decide to modify a file, you [check out the file](#) to edit it. After you finish editing the file, you [check in the new version of the file](#), which becomes part of the master copy of the source control project. If you check in incorrect changes, most source control providers allow you to access previous versions of files so you can undo revisions.

A source control provider helps track changes to the software project. When you check in a file, the provider prompts you to describe the changes you made. The source control provider maintains this information so you clearly document the evolution of the project. You can view a revision history for each file in source control.



Tip You can use a [sort option](#) to organize items within a project. The sort option you select does not alter the organization of the project on disk, which allows you to merge and compare fewer changes when you submit the project to source control.

In addition to maintaining the source code, the source control provider can manage other aspects of a software project. For example, you can use a source control provider to track changes made to feature specifications and other documents.

After you [select the third-party source control provider](#) you want to use with LabVIEW, you can [configure LabVIEW settings](#) for that provider.

If you want to perform source control operations that are not available in LabVIEW, work directly from the source control provider for those specific functions.

Refer to the Software Configuration Management and LabVIEW article on ni.com for more information on how to manage and control source code.

Selecting a Source Control Provider

LabVIEW supports several third-party source control providers. Available source control operations in LabVIEW are the same regardless of which third-party provider you select. Specific support or functionality for each operation varies by provider. For example, you cannot create Perforce configuration files in LabVIEW.

In some cases, you might decide to use a specific source control provider because your company has standardized on that application. If not, you must decide which provider you want to use for managing your files. Consult the source control administrator at your company to find out if you should use a specific provider.

After you select and install a source control provider, you must [configure LabVIEW](#) to work with that provider. You can configure LabVIEW to work with only one source control provider at a time.

LabVIEW includes two source control integration interface types. On Windows, LabVIEW integrates with any source control provider that supports the Microsoft Source Code Control Interface. On non-Windows platforms, LabVIEW integrates with Perforce using a command line interface.

Refer to the Software Configuration Management and LabVIEW article at ni.com for more information on source code control integration with LabVIEW.

Source Control Operations in LabVIEW

After you [configure LabVIEW](#) to work with a third-party source control provider, you can perform source control operations on any file or folder of items in a LabVIEW project or on individual VIs. If you perform source control operations on a folder, all appropriate items within the hierarchy are affected. For example, if you add files to source control, LabVIEW adds only files within the folder that you have not yet added to source control. Access the following operations by selecting **Tools»Source Control** and selecting among the available options. Within a LabVIEW project, you also can use the Source Control toolbar buttons or right-click a file or folder in the [Project Explorer](#) window and select the option from the shortcut menu.

You also can [configure source control on individual LabVIEW projects](#) from the Source Control page of the Project Properties dialog box. Configure source control for individual LabVIEW projects if you want to use a different source control project than the one you specify for the LabVIEW environment, or if you do not want to use source control with a LabVIEW project. LabVIEW projects use the source control configuration you specify for the LabVIEW environment by default. Select **Project»Properties** or right-click the project root and select **Properties** from the shortcut menu to display the **Project Properties** dialog box. Select **Source Control** from the **Category** list to display the **Source Control** page.

Refer to the following caveats when you use source control:

- Consider [separating compiled code](#) from VIs and other file types that you store in source control. By default, when you edit a VI, LabVIEW automatically recompiles all VIs that call the changed VI, which introduces unsaved changes to the calling VIs. If the calling VIs are stored in source control, you must check them out and then check them back in. However, if you separate compiled code from the VIs within a [VI hierarchy](#), changes to one VI do not produce unsaved changes elsewhere in the VI hierarchy. This behavior allows you to check out only the VI you want to modify.
- If you use source control with VIs outside of a LabVIEW project, you cannot perform source control operations on project-specific items, such as project libraries `.lvlib` or projects `.lvproj`.
- If you attempt to perform source control operations on a VI in an LLB, LabVIEW performs the operations on the LLB that contains the VI, not on the VI itself. You cannot perform a source control operation on only one VI in an LLB.
- If a VI is [reentrant](#), you cannot perform source control operations on the clone of the source VI. LabVIEW dims the source control operation items in the **Tools»Source Control** menu of the clone VI.

You can complete the following source control operations in LabVIEW:

- **Get Latest Version** — *Copies the latest version of the selected file* from source control to the local directory to synchronize the two versions. The latest version of the file in source control overwrites the version in the local directory.
- **Check In** — *Checks the selected file into source control*. The version of the file in source control is updated to reflect the changes you made. If the file selected has unsaved changes, the Unsaved Files dialog box appears. You do not have to save the file to check it into source control.
- **Check Out** — *Checks out the selected file from source control*. If you try to edit a file in source control that you did not check out, LabVIEW prompts you to check out the file if you configured source control to enable the prompt.
- **Undo Check Out** — *Cancels a previous check-out operation* and restores the contents of the selected file to the previous version. Any changes you made to the file are lost.
- **Add to Source Control** — *Adds the selected file to source control*. LabVIEW prompts you to add any dependent files, such as subVIs, to source control if you configured source control to enable the prompt.

**NOTE**

This operation is available after you save a file.

- **Remove from Source Control** — *Removes the selected file from source control*.



Caution Be careful when you remove files from source control. Some source control providers delete the local directory copy of the file, all previous versions of the file that the provider maintains, and the history log for the file.

- **Show History** —Displays the source control history of the selected file. The history contains a record of changes to the file after it was added to source control. The history provides information about the previous versions of the file, such as file check-in dates and user actions. You cannot perform this operation on folders.
- **Show Differences** —Displays the differences between the local copy of the selected file and the version in source control. For text files, LabVIEW uses the default comparison tool of the source control provider. If you select a VI to compare, LabVIEW displays the results in the Differences dialog box. You cannot perform this operation on folders.

Note If LabVIEW and the third-party source control provider are not compatible, LabVIEW launches the default comparison tool of the third-party source control provider. Perforce SCM and Rational ClearCase are not compatible with the procedure LabVIEW uses to complete a graphical differencing of VIs.

- **Properties** —Displays the source control properties for the selected file, including its check-out status and modification dates. You cannot perform this operation on folders.
- **Refresh Status** —Updates the source control status of the files in the LabVIEW project or of the VI if you are working outside a LabVIEW project.
- **Run Source Control Client** —Launches the file management client of the source control provider.



Note The following source control operation appears in the shortcut menu when you right-click the project root.

- **Get Latest Version of All Files** — *Locates and retrieves the latest version of files* that are missing from the project or that are out-of-date. LabVIEW retrieves the latest version of any missing or out-of-date files from source control. Information about missing files that LabVIEW cannot retrieve appears in the Missing File Information dialog box.

Facilitating Source Control by Separating Compiled Code from VIs and Other File Types

By default, a VI contains two kinds of code: the graphical source code that you edit and a compiled version of this code that LabVIEW uses to run the VI. When you edit the source code of the VI, LabVIEW automatically recompiles the VI to reflect your changes in the compiled version of the code. LabVIEW also automatically recompiles all VIs that call the changed VI, resulting in unsaved changes to the calling VIs. If the calling VIs are stored in *source control*, these pending unsaved changes force you to check out the VIs when you update their subVIs.

To avoid this problem, LabVIEW allows you to separate compiled code from the graphical source code of a VI, thereby creating source-only VIs. After you separate the compiled code from a VI, LabVIEW still compiles the VI when you modify subVIs. However, LabVIEW stores the recompiled version of the code in a *compiled object cache* rather than in the VI in source control. Because the VI in source control does not change, you no longer have to check the VI out of source control when you change subVIs.

Determining Whether to Separate Compiled Code from VIs

Separate compiled code from VIs for the following reasons:

- **To simplify source control** —If you separate compiled code from all files within a *VI hierarchy* or LabVIEW project, changes to one VI do not produce unsaved changes elsewhere in the VI hierarchy. This behavior allows you to check out only the VI you want to modify rather than all the VIs that call the changed VI.
- **To prepare VIs in source control to be upgraded to a new version of LabVIEW** — You can upgrade source-only VIs to a new version of LabVIEW without checking out the VIs from source control.
- **To improve load time for VIs** —LabVIEW can load source-only VIs more quickly than regular VIs. To maximize this benefit, separate compiled code from all the files in a VI hierarchy or project.

not *LabVIEW Run-Time Engine compiled object cache build a source distribution*

Facilitating Source Control for Other File Types

LabVIEW also allows you to create source-only project libraries, LabVIEW classes, and XControls that prevent member files from causing unsaved changes in the versions stored in source control. Unlike for source-only VIs, LabVIEW does not actually separate compiled code from these file types, some of which do not even contain compiled code. Instead, when you select the **Separate compiled code from source file** option for these file types, LabVIEW stores an up-to-date version of each file in the object cache. If you change a member VI of the library, class, or XControl, LabVIEW updates the version of the file in the object cache rather than asking you to save the version that is in source control. Therefore, source-only libraries, classes, and XControls allow all the same benefits as source-only VIs.

Build Specifications

Building and Distributing Applications

Build specifications allow you to build and to distribute customized LabVIEW applications.



NOTE

To build a stand-alone application, installer, .NET Framework assembly, packed project library, shared library, or zip file, you must have the Application Builder installed. The LabVIEW Professional Development System includes the Application Builder. If you use another LabVIEW development system, you can purchase the Application Builder separately at ni.com.

Getting Started

Use the following topics to get started with or to troubleshoot LabVIEW-built applications.

- Developing and Distributing an Application

Debugging

- Debugging Applications and Shared Libraries
- Debugging VIs

Application Builder VIs

- Build VI
- Deploy VI

- Clean VI



RELATED INFORMATION

- [Building VI Server Applications](#) on page 260
- [Debugging Applications and Shared Libraries](#) on page 1437
- [Debugging Techniques](#) on page 329
- [Developing and Distributing an Application](#) on page 1497
- [Debugging Applications and Shared Libraries](#) on page 1437
- [Debugging Techniques](#) on page 329

Using Build Specifications

Use **Build Specifications** in the [Project Explorer](#) window to [create](#) and [configure](#) build specifications for LabVIEW builds. A build specification contains all the settings for the build, such as files to include, directories to create, and settings for VIs. The following table lists the LabVIEW requirements for each build specification.

Build Specification	Requires
Stand-alone application	Application Builder or Professional Development System
Installers	Application Builder or Professional Development System
.NET Framework assemblies	Application Builder or Professional Development System
Packages	Application Builder or Professional Development System
Packed libraries	Application Builder or Professional Development System
Shared libraries	Application Builder or Professional Development System
Source distributions	Base Development System or Full Development System
Web services	Base Development System or Full Development System
Zip files	Application Builder or Professional Development System



NOTE

You must have the Application Builder installed to build stand-alone applications, shared libraries, installers, and zip files. The LabVIEW Professional Development System includes the Application Builder. If you use the LabVIEW Base Development System or Full Development System, you can purchase the Application Builder separately by visiting the National Instruments website. If you have already purchased the Application Builder, select **Help»Activate LabVIEW Components** to activate the product.

The build specification interface is a set of dialog boxes that allow you to customize your build specification type. Each dialog box has pages unique to each build specification type. For example, when building a stand-alone application, the Application Properties dialog box allows you to select which VIs launch at startup and which VIs are always included.



Note If you previously hid **Build Specifications** in the Project Explorer window, you must *display the item* again to access it in the **Project Explorer** window.

You can create and configure the following types of *build specifications*:

- **Stand-alone applications** —Use stand-alone applications to provide other users with executable versions of VIs. Applications are useful when you want users to run VIs without installing the LabVIEW development system. Stand-alone applications require the *LabVIEW Run-Time Engine*. (Windows) Applications have a .exe extension. (macOS) Applications have a .app extension.
- **Installers** — (Windows) Use installers to distribute stand-alone applications, shared libraries, and source distributions that you create with the Application Builder. Installers that include the LabVIEW Run-Time Engine are useful if you want users to be able to run applications or use shared libraries without installing LabVIEW.
- **.NET Framework assemblies** — (Windows) Use .NET Framework assemblies to package VIs for the Microsoft .NET Framework. You must install the .NET Framework 4.0 to build a .NET Framework assembly using the Application Builder.
- **Packages** — (Windows) Use *packages* to distribute source distributions, packed project libraries, shared libraries,.NET assemblies, or executables to clients. You can distribute packages to clients through NI Package Manager or SystemLink. Your clients can use Package Manager or SystemLink to subscribe to a feed to find and install your packages. Packages have a .nipkg extension. (NI Linux Real-Time) You can also create opkg packages .ipk on NI Linux Real-Time targets if you install the LabVIEW Real-Time Module. Your clients can install packages through SystemLink or from the command line on the NI Linux Real-Time target. Package Manager does not support .ipk files.
- **Packed project libraries** —Use packed project libraries to package multiple LabVIEW files into a single file. When you deploy VIs in a packed library, fewer files deploy because the packed library is a single file. The top-level file of a packed library is a project library. Packed libraries contain one or more VI hierarchies that compile for a specific operating system. Packed libraries have a .lvlibp extension.
- **Shared libraries** —Use shared libraries if you want to call VIs using text-based programming languages, such as LabWindows™/CVI™, Microsoft Visual C++, and Microsoft Visual Basic. Using shared libraries provides a way for programming languages other than LabVIEW to access code developed with LabVIEW. Shared libraries are useful when you want to share the functionality of the VIs you build with other developers. Other developers can use the shared libraries but cannot edit or view the block diagrams unless you *enable debugging*. (Windows) Shared libraries have a .dll extension. (macOS) Shared libraries have a .framework extension. (Linux) Shared libraries have a .so extension. You can use .so or you can begin with lib and end with .so, optionally followed by the version number. This allows other applications to use the library.



Note (Linux) Visit ni.com/info and enter the Info Code `exjze4` for more information about the `libstdc++.so` library that installs with LabVIEW.

- **Source distributions** —Use source distributions to package a collection of source files. Source distributions are useful if you want to send code to other developers to use in LabVIEW. You can configure settings for specified VIs to add passwords, remove block diagrams, or apply other settings. You also can select different destination directories for VIs in a source distribution without breaking the links between VIs and subVIs.
- **Zip files** —Use zip files when you want to distribute files or an entire LabVIEW project as a single, portable file. A zip file contains compressed files, which you can send to users. Zip files are useful if you want to distribute selected source files to other LabVIEW users. You also can use the Zip VIs to create zip files programmatically.

You can distribute these files without the LabVIEW development system; however, to run stand-alone applications and shared libraries, users must have the [LabVIEW Run-Time Engine](#) installed.



[Building and Distributing Applications Home](#)

Creating Build Specifications

In the Project Explorer window, expand **My Computer**, right-click **Build Specifications**, select **New» your application type** from the shortcut menu. Use the following dialog boxes to configure settings for the build specification:

- Application Properties
- Packed Library Properties
- Shared Library Properties
- Source Distribution Properties
- Zip File Properties
- (Windows) .NET Framework Assembly Properties
- (Windows) Installer Properties
- (Windows) Web Service Properties
- (Windows) Package Properties

After you define these settings, click the **OK** button to close the dialog box and update the build specification in the project. The build specification appears under **Build Specifications**.

Right-click a specification and select **Build** from the shortcut menu to complete the build. You also can build the application programmatically using the Build VI or the [ExecuteBuildSpec](#) command line operation.

You also can select **Build All** from the shortcut menu to build all specifications under **Build Specifications**.

If you rebuild a given specification, LabVIEW overwrites the existing files from the previous build that are part of the current build. However, if you enable the **Auto increment** checkbox—or the **Auto increment product version** checkbox for installers—on the **Version Information** page, and you use the [VersionNumber] or [ProductVersion] tags in the appropriate build destination fields, LabVIEW creates a new directory that includes the version number instead of overwriting the previous build on disk.



NOTE

You must save a project to create build specifications. If you use an unsaved project, LabVIEW prompts you to save before opening the dialog box you use to configure each build specification.

Refer to the Generate Programmatic Build Command Line String VI in the `labview\examples\Application Control\Application Builder API` directory for an example of building a project build specification from the command line.

Related Information

[Developing and Distributing an Application](#)

Preparing Files for Build Specifications

Before you create a build specification for your application, prepare your files for distribution. For example, if you plan to distribute a stand-alone application to a system that does not have the LabVIEW Development System installed, be sure any properties and methods you use are supported in the LabVIEW Run-Time Engine.



RELATED INFORMATION

[VI Server Property and Method Characteristics - LabVIEW Programming Reference Manual](#)

Configuring Build Specifications

When you create a build specification, you can configure advanced features for each build specification type to customize the build. To modify the properties of an existing build specification, right-click the build specification in the Project Explorer window and select **Properties** from the shortcut menu. You can use the pages of the build specification properties dialog box to configure advanced settings for the application.

Related Topics

The following topics discuss the steps to help you configure your build specifications:

- [Configuring Run-Time Languages for Applications and Shared Libraries](#)
- [Copying LabVIEW Work Environment Options](#)
- [Executing a VI before or after a Build](#)



RELATED INFORMATION

- [Creating Build Specifications](#) on page 392
- [Configuring Run-Time Languages for Applications and Shared Libraries](#) on page 1505
- [Copying LabVIEW Work Environment Options](#) on page 1507
- [Running VIs](#) on page 226

Referencing Source Files in Stand-Alone Applications and Shared Libraries

The Application Builder stores source files inside an application or shared library based on the directory structure of the source files on disk. This internal file layout preserves source file hierarchy inside the application.

If you [call VIs dynamically](#), use [relative paths](#) to ensure the application loads the VIs correctly at run time. If the application references a data file outside of the application, you can use the Application Directory VI to determine the path to the file.

If you want to call VIs from the `labview\vi.lib` directory dynamically, create a wrapper VI for the file you want to load by adding the VI to the block diagram of a new VI. Prepare the wrapper VI as a [subVI](#), and ensure the [connector pane](#) for the wrapper VI matches the connector pane of the original `vi.lib` VI.

Using the LabVIEW 8. File Layout

If you enable the **Use LabVIEW 8.x file layout** option on the Advanced page of the Application Properties or Shared Library Properties dialog boxes, the Application Builder stores source files inside the application as a flat list. Because of this file structure, LabVIEW moves any files with conflicting filenames to different folders. National Instruments recommends you only enable this option if the application you build relies on the legacy behavior to store files in a flat list inside the application.

You can use the Built App File Layout VI located in the `labview\vi.lib\appbuilder` directory to determine if your application is using the legacy file layout for storing VIs inside a stand-alone application or shared library.

Path to source files	Path to files in application	Path to files in application v layout
C:\Source\foo.vi	C:\..\Application.exe\foo.vi	C:\..\Application.exe

Path to source files	Path to files in application	Path to files in application v layout
C:\Source\xxx\a.vi	C:\..\Application.exe\xxx\a.vi	C:\..\Application.exe
C:\Source\yyy\b.vi	C:\..\Application.exe\yyy\b.vi	C:\..\Application.exe

Managing Build Specifications

After you configure build specifications in the Project Explorer window, LabVIEW provides ways to manage and track your build specifications. Keeping track of your build specifications throughout development can help improve your development cycle.

LabVIEW also provides functionality when working with your build specifications inside the project. You can duplicate build specifications in the Project Explorer window. Right-click the build specification item you want to duplicate and select **Duplicate** from the shortcut menu to create a copy of the build specification item.

You also can drag and drop build specification items to rearrange the build order within the same build specification.

To delete the build specification files that LabVIEW creates during the most recent build, right-click the build specification and select **Clean** from the shortcut menu. Selecting **Clean** does not delete files that LabVIEW creates before the most recent build. You also can use the Clean VI to delete files from the most recent build.

If you build a stand-alone application, you can copy your LabVIEW work environment options into the build.

Applications in LabVIEW

Managing Applications

You can track builds in LabVIEW to help manage multiple versions of an application. Tracking builds is useful if you need to distribute multiple versions of a build. For example, you may want to distinguish between major and minor build versions when distributing to your end users.

You can track builds using the **Version Number** feature on the **Version Information** page of the properties dialog box. If you place a checkmark in the **Auto Increment** checkbox, LabVIEW automatically updates the build component of the version number. To edit the **Major**, **Minor**, or **Fix** components, you must remove the checkmark and manually increment the numbers. Through the **Version Information** page, you also can assign an internal name for the build or provide a description visible to your end users. For example, you may want to fill in the **Description** field to provide your users with a description of any major changes to your application.

Related Topics

The following topics discuss the steps to help you manage your build specifications:

- [Creating an About Dialog Box for a Stand-Alone Application](#)
- [Debugging Applications and Shared Libraries](#)
- [Enabling the Web Server in Stand-Alone Applications](#)
- [Retrieving Application and Shared Library Versions Programmatically](#)
- [Using a Configuration File with a Stand-Alone Application](#)

Features of Built Applications

You can use LabVIEW-built applications on other computers and from other programming languages. LabVIEW-built applications have specific features. For example, the data type characteristic of the .NET Framework assembly has specific features.

Related Topics

The following topics discuss characteristics of different types of LabVIEW-built applications:

- Characteristics of LabVIEW-Built .NET Framework Assemblies
- Characteristics of LabVIEW-Built Shared Libraries



RELATED INFORMATION

[Building VI Server Applications](#) on page 260

[Characteristics of LabVIEW-Built .NET Framework Assemblies](#) on page 1511

[Characteristics of LabVIEW-Built Shared Libraries](#) on page 1513

Installing Stand-Alone Applications and Shared Libraries

If you build a stand-alone application or a shared library with Windows, consider building an installer when distributing your builds. If you build an installer, you can include any components users will need to run your application. For example, include the LabVIEW Run-Time Engine if you will be running your application on a computer that does not have the LabVIEW Development System installed.

Related Topics

The following topics discuss considerations for installing stand-alone applications and shared libraries:

- Caching Installer Distribution Components
- Caveats and Recommendations for Building Installers
- Including Additional National Instruments Installers in an Installer Build
- Installing LabVIEW-Built Installers Silently
- Uninstalling LabVIEW-Built Installers
- Using the LabVIEW Run-Time Engine



RELATED INFORMATION

[Developing and Distributing an Application](#) on page 1497

[Caching Installer Distribution Components \(Windows\)](#) on page 1514

[Caveats and Recommendations for Building Installers](#) on page 1514

[Including Additional National Instruments Installers in an Installer Build](#) on page 1516

[Installing LabVIEW-Built Installers Silently](#) on page 1518

[Uninstalling LabVIEW-Built Installers](#) on page 1519

[Using the LabVIEW Run-Time Engine](#) on page 397

Using the LabVIEW Run-Time Engine

You must install the LabVIEW Run-Time Engine on any computer on which you want to run any type of LabVIEW-built application. The LabVIEW Run-Time Engine includes the libraries and other files necessary to run basic application types built in LabVIEW.



NOTE

The LabVIEW Run-Time Engine is multilingual.

When you create build specification for an installer, the Application Builder includes the LabVIEW Run-Time Engine by default.

All types of applications built with one particular version of LabVIEW can share the same LabVIEW Run-Time Engine, so you need to install the LabVIEW Run-Time Engine for that version only once. However, if you want to run multiple LabVIEW-built applications created with different versions of LabVIEW on the same computer, you must install each version of the LabVIEW Run-Time Engine that corresponds with each version of LabVIEW used to create the application types. For example, if you want to run an application created in LabVIEW 7.0 and an application created in LabVIEW 8.0 on the same computer, the computer must have versions 7.0 and 8.0 of the LabVIEW Run-Time Engine.

Considerations for the LabVIEW Run-Time Engine

- (Windows) If you want to include the LabVIEW Run-Time Engine in an installer built with the Application Builder, remember that you must log on as an Administrator or a user with administrator privileges to run that installer. (Linux) When you install the

LabVIEW Run-Time Engine on a target computer where you want to run an application that uses the Mathematics or Signal Processing VIs, enter **yes** when prompted to install the LabVIEW Run-Time Advanced Analysis Support package `labview-rte-aal-1.1-1.i386.rpm`.

- Some VI Server properties and methods are not supported in the LabVIEW Run-Time Engine. Avoid using these properties and methods in the VIs you include in an application or shared library.
- Incorporate error handling into the VIs of the application because LabVIEW does not display automatic error handling dialog boxes in the LabVIEW Run-Time Engine.
- If the VI uses custom run-time menus, make sure the application menu items that the VI uses are available in the LabVIEW Run-Time Engine.
- When you close all front panel windows in an application, the application stops. If the VI you build into the application contains code that executes after the last front panel window closes, this code does not execute in the application. Avoid writing block diagram code that executes after the last front panel window closes.
- If you reference a VI in an application using the Call By Reference node, if a VI uses Property Nodes to set front panel properties, or if a front panel window appears to users, remove the checkmark in the **Remove front panel** checkbox for that VI from the **Use default save settings** section on the Source File Settings page of the Application Properties dialog box. If you remove the front panel window, the Call By Reference node or Property Nodes that refer to the front panel window will return errors that might affect the behavior of the application.

Using ActiveX with LabVIEW

To access the objects, properties, methods, and events associated with other ActiveX-enabled applications, you can use LabVIEW as an ActiveX client. To allow other applications to access LabVIEW objects, properties, and methods, you also can use LabVIEW as an ActiveX Server. Refer to ActiveX and LabVIEW at ni.com for more information about using ActiveX with LabVIEW.

ActiveX Objects, Properties, and Methods

ActiveX-enabled applications include objects that have exposed properties and methods that other applications can access. Objects can be visible to the users, such as buttons, windows, pictures, documents, and dialog boxes, or invisible to the user, such as application objects. You access an application by accessing an object associated with that application and setting a property or invoking a method of that object.

ActiveX Events

Events are the actions taken on an object, such as clicking a mouse or pressing a key. Whenever these actions occur to the object, the object sends an event to alert the ActiveX container, along with the event-specific data. The ActiveX object defines the events available for an object.

To use ActiveX events in an application, you must register for the event and handle the event when it occurs. ActiveX event registration is similar to [dynamic event registration](#). In

dynamic event registration, you must place an event structure within a While Loop, wire an object reference to the event structure, and select the event you want LabVIEW to generate. For example, you can call a Windows-based tree control from an ActiveX container and specify that you want to generate a Double Click event for the items displayed in the tree control. However, the architecture of an ActiveX event VI is different than the architecture of an event-handling VI because you must use the Register Event Callback function instead of the Register for Events function to specify and register events.



NOTE

National Instruments recommends that you use the Unregister For Events function to unregister for events when you no longer need to handle them. If you do not unregister for events, LabVIEW continues to generate and queue the events as long as the VI runs, even if no Event structure is waiting to handle them, which consumes memory and can hang the VI if you enable front panel locking for the events.

Refer to the following table for information about the components of generating and registering an ActiveX event in LabVIEW.

Component	Description	Use Case
ActiveX object	This is the ActiveX object for which you want to generate an event.	After you select the ActiveX object for which you want to generate an event, you can use an automation refnum to specify the event.
Register Event Callback function	This function is a growable node capable of handling multiple events, similar to the Register For Events function.	After you wire an ActiveX object reference to the Register Event Callback function, you can specify the event(s) you want to generate for that object. When you specify the event(s) you want to generate for an object, you register the ActiveX object for that event
Callback VI	This VI contains the code that you write to handle an event.	You can create this VI after you register for an event with the Register Event Callback function. Different events may have different event data formats. Therefore, changing the event after you create a callback VI might break wires on the block diagram. Select the event before you create the callback VI.

ActiveX VIs, Functions, Controls, and Indicators

Refer to the following table for information about which VIs, functions, controls, and indicators to use to access the objects, properties, methods, and events associated with other ActiveX-enabled applications.

VI, Function, Control, or Indicator Name	Description	Example	How to Use
Automation refnum control	This control creates a reference to an ActiveX object.	You can create a reference to a word document embedded in an ActiveX container on the front panel of a VI.	Right-click this control on the front panel to select an object from the type library you want to access.
Automation Open function	This function opens an ActiveX object.	You can use this function to open a Microsoft Excel document.	Right-click the function and select Select ActiveX Class to choose the class of the object from the shortcut menu. After you open the refnum, you can pass it to other ActiveX functions.
ActiveX container	This container accesses and displays an ActiveX object on the front panel.	You can use the ActiveX container to display a button from another ActiveX-enabled application, such as a web browser.	Right-click the container and select Insert ActiveX Object from the shortcut menu. Then select the object you want to access.
Property Node	This function gets (reads) and sets (writes) the properties associated with an ActiveX object.	You can use this function to set the properties of an object you embed in an ActiveX container on the front panel.	Right-click an ActiveX object on the front panel or block diagram and select Create»Property Node . Then select a property from the shortcut menu to set a property for the object.
Invoke Node	This function invokes methods associated with an ActiveX object.	Similar to how you set properties with a Property Node, you can use this function to invoke methods of an object you embed in an ActiveX container on the front panel.	Right-click an ActiveX object on the front panel or block diagram and select Create»Invoke Node . Then select a method from the shortcut menu to invoke a method for the object.
Register Event Callback	This function handles events that occur on an ActiveX object.	You can use this function to generate and register an event, such as adding data to a Microsoft Excel spreadsheet.	
Variant control and indicator	Together, this control and indicator pass data to or from ActiveX controls.	You can use this control and indicator to pass data to and from an ActiveX-enabled application, such as Microsoft Excel or Microsoft Word.	Navigate to the Variant & Class palette to locate and drop this control and indicator on the front panel or block diagram.

Data Type Mapping

LabVIEW converts the data types of some ActiveX properties and methods into LabVIEW data types to read and interpret the data. The following table lists the ActiveX data types and the corresponding converted LabVIEW data types.

ActiveX Data Type	LabVIEW Data Type
char	 , 
short	 , 
long	 , 
hyper	 , 
float	 , 
double	
BSTR	 , 

LabVIEW as an ActiveX Client

When LabVIEW accesses the objects associated with another ActiveX-enabled application, LabVIEW acts as an ActiveX client. You can use LabVIEW as an ActiveX client in the following ways:

- Open an application, such as Microsoft Excel, create a document, and add data to that document.
- Embed a document, such as a Microsoft Word document or an Excel spreadsheet, on the front panel in a container.
- Place a button or other object from another application, such as a **Help** button that calls the other application help file, on the front panel.
- Link to an ActiveX control you built with another application.

LabVIEW accesses an ActiveX object with the automation refnum control or the ActiveX container, both of which are front panel objects. Use the automation refnum control to select an ActiveX object. Use the ActiveX container to select a displayable ActiveX object, such as a button or document, and place that object on the front panel. Both objects appear as automation refnum controls on the block diagram.

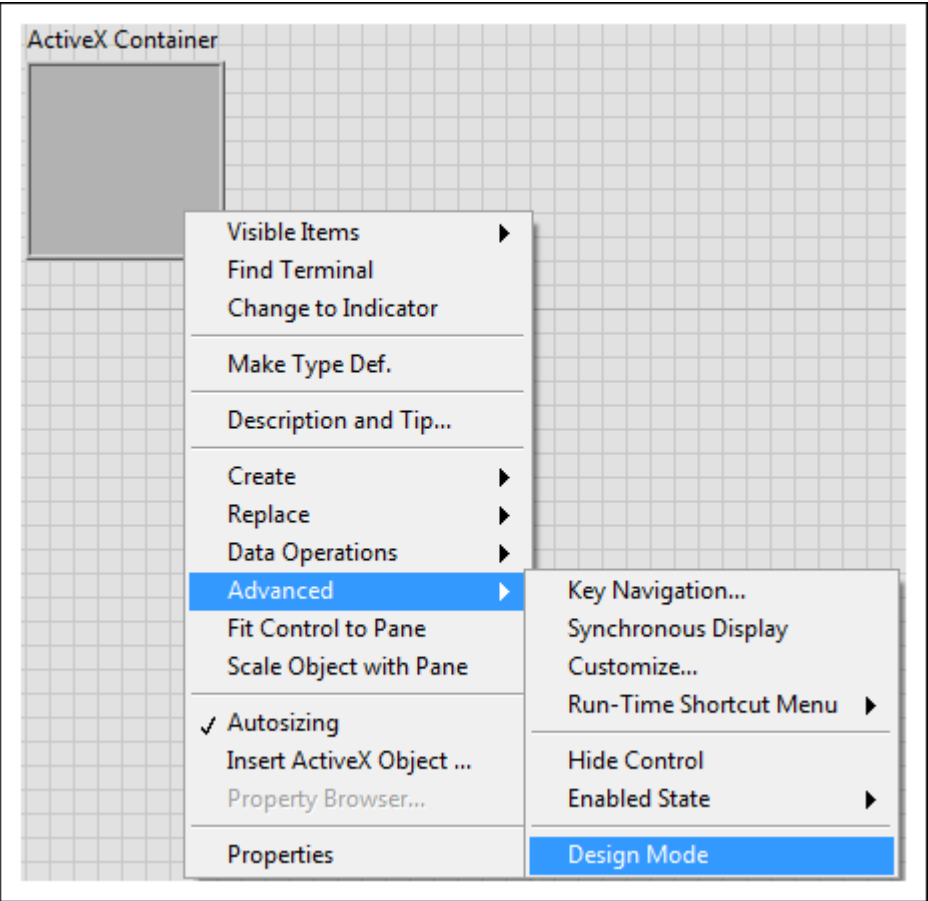
Design Mode for ActiveX Objects

When you select the design mode option, the ActiveX container owns the references to controls and creates and releases new controls. Controls have distinct design and run modes. Each time you run a VI, LabVIEW removes the design mode control and replaces it with a new control in run mode. When the VI stops, LabVIEW removes the run mode control

and creates a new design mode control. The relationship between design mode and run mode when you use ActiveX and LabVIEW is similar to the relationship between edit mode and run mode in other areas of LabVIEW programming, such as VI Scripting or VI Server.

Built applications do not include design mode control. LabVIEW creates the run mode control when the VI starts and releases the control when the VI stops. LabVIEW generates events only in run mode, not in design mode. The default mode is run mode.

Refer to the following illustration for information about how to configure design mode.



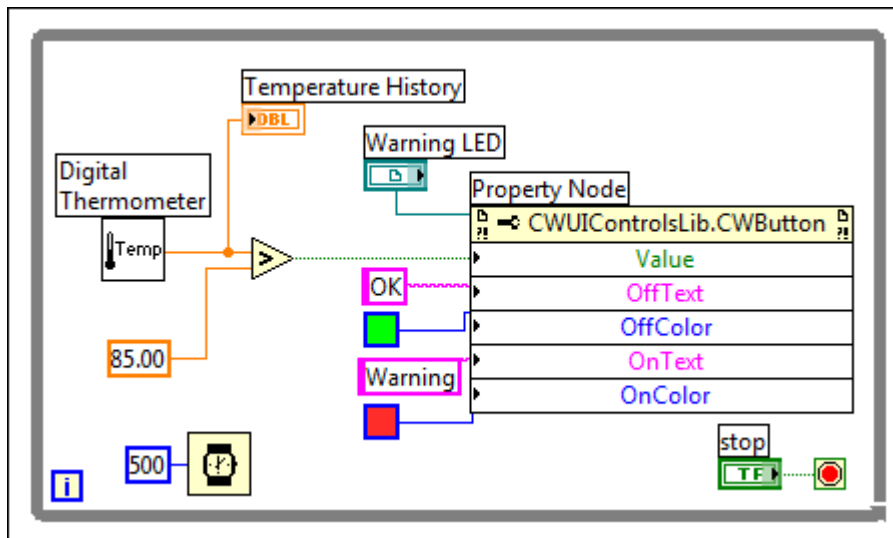
Methods for Setting ActiveX Properties

After you open an ActiveX server or insert an ActiveX control or document, you can set the properties associated with that control or document using the ActiveX Control Property Browser, property pages, and the Property Node.

Name	Description	How To Access
ActiveX Control Property Browser	Use this browser to view and set all the properties associated with an ActiveX control or document in an ActiveX container. The ActiveX Control Property Browser is an easy way to set the properties of an ActiveX object interactively.	Access the ActiveX Control Property Browser in either of the following ways: - Right-click the control or document in the container on the front panel and select Property Browser from the shortcut menu. Select View»ActiveX Control Property Browser.
	Note You can use the browser only to set properties while in edit mode and only with ActiveX objects in a container.	
ActiveX Property Pages	Many ActiveX objects include property pages, which organize the properties associated with the object on separate tabs. Like the ActiveX Control Property Browser, ActiveX property pages are an easy way to set the properties of an ActiveX object interactively. Also like the ActiveX Control Property Browser, you can use property pages only to set properties while in edit mode and with ActiveX objects in a container.	To access ActiveX property pages, right-click the object in the container on the front panel and select the name of the object from the shortcut menu.
	Note Not all ActiveX objects provide property pages.	
Property Nodes	Use the Property Node to set ActiveX properties programmatically.	Complete the following steps to use a Property Node to set ActiveX properties programmatically: - Obtain a reference to the ActiveX object for which you want to create an ActiveX property. - Add a Property Node to the block diagram - Wire the reference to the object to the reference input of the node.

The following block diagram example demonstrates how to use a Property Node to programmatically change the Value property of an ActiveX control to warn a user when a temperature exceeds a certain limit. The Value property of the CWButton ActiveX control, which is part of the National Instruments Measurement Studio User Interface ActiveX Library, is changed when the temperature reaches 85 degrees Fahrenheit or higher.

In this example, the CWButton control acts as an LED, changes colors, and displays **Warning** when the temperature reaches the limit, which is the “on” state of the CWButton control.



Note Instead of setting the OffText, OffColor, OnText, and OnColor properties for the CWButton control properties programmatically, you can use the ActiveX Control Property Browser or property pages to set properties for those controls.

LabVIEW as an ActiveX Server

You can access a LabVIEW application, VIs, and properties and methods for controls through ActiveX calls from other applications. Other ActiveX-enabled applications, such as Microsoft Excel, can request properties, methods, and individual VIs from LabVIEW. LabVIEW acts as an ActiveX server.

For example, you can embed a VI graph in an Excel spreadsheet and, from the spreadsheet, enter data in the VI inputs and run the VI. When you run the VI, the data plot to the graph.

Support for Custom ActiveX Automation Interfaces

If you write to a third party ActiveX client that uses LabVIEW to access properties and methods from a LabVIEW ActiveX server, you can access custom interfaces exposed by the server. You can do this without IDispatch.

However, the developer of the ActiveX server must make sure the parameters of the properties and methods in these custom interfaces have Automation (IDispatch) data types. The developer of the server must ensure the parameters have Automation (IDispatch) data type in order to expose multiple interfaces from one object, rather than through multiple objects. Without Automation data types, you must make a connection for each object to which you connect, rather than just one connection. You still can use the interfaces in the LabVIEW environment.

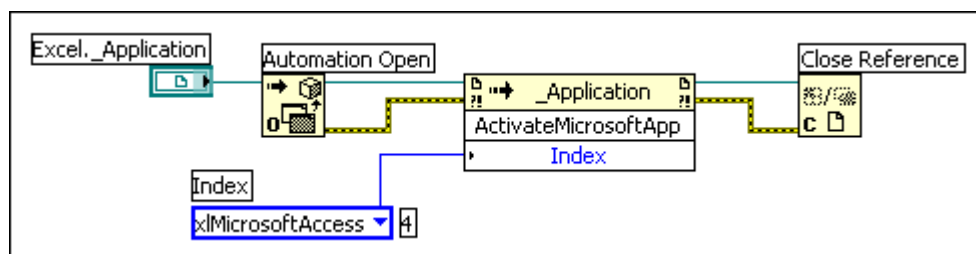
Using Constants to Set Parameters in ActiveX VIs

Some parameters in ActiveX nodes accept a discrete list of valid values. Select the descriptive name in the ring constant to set these parameter values.

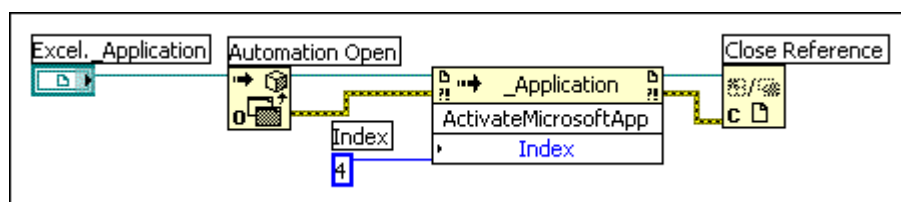
To access the ring constant when building an ActiveX VI, right-click the parameter of the node that accepts data values and create a constant. The selections available in the ring constant depend on the refnum passed to the node. To view the corresponding numeric data value, right-click the ring constant and select **Visible Items»Digital Display** from the shortcut menu.

The following block diagrams illustrate how to use ring and numeric constants to set parameter values. The block diagrams access the Microsoft Excel Application and execute a method, and the **Index** parameter includes several options. When you select an option, LabVIEW displays the corresponding numeric value of the option.

By default, as displayed in the following block diagram, LabVIEW displays the numeric value of the currently selected option, **MicrosoftAccess**, in a box next to the ring constant.



However, you also can choose to display the numeric value of an option in a numeric constant. As displayed in the following block diagram, LabVIEW displays the numeric value of an option in a numeric constant wired to the **Index** input.



Loading VIs with an Updated ActiveX Type Library

When you load a VI with a change in the major version of an ActiveX Type Library, LabVIEW launches a warning dialog box to inform you of the change. When you load a VI with a change in the minor version or time stamp of an ActiveX Type Library, the VI, once loaded, includes an asterisk in its title bar and in the list of open VIs displayed in the **Window** menu. When you save the VI, the asterisk disappears until you make a new change.

Deploying LabVIEW Applications on Windows Vista and Later Operating Systems

The security changes introduced by Windows Vista result in the following two issues to consider when using the LabVIEW Application Builder on Windows Vista or later:

- LabVIEW embeds an **application manifest** in every stand-alone application and installer.
- Stand-alone applications and installers for stand-alone applications are not **signed**.

In addition to this help topic, you can refer to *User Account Control, Digital Signing, and Manifests in LabVIEW Built Applications* for the most up-to-date information about the LabVIEW Application Builder on Windows Vista and later.

Application Manifests

LabVIEW embeds an application manifest in every stand-alone application and installer. Because the manifest is embedded in the application, you must **retrieve the manifest** before it appears in the directory with the application. The manifest is an XML file with the extension `.manifest`. The manifest contains information that describes the application properties to Windows. While manifests are not new to Windows, the security section of the manifest is a new feature. You can now specify the requested execution level in the manifest for an application you deploy on Windows Vista or later.

Specifying the execution level of the application determines the required security privileges you need to launch an application. The execution level of the application determines if the application displays a dialog box prompting the user to give administrator consent or to enter an administrator password before launching the application. Applications that require administrator privileges have a security shield overlay on the application icon. Whether the application runs as an administrator or standard user then determines several properties of the application, including the directories to which the application can write. If you launch an application as a standard user, the application cannot write to protected directories and registries. The Program Files directory is an example of a protected directory on Windows Vista and later.

Microsoft recommends that applications run without requiring administrator privileges. Therefore, design applications so they do not attempt to access protected areas of the operating system. Running without administrator privileges gives all users the opportunity to run the application as intended.

The following code is an example manifest for a stand-alone application.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">

  <assemblyIdentity version="1.0.0.0"
```

```

processorArchitecture="X86"

name="National Instruments.LabVIEW.LabVIEW "

type="win32"␣

<description>LabVIEW Development System</description>

<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">

<security>

<requestedPrivileges>

<requestedExecutionLevel

level="asInvoker"

uiAccess="false"␣

</requestedPrivileges>

</security>

</trustInfo>

</assembly>

```

The requested execution level of a stand-alone application can be any of the following values:

- **asInvoker** —(Default) The stand-alone application executes with the same access privileges as the user, either standard or administrator. The application does not display a dialog box prompting the user for consent before it runs. If the application runs as standard user, the application cannot write to protected directories. If the application runs as administrator, the application can write to protected directories.
- **highestAvailable** —The stand-alone application executes with the highest access privileges possible. If the user has only standard user privileges, the application does not display a dialog box prompting the user for consent, but the application cannot write to protected directories. If the user has administrative privileges, the application displays a dialog box prompting the user to either consent or to enter the administrator password and can write to protected directories.
- **requireAdministrator** —The stand-alone application executes at the administrator level. If the user has only standard privileges, the application displays a dialog box prompting the user to enter the administrator password and can write to protected directories. If the user has administrator privileges, the application displays a dialog box prompting the user for consent before it runs. The application can write to protected directories.

If you develop a stand-alone application in LabVIEW 8.2.1 or later and deploy the application on Windows XP or earlier, a stand-alone application that includes a manifest and has the requested execution level specified does not exhibit the Windows Vista and later behavior. Windows XP ignores the security section of the manifest.

The `install.exe` in LabVIEW-built installers also contains a manifest. The default requested execution level is `requireAdministrator`. Because the `install.exe` updates system-level files, settings, and configurations, avoid changing the requested execution level of `install.exe`. Changing the requested execution level of `install.exe` might cause the installer to function incorrectly.

Digital Signing

Digital signing lets users verify who created an application or installer and decide whether the application or installer is safe to install or run. Because Windows Vista and later offer additional levels of security, users might encounter additional dialog boxes warning them about the risks of installing or running unsigned applications or installers. By default, LabVIEW does not sign stand-alone applications or installers so you can customize the digital signature. To reduce the amount of dialog boxes a user encounters, sign stand-alone applications and installers using the Windows Security page of the Application Properties dialog box or the Windows Security page of the Installer Properties dialog box, respectively, when you build.

You need the following tools to sign an application or installer:

- **Signing certificate** —You can purchase signing certificates from a number of vendors, including Verisign. You can purchase signing certificates only for companies or organizations. Individuals cannot purchase signing certificates. Signing certificates must be personal information exchange (PFX) files.
- **Signing tool** —The Windows Driver Kit includes the signing tool. You use the signing tool to apply the digital signature.

To sign a stand-alone application, you need to sign only the `.exe` file. To sign a LabVIEW-built installer, you need to sign only the `install.exe`. You do not need to sign any other files for stand-alone applications and installers. Digital signing involves procuring third-party providers for digital certificates, and no single procedure for signing code exists.



RELATED INFORMATION

[*User Account Control, Digital Signing, and Manifests in LabVIEW Built Applications*](#)

Deploying Builds on Windows Vista and Later Operating Systems

Windows Vista and later operating systems from Microsoft include additional levels of security, including User Account Control. By prompting the user to enter an administrative password prior to installing applications or completing other tasks, User Account Control helps prevent users from inadvertently introducing viruses or other types of changes that might destabilize the system. Additionally, a user who logs into Windows Vista or later as a

standard user can write only to specific locations on disk. Standard user is the default login on these operating systems.

Related Topics

The following topics discuss considerations for deploying build specifications on Windows Vista and later:

- [Deploying LabVIEW Applications on Windows Vista and Later Operating Systems](#)
- [Working with the Application Manifest](#)

Deploying Installers to Windows Embedded Standard Targets

After you develop a VI on the host computer, you can build the VI into an executable application. You also can build an installer that includes the executable application or other files and deploy or run the installer on a Windows Embedded Standard target. If you build an installer, include any components users need to run your application.



NOTE

To build a stand-alone application or installer, you must have the Application Builder installed. The LabVIEW Professional Development System includes the Application Builder. If you use another LabVIEW development system, you can purchase the Application Builder separately by visiting the National Instruments website.

To build an application or installer, you first must create a build specification. A build specification contains all the settings for the build, such as files to include, directories to create, and settings for VIs. You can create a build specification at any time by right-clicking **Build Specifications** under the Windows Embedded Standard target in the [Project Explorer](#) window and selecting **New»Application (EXE)** or **New»Installer** from the shortcut menu.

After creating the build specification, right-click the build specification in the [Project Explorer](#) window and select one of the following options from the shortcut menu:

- **Build** —Builds the build specification, such as an application or installer. This option does not deploy the installer to the target automatically.
- **Deploy** —Builds the installer and deploys the installer to the target. This option does not run the installer on the target automatically.
- **Install** —Builds, deploys, and runs the installer on the target to the location specified in the build specification.

Related Information

[Developing and Distributing an Application](#)

Creating Build Specifications

Using Build Specifications

Windows Connectivity

LabVIEW provides access to other Windows-based applications using .NET and ActiveX technologies.

The .NET technology is the programming basis of the .NET environment you use to build, deploy, and run Web-based applications, smart client applications, and XML Web services. You must install .NET. Refer to the Microsoft Developer Network (MSDN) website for more information about installing .NET.



NOTE

Creating and communicating with .NET Framework objects in LabVIEW requires the .NET CLR 4.0 that installs with LabVIEW. You must use a .NET 2.0 configuration file if you want to load .NET 2.0 mixed-mode assemblies. Refer to the *Requirements for using .NET Framework with LabVIEW* for more information about .NET restrictions in LabVIEW. NI strongly recommends that you use .NET Framework objects only in LabVIEW projects.

ActiveX refers to Microsoft's ActiveX technology and OLE technology. With ActiveX Automation, a Windows-based application, such as LabVIEW, provides a public set of objects, commands, and functions that other Windows-based applications can access. Refer to the MSDN documentation for more information about ActiveX.

- Inside OLE, by Kraig Brockschmidt, second edition
- Essential COM, by Don Box

Refer to ActiveX and LabVIEW at ni.com for more information about using ActiveX with LabVIEW.



RELATED INFORMATION

[Requirements for Using .NET Framework Assemblies in LabVIEW](#) on page 791

[Loading .NET Framework 2.0, 3.0, and 3.5 Assemblies in LabVIEW](#) on page 1484

[Creating .NET Objects](#) on page 792

[Using Libraries in LabVIEW Projects](#) on page 370

[Callback VIs](#) on page 249

[Using .NET with LabVIEW](#) on page 789

[Using ActiveX with LabVIEW](#) on page 398

[Importing Web Services](#) on page 726

[Calling Non-Modal Windows Programmatically](#) on page 1036

[Importing a Web Service as a Library of VIs](#) on page 1036

[Registering and Handling .NET and ActiveX Events](#) on page 1479

Loops and Structures in LabVIEW

Loops and Other Structures

Structures are graphical representations of the loops and case statements of text-based programming languages.

The following topics provide information about using structures, located on the Structures palette.

Structure Use Case	Structure
Repeating blocks of code	
A set number or until an error	For Loop
Until a condition or until an error	While Loop
With time bounds and delays until a stop condition	Timed Loop
With time bounds and delays in order until a stop condition	Timed Loop with Frames
Executing all code in order	
With all subdiagrams visible	Flat Sequence structure
With one subdiagram visible	Stacked Sequence structure

Structure Use Case	Structure
With time bounds and delays	Timed Sequence structure
Executing code conditionally	
Based on values	Case structure
Based on events	Event structure
Disabling code	
Based on conditions	Conditional Disable structure
Of specific blocks	Diagram Disable structure
Based on compilation results	Type Specialization structure
Increasing memory efficiency	In Place Element structure



RELATED INFORMATION

[For Loops: Repeating Operations a Set Number of Times](#) on page 413

[While Loops: Repeating Operations Until a Condition Occurs](#) on page 414

[Configuring Timed Loops and Timed Loops with Frames \(Real-Time, Windows\)](#) on page 431

[Sequence Structures: Executing Sections of Code Sequentially](#) on page 421

[Configuring Timed Sequence Structures \(Real-Time, Windows\)](#) on page 435

[Case Structures: Executing a Section of Code Based on Input Values](#) on page 420

[Disable Structures: Preventing Code from Executing](#) on page 422

[In Place Element Structures: Increasing Memory Efficiency](#) on page 425

[Loops and Other Structures](#) on page 411

[Executing One or More Sections of Code Using Structures](#) on page 419

[In Place Element Structures: Increasing Memory Efficiency](#) on page 425

[Configuring Settings of a Timed Structure \(Real-Time, Windows\)](#) on page 436

[Case Structures: Executing a Section of Code Based on Input Values](#) on page 420

[Adding Structures to the Block Diagram](#) on page 1264

[Resizing Structures](#) on page 1264

[Adding Objects inside Structures](#) on page 1265

[Replacing One Structure with Another Structure](#) on page 1265

[Removing Structures without Deleting Objects in the Structure](#) on page 1266

Controlling Repetitive Operations Using Loops

Use the For Loop and the While Loop to repeat blocks of code or operations on the block diagram.



NOTE

If you open a refnum inside a For Loop or While Loop, close that refnum for each iteration of the loop. Otherwise, LabVIEW repeatedly allocates memory for the refnum and does not free the memory until the VI finishes running.

Related Topics

- Specifying the number of times a For Loop executes
- Specifying conditions for While Loops
- Processing individual elements in an array or a collection with a loop
- Handling errors using conditional terminals on loops
- Transferring values between loop iterations
- Configuring a Timed Loop



RELATED INFORMATION

[For Loops: Repeating Operations a Set Number of Times](#) on page 413

[While Loops: Repeating Operations Until a Condition Occurs](#) on page 414

[Refnum Controls and Indicators](#) on page 160

[Specifying the Number of Times Operations in For Loops Execute](#) on page 1274

[Specifying Conditions for While Loops](#) on page 1279

[Processing Individual Elements in an Array or a Collection with a Loop](#) on page 1266

[Handling Errors Using Conditional Terminals on Loops](#) on page 1273

[Transferring Values between Loop Iterations](#) on page 415

[Configuring Settings of a Timed Structure \(Real-Time, Windows\)](#) on page 436

For Loops: Repeating Operations a Set Number of Times

A For Loop executes the section of code inside the loop, called a subdiagram, a set number of times. The count **N** terminal sets the number of times to repeat the subdiagram. One subdiagram repetition is called an iteration.

Use a For Loop any time you want an operation to occur a set number of times. For example, if you want to write a set number of measurements to a file, place the code that writes measurements to the file inside a For Loop. The numeric value you wire to the count terminal determines the number of times the writing operation occurs. After the For Loop executes the subdiagram for the number of iterations you set, the For Loop stops.

To stop a For Loop when a condition occurs, add a conditional terminal. If you add a conditional terminal, the For Loop stops when the condition occurs or when the loop finishes executing the number of iterations you set.

Passing Data through For Loops

The For Loop does not pass values from one iteration to the next by default. To access data from the previous iteration and use it in the next iteration, add shift registers to the For Loop. For example, if you want to use the result of a mathematic operation in one iteration as the initial value for a mathematic operation in the next iteration, use shift registers.

When you wire an array or a collection data type to the input tunnel of a For Loop, the For Loop enables auto-indexing by default. When auto-indexing is enabled, the For Loop reads and processes one element in the array or collection per loop iteration, rather than reading and processing the entire array or collection each iteration. You can enable or disable auto-indexing depending on how you want the For Loop to process the data.

Improving For Loop Execution Speed

If rapid For Loop execution is crucial to the operation of a VI and the machine running the VI has multiple processors, enable parallel For Loop iterations, unless enabling parallel iterations produces an error or warning. When you enable parallel iterations, processors execute iterations simultaneously to improve performance.

While Loops: Repeating Operations Until a Condition Occurs

Similar to a Do Loop or a Repeat-Until Loop in text-based programming languages, a While Loop, shown as follows, executes a subdiagram until a condition occurs.



The While Loop executes the subdiagram until the conditional terminal, an input terminal, receives a specific Boolean value. The conditional terminal in a While Loop behaves the same as in a [For Loop with a conditional terminal](#). However, the While Loop does not include a set iteration count and runs infinitely if the condition never occurs.

The iteration terminal, shown as follows, is an output terminal that contains the number of completed iterations.



The iteration count for the While Loop always starts at zero.

Passing Data through While Loops

The While Loop does not pass values from one iteration to the next by default. To access data from the previous iteration and use it in the next iteration, add shift registers to the While Loop. For example, if you want to use the result of a mathematic operation in one iteration as the initial value for a mathematic operation in the next iteration, use shift registers.

When you wire an array or a collection data type to the input tunnel of a While Loop, the While Loop disables auto-indexing by default. You can enable auto-indexing by right-clicking the tunnel at the loop border and selecting **Enable Indexing** from the shortcut menu. When auto-indexing is enabled, the While Loop handles one element in the array or collection per loop iteration. If the number of loop iterations is greater than the array or collection size, the While Loop handles all the elements in the array or collection and uses the default value of the array or collection element type for later iterations. You can enable or disable auto-indexing depending on how you want the While Loop to process the data.

Transferring Values between Loop Iterations

Use the following objects to transfer values on For Loops, While Loops, or (Real-Time, Windows) Timed Loops :

- Shift registers — Passes one value or multiple values to the next loop iteration.
- Error registers — Passes error clusters on parallel For Loops.
- Feedback Node — Passes the value from the previous execution or sets the value each time the node executes.



RELATED INFORMATION

[For Loops: Repeating Operations a Set Number of Times](#) on page 413

[While Loops: Repeating Operations Until a Condition Occurs](#) on page 414

[Characteristics of Timed Structures \(Real-Time, Windows\)](#) on page 427

[Shift Registers: Passing Values between Loop Iterations](#) on page 416

[Passing One Value to the Next Loop Iteration](#) on page 1282

[Passing Multiple Values to the Next Loop Iteration](#) on page 1282

[Error Registers: Handling Errors on Parallel Iterations](#) on page 416

[Parallel Iterations: Improving For Loop Execution Speed](#) on page 1275

[Feedback Nodes: Storing Data between Loop Iterations](#) on page 417

[Passing the Last Value from the Previous Execution to the Feedback Node](#) on page 1286

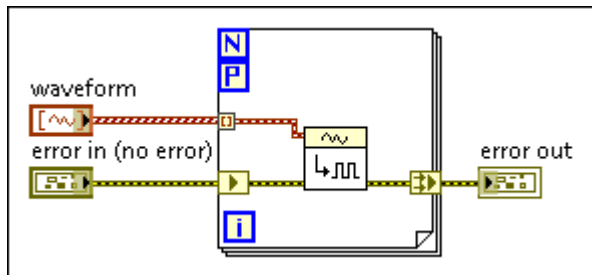
[Resetting the Input Value on a Feedback Node Each Time a Loop Executes](#) on page 1286

[Data Communication Methods in LabVIEW](#) on page 651

Error Registers: Handling Errors on Parallel Iterations

In a **For Loop** with **parallel iterations enabled**, LabVIEW uses error registers to pass **error clusters** through the loop. Error registers support the error handling that occurs at each iteration of the loop independent of when each iteration executes.

An error register appears as a pair of terminals on the sides of a parallel For Loop, as shown in the following block diagram.



The left side terminal of the error register behaves like a non-indexing input tunnel and produces the same value on every iteration. The right side terminal of the error register merges the values of each iteration such that the error or warning value from the earliest iteration, by index, is the output value of the error register. If the For Loop iterates zero times, the value you wire into the left side tunnel carries forward to the output on the right side tunnel.

LabVIEW preserves the best practice of flowing errors through a **shift register** by automatically converting shift registers to error registers when you configure parallel iterations on a For Loop. You can also change the tunnel type by right-clicking the tunnel and selecting the type of tunnel to create.

Shift Registers: Passing Values between Loop Iterations

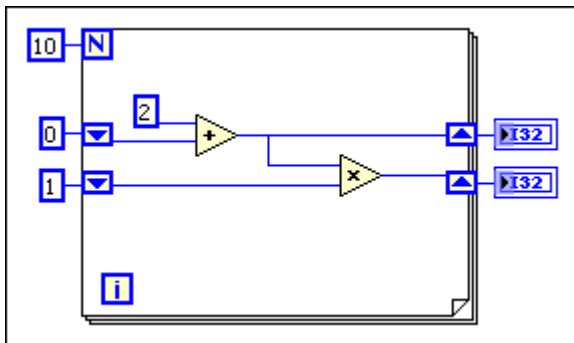
Use shift registers when you want to **pass values from previous iterations through a loop** to the next iteration. A shift register appears as a pair of terminals, shown as follows, directly opposite each other on the vertical sides of the loop border.



The terminal on the right side of the loop contains an up arrow and stores data on the completion of an iteration. LabVIEW transfers the data stored in the right terminal of the shift register to the left terminal. The loop then uses the data from the left terminal as the initial values for the next iteration. This process continues until all iterations of the loop execute. After the loop executes, the terminal on the right side of the loop returns the last value stored in the shift register.

A shift register transfers any data type and automatically changes to the data type of the first object wired to the shift register. The data you wire to the terminals of each shift register must be the same type. When an error cluster is wired to a shift register on a **parallel For Loop**, the shift register converts to an **error register**.

You can add more than one shift register to a loop. If you have multiple operations that use previous iteration values within a loop, use multiple shift registers to store the data values from those different processes in the structure, as shown in the following block diagram.



In the previous block diagram, the upper right shift register terminal transfers 2, the sum of 0 and 2 in the first iteration, to the upper left terminal as the initial value for the second iteration of the addition operation. The lower right terminal transfers 2, the product of 1 and 2 in the first iteration, to the lower left terminal as the initial value for the second iteration of the multiplication operation.

The second iteration adds 2 and 2 and transfers the outcome, 4, to the upper left terminal for the third iteration. The second iteration also multiplies 2 and 4 and transfers the outcome, 8, to the lower left terminal for the third iteration.

After ten iterations, the upper right terminal transfers the final outcome of the addition operation to the upper indicator and the lower right terminal transfers the final outcome of the multiplication operation to the lower indicator.



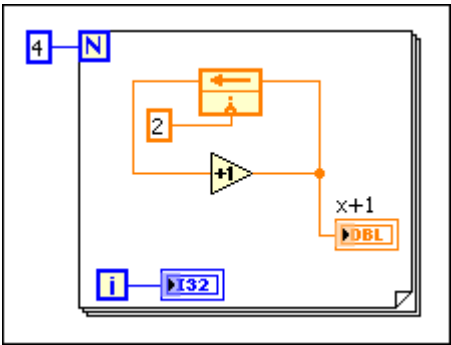
Feedback Nodes: Storing Data between Loop Iterations

The Feedback Node, shown as follows, operates similarly to [shift registers](#) in loops.



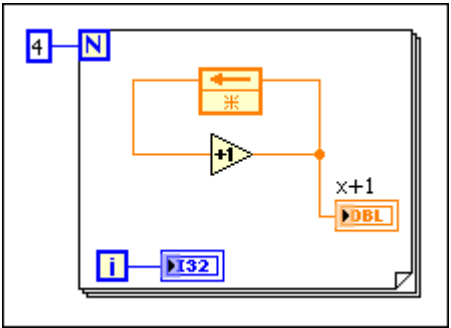
Like a shift register, the Feedback Node stores data when the loop completes an iteration and then sends the stored data to the next loop iteration. The next iteration reads the data, writes a new value to the data, and sends it through the loop again. This process continues until the loop execution completes.

When [the Feedback Node reinitializes after the first execution](#), the values for each loop iteration from one execution to another do not change. In this case, wire an initializer terminal to the Feedback Node, as shown in the following block diagram.



First Execution		Second Execution	
Iteration	End Value	Iteration	End Value
0	3	0	3
1	4	1	4
2	5	2	5
3	6	3	6

The Feedback Node behaves differently if you *do not wire an initial value to the initializer terminal*, shown as follows.



First Execution		Second Execution	
Iteration	End Value	Iteration	End Value
0	1	0	5
1	2	1	6
2	3	2	7
3	4	3	8

In the previous block diagram, the initializer terminal uses 0, the default value for the data type, as the initial value in the first execution and does not reinitialize the Feedback Node in any subsequent executions.

Although you can separate the node and initializer terminal in loops and nested loops, you cannot move the initializer terminal outside the set of nested structures that contains the node. You also cannot separate the node from the initializer terminal when you create a subVI. If you attempt to create a subVI from a section of the block diagram that contains either the node or the initializer terminal, but not both, LabVIEW returns an error.



NOTE

You can use the Appearance page of the Properties dialog box to change the appearance of a Feedback Node to z-transform view, which looks like that of a z^{-1} block in feedback control theory and signal processing. Changing the node to z-transform view changes only the cosmetic appearance of the Feedback Node and does not change the functionality of the node.



[Data Communication Methods Home](#)

Executing One or More Sections of Code Using Structures

The following structures contain multiple subdiagrams. A subdiagram is a section of code inside the structure.

- Case structure
- Disable structure
- Event structure
- In Place Element structure
- Sequence structure

Related Topics

- Adding, duplicating, moving between, deleting, and rearranging subdiagrams
- Specifying values of case subdiagrams on Case structures
- Passing data through frames of Flat Sequence structures
- Passing data through frames of Stacked Sequence structures

- Selecting symbols and values for Conditional Disable structures



RELATED INFORMATION

[Case Structures: Executing a Section of Code Based on Input Values](#) on page 420

[Disable Structures: Preventing Code from Executing](#) on page 422

[In Place Element Structures: Increasing Memory Efficiency](#) on page 425

[Sequence Structures: Executing Sections of Code Sequentially](#) on page 421

[Adding and Duplicating Subdiagrams](#) on page 1289

[Moving between Subdiagrams](#) on page 1290

[Deleting Subdiagrams](#) on page 1291

[Rearranging Subdiagrams](#) on page 1291

[Creating a Case Structure to Execute Conditional Code](#) on page 1292

[Passing Data through Frames of a Flat Sequence Structure](#) on page 1299

[Passing Data through Frames of a Stacked Sequence Structure](#) on page 1299

[Configuring Conditions for Conditional Disable Structures](#) on page 1301

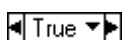
Case Structures: Executing a Section of Code Based on Input Values

A Case structure, shown as follows, has two or more [subdiagrams](#). Refer to the Case Structure - Selector Data Types VI in the `labview\examples\Structures` directory for an example of using Case structures.



Only one case subdiagram is visible at a time, and the structure executes only one subdiagram at a time. An input value determines which subdiagram executes. The Case structure is similar to switch statements or if...then...else statements in text-based programming languages.

The case selector label at the top of the Case structure, shown as follows, contains the name of the selector value that corresponds to the case in the center and decrement and increment arrows on each side.



Sequence Structures: Executing Sections of Code Sequentially

A sequence structure contains one or more **subdiagrams**, or frames, that execute in sequential order. Within each frame of a sequence structure, as in the rest of the block diagram, data dependency determines the execution order of nodes.

There are two types of sequence structures—the **Flat Sequence** structure and the **Stacked Sequence** structure. Use sequence structures sparingly because they hide code. Rely on data flow rather than sequence structures to control the order of execution. With sequence structures, you break the left-to-right data flow paradigm whenever you use a sequence local variable.



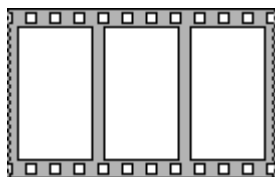
TIP

To help control data flow, you can use error clusters instead. If **flow-through parameters** are not available and you must use a sequence structure in a VI, consider using a Flat Sequence structure.

The output tunnels of a sequence structure can have only one data source, unlike **Case** structures. The output can emit from any frame. As with Case structures, data at input tunnels is available to all frames in either the Flat Sequence or the Stacked Sequence structure.

Flat Sequence Structure

The Flat Sequence structure, shown as follows, executes frames from left to right and when all data values wired to a frame are available. The data leaves each frame as the frame finishes executing. This means the input of one frame can depend on the output of another frame.

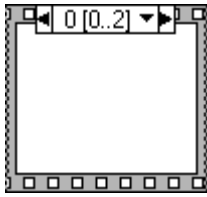


When you **add or delete frames** in a Flat Sequence structure, the structure resizes automatically.

If you **change a Flat Sequence** to a Stacked Sequence and then back to a Flat Sequence, LabVIEW moves all input terminals to the first frame of the sequence. The final Flat Sequence operates the same as the Stacked Sequence. After you change the Stacked Sequence to a Flat Sequence with all input terminals on the first frame, you can move wires to where they were located in the original Flat Sequence.

Stacked Sequence Structure

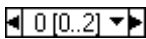
The Stacked Sequence structure, shown as follows, stacks each frame so you see only one frame at a time and executes frame 0, then frame 1, and so on until the last frame executes.



The Stacked Sequence structure returns data only after the last frame executes. Use the Stacked Sequence structure if you want to conserve space on the block diagram.

Unlike when you pass data between frames in the *Flat Sequence* structure, you need to *use sequence locals to pass data* from frame to frame in the Stacked Sequence structure.

The sequence selector identifier, shown as follows, at the top of the Stacked Sequence structure contains the current frame number and range of frames.



Use the sequence selector identifier to *navigate through the available frames* and *rearrange frames*. The frame label in a Stacked Sequence structure is similar to the case selector label of the Case structure. The frame label contains the frame number in the center and decrement and increment arrows on each side.

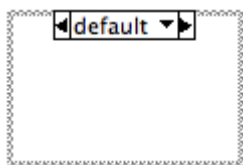
You cannot enter values in the frame label. When you add, delete, or rearrange frames in a Stacked Sequence structure, LabVIEW automatically adjusts the numbers in the frame labels.

Disable Structures: Preventing Code from Executing

A disable structure contains multiple *subdiagrams* of which exactly one compiles and executes. The subdiagram to execute is determined at compile time. Code within the inactive subdiagrams does not execute at run time or compile. You can use disable structures to make sections of code on the block diagram inactive.

Conditional Disable Structure

The Conditional Disable structure, shown as follows, has one or more subdiagrams, or cases, exactly one of which LabVIEW uses for the duration of execution, depending on the configuration of the conditions of the subdiagram. When compiling, LabVIEW does not include any code in the inactive subdiagrams of the Conditional Disable structure.



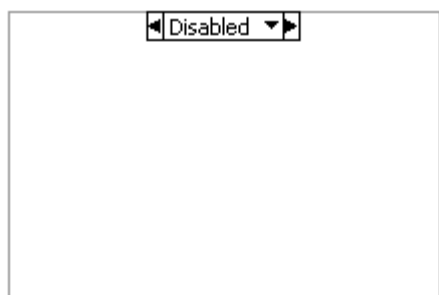
Use the Conditional Disable structure to define conditions in which specific code compiles and executes. For example, if one section of a VI is target-specific, you can put this code in a Conditional Disable structure and configure it to run on the specific target. You might configure a Conditional Disable structure for Windows, Mac, Unix systems or FPGA targets.

If you [convert from a Conditional Disable structure](#) to a Diagram Disable or Type Specialization structure, the conditions of the Conditional Disable structure do not transfer to the Diagram Disable or Type Specialization structure.

Refer to the Conditional Disable Structure VI in the `labview\examples\Structures\Disable Structures` directory for an example of using Conditional Disable structures.

Diagram Disable Structure

In the Diagram Disable structure, shown as follows, LabVIEW does not include any code in the **Disabled** subdiagrams when compiling.



Use the Diagram Disable structure if you want to disable specific code on the block diagram so that it does not compile at run time. For example, you can use the Diagram Disable structure as a [debugging tool](#) to comment out code, replace code, and compile a VI without deleting the code in the Disabled subdiagram of the structure.



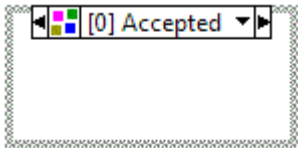
NOTE

If you want to enable a section of code only when certain conditions are met, use the Conditional Disable structure.

Refer to the Diagram Disable Structure VI in the `labview\examples\Structures\Disable Structures` directory for an example of using Diagram Disable structures.

Type Specialization Structure

The Type Specialization structure, shown as follows, has one or more subdiagrams, exactly one of which LabVIEW compiles and executes, depending on the order and the compilation result of the subdiagram. When compiling and executing, LabVIEW does not include any code in the inactive subdiagrams of the Type Specialization structure.



Use the Type Specialization structure to customize sections of code in a *malleable VI*. *vim* for specific data types or to force a malleable VI to accept only a subset of the acceptable data types. For example, you can use the Type Specialization structure to configure a malleable VI to behave differently when the input data is an integer versus an array of integers so that the malleable VI works in an expected way for both data types.

Refer to the `labview\examples\Malleable VIs\Type Specialization Structure\Malleable VIs - Type Specialization Structure.lvproj` for an example of using Type Specialization structures.

User-Defined Objects in Disable Structures

When LabVIEW loads a VI with user-defined objects, such as subVIs and type definitions, in the **Disabled** subdiagram of a Diagram Disable structure or in the inactive subdiagrams of a Conditional Disable or Type Specialization structure, LabVIEW does not load these objects into memory. However, when you display the block diagram of the VI, if LabVIEW cannot find the objects, the missing objects appear with a question mark icon. The VI does not break because LabVIEW does not include the code when compiling and executing the VI. Furthermore, LabVIEW does not automatically search for these missing objects when you enable the subdiagram that contains the subVI.

To force LabVIEW to locate missing subVIs in a Diagram Disable structure, enable the subdiagram that contains the missing subVI and save the VI that calls the subVI. The next time you open the VI that calls the subVI, LabVIEW searches for the subVI. You then can disable the subdiagram that contains the subVI again.

	Note LabVIEW checks the syntax of code in inactive subdiagrams. However, broken code within the inactive subdiagrams does not prevent the VI from compiling and executing.
	Note Like a Case structure, the disable structures support tunnels. However, by default you do not have to wire the output tunnels of disable structures in every case. All tunnels not connected by a wire use the <i>default value</i> for the tunnel data type. You can <i>configure the tunnels</i> to wire the input and output tunnels automatically in unwired cases.

Behavior of Disable Structures within Built Applications

Disable structures determine which sections of code are included in built applications. If a VI is only called in a disabled diagram, it will be excluded from the built application. LabVIEW does not reevaluate the condition on the Conditional Disable structure or the compilation result of the Type Specialization structure after an application has been built.

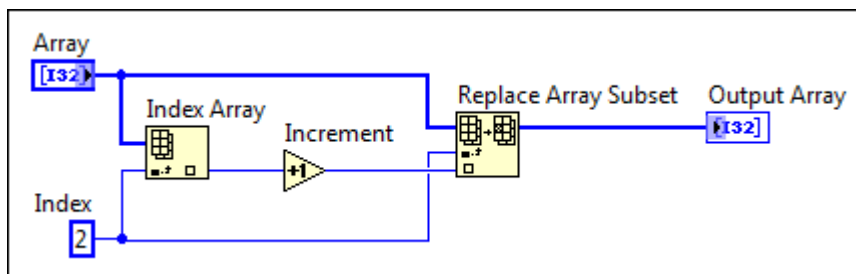
In Place Element Structures: Increasing Memory Efficiency

Use the In Place Element structure to control how the LabVIEW compiler performs common operations more efficiently than traditional LabVIEW programming methods and to increase memory and VI efficiency. Many LabVIEW operations require LabVIEW to copy and maintain data values in memory, thereby decreasing execution speed and increasing memory usage.

The In Place Element structure uses border nodes, or nodes that are attached to the border of the In Place Element structure, to perform operations on data. When you move or delete a border node, LabVIEW automatically moves or deletes the corresponding node on the opposite side of the structure.

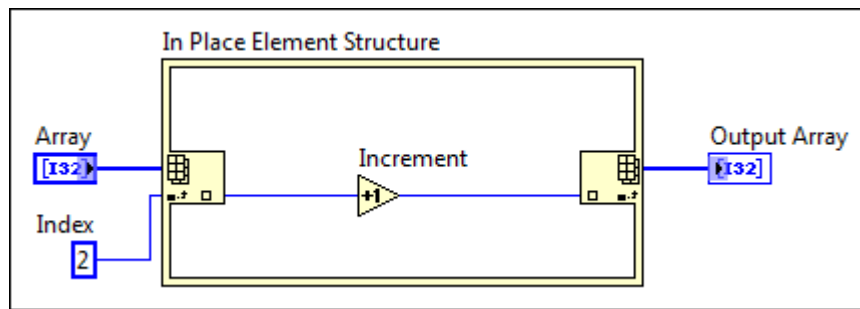
The In Place Element structure can execute common LabVIEW operations more efficiently than traditional LabVIEW programming methods.

The following block diagram shows a situation in which you can use the In Place Element structure to improve the execution and memory efficiency of a VI.



In the previous block diagram, LabVIEW indexes an array of 32-bit unsigned integers, increments the third element of the array, and replaces the element in the same location of the array. To replace the new value in the same array, the Replace Array Subset function requires you to wire the array and array index values to the **array** and **index** inputs of the function. When you run the VI, LabVIEW makes a new copy of the array element and stores the copy in memory.

Use the In Place Element structure to avoid making extra copies of the array element, as shown in the following block diagram.



In the previous block diagram, the Array Index / Replace Elements border node of the In Place Element structure indexes an array of 32-bit unsigned integers, increments the third element of the array, and replaces the element in the same location in the array, similar to the first block diagram example. However, in this instance, LabVIEW does not have to create or store extra copies of the element in memory.

The In Place Element structure requires you to replace each indexed element of the array in the array with the node on the right side of the structure. Because the structure requires you to replace the array element and the LabVIEW compiler knows which element to replace, LabVIEW does not have to create or store extra copies of the array elements in memory.

Controlling Timing in LabVIEW Applications (Real-Time, Windows)

Use timed structures to monitor and control the timing of your code. Timed structures work best with real-time applications and advanced Windows applications. For other applications, consider using the following objects instead of timed structures. These objects provide you with slightly less control over the timing of your code but require less configuration than timed structures.

- While Loop
- Wait (ms) function
- Wait Until Next ms Multiple function
- *Other loops and structures in LabVIEW*

Determining Which Timed Structure to Use

If you decide timed structures are appropriate for your application, the following information can help you decide which timed structure to use.

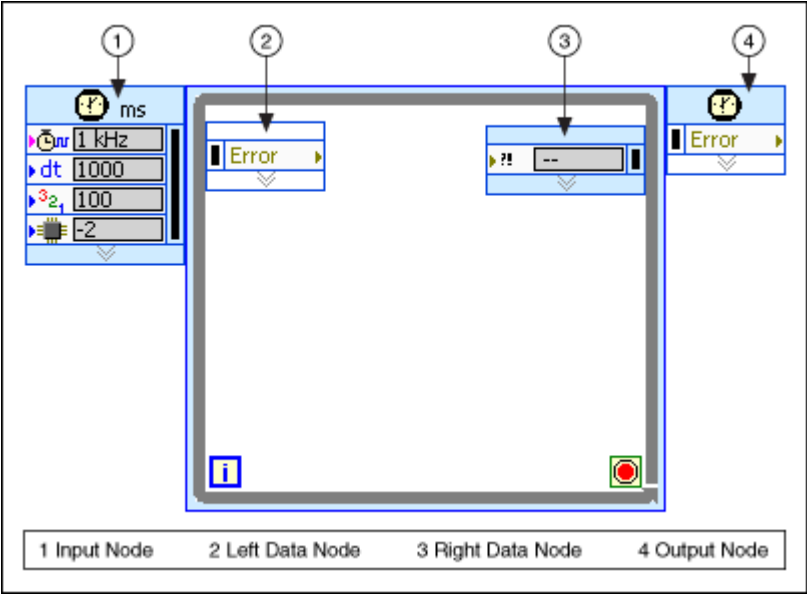
Timed Structure	Purpose of the Structure	Execution
Timed Loop	Timed Loops work well for: • Code you want to repeat according to specific timing characteristics • Code that needs to iterate or that has stop conditions	Executes code repeatedly until a stop condition is met.
Timed Loop with Frames	Timed Loops with frames work well for: • Code you want to run in sequence • Large applications—You can monitor the timing properties of complex code more easily because you receive timing information for each individual frame.	Executes code repeatedly, in sequence, until a stop condition is met. A Timed Loop with frames behaves like a Timed Loop that contains an embedded Sequence structure.
Timed Sequence	Timed Sequence structures work well for: • Code you want to execute only once with precise timing, execution feedback, timing characteristics that change dynamically, or several levels of execution priority	Executes code in sequence only once. Unlike Timed Loops, Timed Sequence structures do not repeat iterations.

Characteristics of Timed Structures (Real-Time, Windows)

All timed structures share the following characteristics:

- A resizable border to enclose a section of the block diagram code, also called a subdiagram
- Input and Output nodes that provide configuration data and return error and timing information
- Tunnels on the structure border that feed data into and out of the structure subdiagrams

The following image shows a Timed Loop on the block diagram.



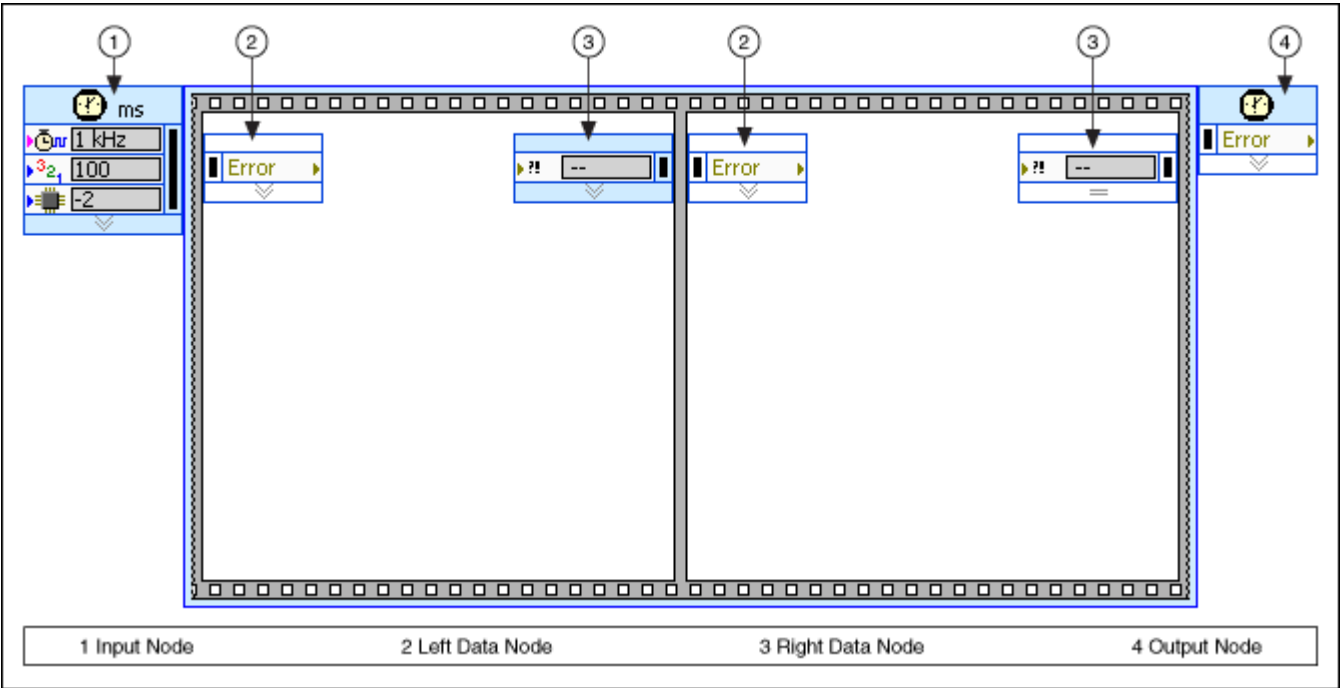
The iteration terminal, shown as follows, is an output terminal that returns the number of completed iterations.



The conditional terminal, shown as follows, is an input terminal. You can pass a Boolean value to the conditional terminal to stop a timed structure from running.



The following image shows a Timed Sequence structure on the block diagram.



As the images show, Timed Loops and Timed Sequence structures contain similar nodes. The following information describes the various nodes on a timed structure.



NOTE

You can use nodes to *configure a Timed structure*. You also can configure a Timed structure with various configuration dialog boxes. However, the configuration dialog boxes give you less precise control over the configuration options of a timed structure. For more advanced applications, you can use the configuration nodes, which provide additional configuration options.

1. Wire data to the Input node to configure the Timed structure. You also can right-click the Input node and select **Configure Input Node** from the shortcut menu to display the Configure Timed Loop dialog box.
2. The Left Data node returns configuration option values and provides timing information about the current or previous loop iteration or frame.
3. Use the Right Data node to configure the next loop iteration or frame. Right-click the Right Data node and select **Configure Input Node** from the shortcut menu to access the Configure Next Frame Timing dialog box. You also can wire data to the Right Data node to *configure the next loop iteration or frame dynamically*.
4. The Output node returns error information from the **Error** input of the Input node, error information generated by the structure during execution, and error information from the subdiagrams that execute within the Timed Loop frames. The Output node also returns timing and status information for the final loop iteration or frame.



Note By default, nodes of a timed structure do not display all of the available input and output terminals. You can resize nodes or right-click a node and use the shortcut menu to *display hidden node terminals*.



Timed Structures Home

Understanding Absolute and Relative Time in Timed Structures

A designated *timing source* governs timed structure execution. However, not all timed structure configuration options refer to the absolute value of the timing source. In Timed Loops with frames and Timed Sequence structures with frames, some configuration options are relative to the timing of the previous frame or the current frame. Options that refer to the start or finish of a frame execution use relative time, while options that refer to the timing source use absolute time.

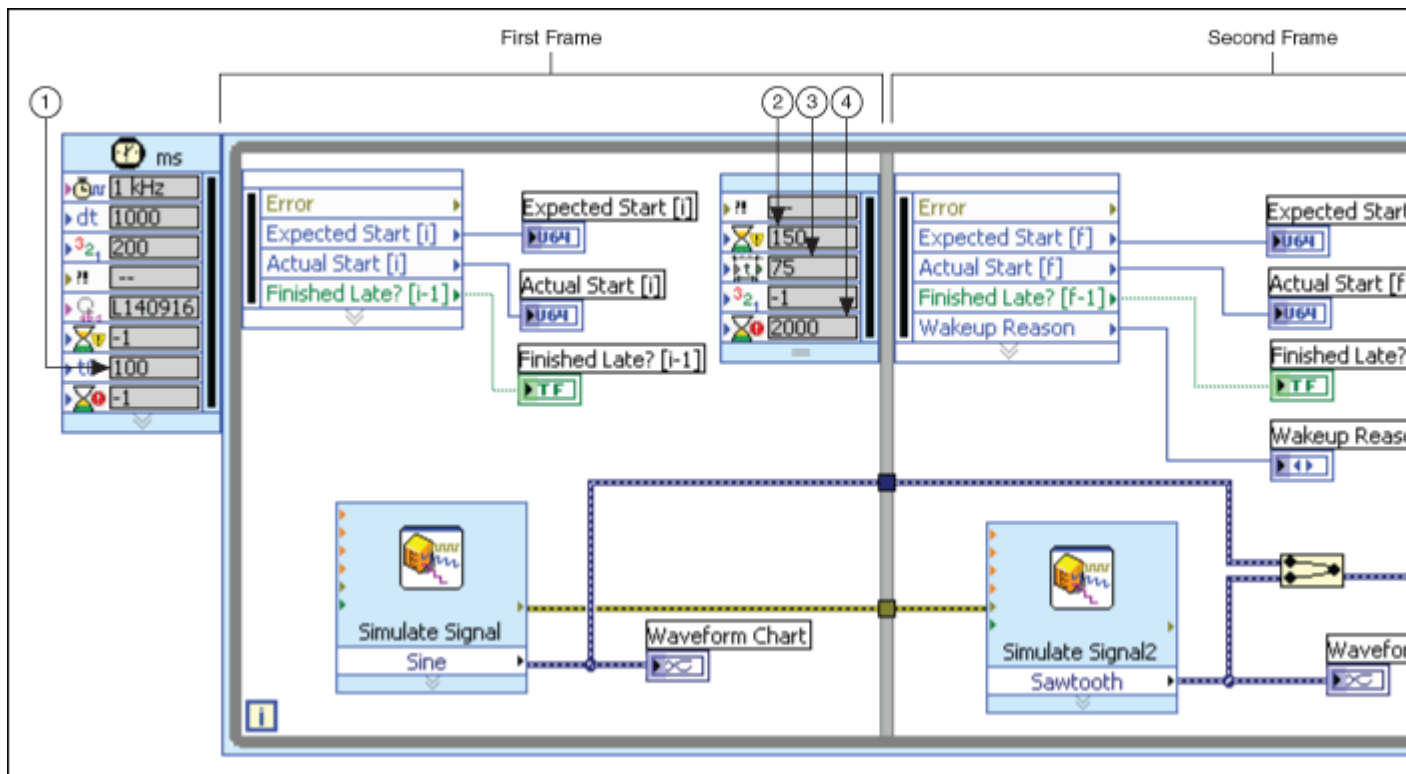
In timed structures, the offset is governed by absolute time, and outputs like **Expected Start**, **Expected End**, **Actual Start**, and **Actual End** are calculated in units of absolute time for frame and iteration timing. The start time, deadline, and timeout for each frame are governed by relative time. The start time and timeout are relative to the beginning of the previous frame execution, the deadline is relative to the beginning of the current frame.

In the following block diagram, a Timed Loop with frames governed by a 1 kHz clock has a period of 1000 ms and an offset of 100 ms. The offset is governed by absolute time, so the loop iterates when the timing source clock reaches 100 ms, 1100 ms, 2100 ms, 3100 ms, and so on, until the loop execution completes.



NOTE

A Timed Loop that is late can use absolute or relative time, depending on the *mode of the Timed Loop*.



1	Offset	Absolute time—refers to loop timing source
2	Deadline	Relative time—refers to start of second frame
3	Start	Relative time—refers to start of first frame
4	Timeout	Relative time—refers to start of end frame

The second frame of the Timed Loop has a start time of 75 ms. Since the start time is relative to the beginning of the execution of the previous frame, the frame execution can start no earlier than 75 ms after the first frame begins to execute. According to absolute time maintained by the timing source, the start time for the second frame is 175 ms in the first iteration, since the second frame executes no earlier than 75 ms after the first frame begins at 100 ms on the timing source clock.

The frame also has a timeout input of 2000 ms. The timeout establishes a maximum wait for the start of one frame after the beginning of the execution of the previous frame. If the second frame does not begin to execute within 2000 ms of the beginning of the execution of the first frame, LabVIEW executes the rest of the iteration untimed and returns Timeout

in the **Wakeup Reason** output on the Left Data node. If the second frame does begin to execute before the 2000 ms for the timeout passes, the deadline of 150 ms then places a limit on the amount of time the frame can spend executing. From the point in time when the second frame begins to execute, 150 ms can pass before the frame will output a TRUE value in the **Finished Late** indicator.



Configuring Timed Loops and Timed Loops with Frames (Real-Time, Windows)

To configure the way a Timed Loop executes, use the Configure Timed Loop or the Configure Timed Loop with Frames dialog box. To display these dialog boxes, double-click the Input node or right-click the node and select **Configure Input Node** from the shortcut menu. You can use these dialog boxes to *specify a timing source*, *set a period*, and *configure advanced options*.

You also can wire data to the Input node to configure the options of a Timed Loop. By default, the Input node of a Timed Loop does not display all of the available input terminals. You can resize the Input node or right-click the node and use the shortcut menu to *display hidden terminals*.

Tips and Tricks for Using Timed Loops

The following list provides additional information about using a Timed Loop.

- Unlike in a While Loop, you do not have to wire the conditional terminal in a Timed Loop.
- You can wire data to the Right Data node to *configure the options of the next loop iteration dynamically*.
- You can *add shift registers* to Timed Loops to pass data from the current iteration to the next iteration.
- To avoid unexpected timing behavior, do not add a VI configured to run at time-critical priority and a Timed Loop to the same block diagram.

Tips and Tricks for Using Timed Loops with Frames

The following list provides additional information about using a Timed Loop with frames.

- You can wire data to the Input node to *configure a Timed Loop with frames*, or you can right-click the Input node and select **Configure Input Node** from the shortcut menu to display the Configure Timed Loop with Frames dialog box and enter values for the options.
- Right-click the structure border to *add*, *delete*, and *merge* frames.

- If you want each frame to execute only once, right-click the Timed Loop structure border and select **Replace with Timed Sequence** from the shortcut menu.

Refer to the `labview\examples\Structures\Timed Loop\Timed Loop.lvproj` for examples of using Timed Loops.

Configuring the Next Frame

Double-click the Right Data node of the current frame or right-click the node and select **Configure Input Node** from the shortcut menu to display the Configure Next Frame Timing dialog box. You can use this dialog box to set a start time, a *priority*, *deadline*, and a *timeout* for the next frame.

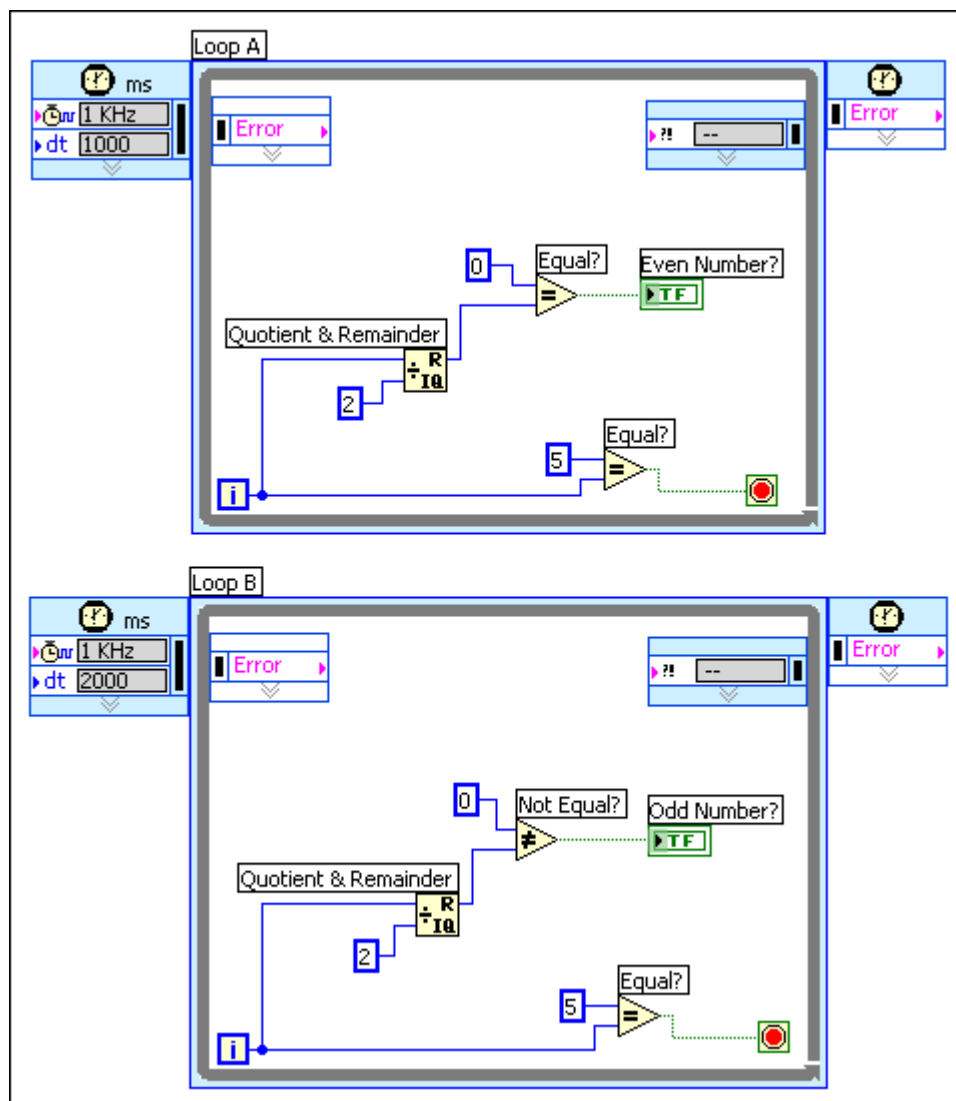
The start time specifies the time the next frame executes. You must specify a start time value relative to the start of the previous frame in absolute units of the frame timing source. Enter a value in the **Start** text box to specify a start time.

You also can use the input terminals of the Right Data node of a frame to *dynamically configure the options* for the next iteration of a Timed Loop or to *dynamically configure the options of the next frame*. By default, the Right Data node of a Timed Loop frame does not display all of the available input terminals. You can resize the Right Data node or right-click the node and use the shortcut menu to *display hidden terminals*.

Setting the Period of a Timed Loop

The period is the length of time between iterations of a Timed Loop specified in absolute units of the structure timing source.

The Timed Loops in the following block diagram use the default 1 kHz timing source. The period (dt) for Loop A is 1,000 ms and the period for Loop B is 2,000 ms, which means Loop A executes every second and Loop B executes every two seconds. Both Timed Loops stop executing after six iterations. Loop A stops executing after six seconds, and Loop B stops executing after 12 seconds.



The subdiagram of a Timed Loop or subdiagrams of a Timed Loop with frames must complete execution before the period specified for each iteration. If the subdiagram or subdiagrams do not complete execution before the specified period, the Timed Loop returns TRUE in the **Finished Late? [i-1]** output of the Output node or the **Finished Late? [i-1]** output of the Left Data node in the first frame of subsequent iterations.

If the subdiagram of the Timed Loop or subdiagrams of a Timed Loop with frames complete execution before the period of the loop, the Timed Loop sleeps and allows other code or structures to execute on the block diagram.



NOTE

Do not set the period of a Timed Loop to 0. National Instruments does not support applications that contain Timed Loops with periods of 0 because these periods cause undefined application behavior. If you need a loop that runs as fast as possible combined with the priority execution features of a Timed Loop, use a While Loop inside a Timed Sequence structure instead.

Setting the Mode of a Timed Loop

Occasionally, an iteration of a Timed Loop might execute later than the time you specified. The mode of a Timed Loop determines how the loop handles any late executions. Use the options in the **Action on Late Iterations** section of the Configure Timed Loop or Configure Timed Loop with Frames dialog box to specify the mode a Timed Loop uses to handle the late execution of a Timed Loop iteration. You also can use the **Mode** input of the Input node to specify the mode for a Timed Loop.

You can handle the late execution of a Timed Loop in the following ways:

- LabVIEW can align the execution with the original established schedule.
- LabVIEW can define a new schedule that starts at the current time.
- The Timed Loop can process the missed iterations.
- The Timed Loop can skip the missed iterations.

For example, if you set a Timed Loop with a period of 100 ms and an offset of 30 ms, you expect the first loop iteration to execute 30 ms after the first timing source starts running and in multiples of 100 ms after that—at 130 ms, 230 ms, 330 ms, and so on. However, the first execution of the Timed Loop might occur after 240 ms have elapsed. Because other Timed Loops or hardware devices might already be running at the schedule you specified, you can align the late Timed Loop with the already running global schedule, which means the Timed Loop can align itself as quickly as possible with the schedule you specified. In this case, the next Timed Loop iteration would run at 330 ms and continue to run in multiples of 100—at 430 ms, 530 ms, and so on. If aligning the Timed Loop with other Timed Loops or other hardware devices is not important, the Timed Loop can run immediately and use the current time as its actual offset. In this case, the subsequent loop iterations run at 240 ms, 340 ms, 440 ms, and so on.

If the Timed Loop is late, it might miss data other Timed Loops or hardware devices generate. For example, if the Timed Loop misses two iterations and some of the data from the current period, a buffer can hold the data from the missed iterations. You can configure the Timed Loop to process the missed data before it aligns with the schedule you specified. However, a Timed Loop that processes the missed iterations causes jitter, which is the variance between the loop cycle time and the time you specified. If you do not want to process the missed data, the Timed Loop can ignore older data in the buffer that the loop iterations missed and process only the latest data, such as the data available at the next period and the subsequent iterations.

Configuring Modes Using the Configuration Dialog Box

By default, the Timed Loop discards any data generated during missed iterations and maintains the original schedule. To display this mode setting, the icon for the **Mode** input of the Input node appears with a D for discard and an M for maintain. Remove the checkmark from the **Discard missed periods** checkbox in the Configure Timed Loop or Configure Timed Loop with Frames dialog boxes to process data any missed or late loop iterations generate. To display this mode setting, the icon for the **Mode** input of the Input node appears with P for process. Remove the checkmark from the **Maintain original phase** checkbox in the **Configure Timed Loop** or **Configure Timed Loop with Frames** dialog box to configure the Timed Loop to execute on a new schedule based on the first iteration of the Timed Loop.

Configuring Modes Using the Input Node

Use the **Mode** input of the Input node to configure the mode of a Timed Loop. Right-click the **Mode** input and select **Create»Constant** or **Create»Control** from the shortcut menu to create an enumerated constant or control. You can use this constant or control to select a mode.

Use the **Mode** input of the Input node to programmatically change the mode of a Timed Loop or maintain the configuration of the modes from the Configure Timed Loop dialog box. Select **No Change** in the enumerated constant or control to maintain the current mode configuration or select an option to handle any late or missed iteration of the Timed Loop.

Refer to the Timed Loop Mode VI in the `labview\examples\Structures\Timed Loop` for an example of setting the mode for a Timed Loop.



Configuring Timed Sequence Structures (Real-Time, Windows)

To configure the initial execution options of a Timed Sequence structure and to specify options for the first subdiagram, or frame, of the sequence, use the Configure Timed Sequence dialog box. Double-click the Input node or right-click the Input node and select **Configure Input Node** from the shortcut menu to display the **Configure Timed Sequence** dialog box. You can use this dialog box to [specify a timing source](#), [set the priority](#), and to [configure advanced options](#) for the Timed Sequence structure.

You also can wire values into the input terminals of the Input node to configure the options of a Timed Sequence structure. By default, the Input node of the Timed Sequence structure does not display all of the available input terminals. You can resize the Input node or right-click the Node and use the shortcut menu to [display hidden terminals](#).

Tips and Tricks for Using Timed Sequence Structures

The following list provides additional information about using a Timed Sequence structure.

- Right-click the Timed Sequence structure border to [add](#), [delete](#), and [merge](#) frames. Unlike in a Flat Sequence structure, every frame of a Timed Sequence structure must have all of its inputs before the structure can execute.
- The Right Data node of the last frame of a Timed Sequence structure does not include configuration options because you do not have to configure another frame or iteration.
- A Timed Sequence structure executes in the data flow of a block diagram ahead of any VI not configured to run at a time-critical priority. You can [change the priority of a Timed Sequence structure](#) to change the order in which it executes relative to other timed structures on the block diagram.

- To avoid unexpected timing behavior, do not add a VI configured to run at time-critical priority and a Timed Sequence structure to the same block diagram.
- If you want each frame of the Timed Sequence structure to execute more than once, you can right-click the Timed Sequence structure border and select **Replace with Timed Loop**.

Configuring the Next Frame

Double-click the Right Data node of the current frame or right-click the node and select **Configure Input Node** from the shortcut menu to display the Configure Next Frame Timing dialog box. You can use this dialog box to specify a start time, *a priority*, *a deadline*, and a *timeout* for the next frame.

The start time specifies the time the next frame starts executing. Specify a start time value relative to the start of the current frame in units of the frame timing source. Enter a value in the **Start** text box of the Configure Next Frame Timing dialog box to specify a start time. The next frame returns TRUE in the **Finished Late?** output of the Left Data node if the frame does not start before the specified start time.

You also can use the input terminals of the Right Data node of a frame to *configure the next frame or iteration of a Timed Loop dynamically*. By default, the Right Data node of a Timed Sequence structure frame does not display all of the available output terminals. You can resize the Right Data node or right-click the Node and use the shortcut menu to *display hidden terminals*.



Timed Structures Home

Configuring Settings of a Timed Structure (Real-Time, Windows)

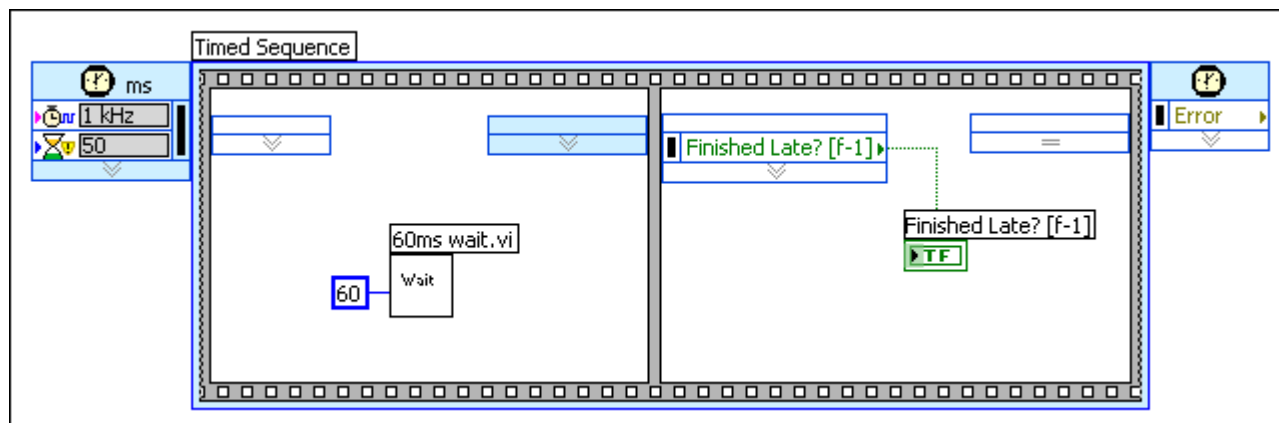
You can use the configuration dialog boxes of timed structures to configure advanced settings such as deadlines, timeouts, and secondary timing sources for timed structures. Double-click the Input node or right-click the Input node and select **Configure Input Node** from the shortcut menu to display the Configure Timed Loop, Configure Timed Sequence, or Configure Timed Loop with Frames dialog box.

Setting a Deadline

A deadline allots a specific amount of time for a subdiagram, or frame, to complete execution, relative to the start of the frame. Use a deadline to set a time limit reference for the execution of a subdiagram. If a frame does not complete before the deadline, the Left Data node of the next frame returns TRUE in the **Finished Late?** output and continues to execute. Specify a deadline value in the same units as the frame timing source.

In the following example, the first frame of the Timed Sequence structure is configured with a deadline of 50. The deadline specifies that the frame must complete execution before 50 ticks of the 1 kHz clock, or 50 ms. However, since there is a VI in the frame calling for a 60 ms wait, the code in the frame takes 60 ms to complete. The **Finished Late?** output

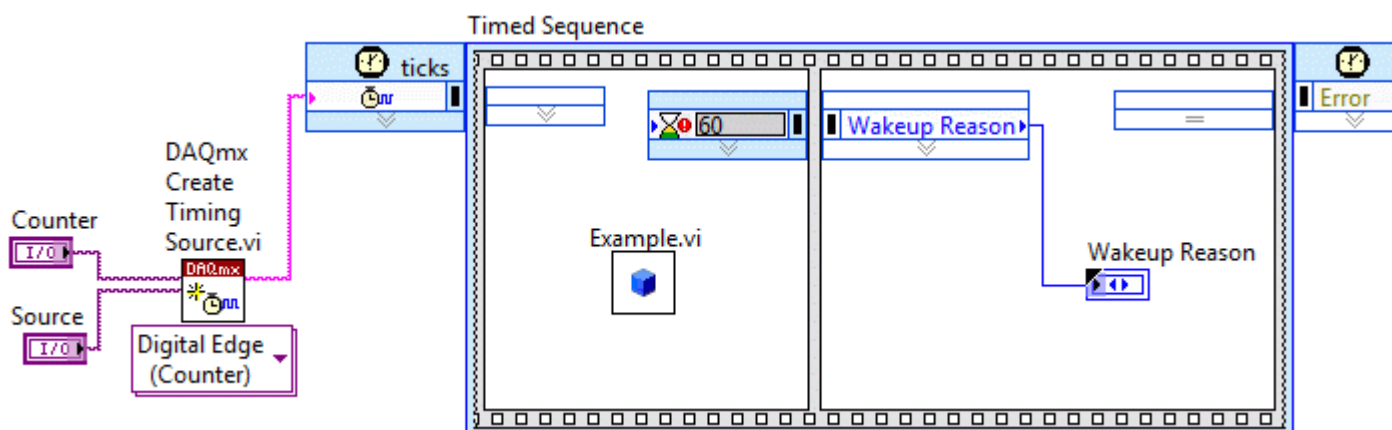
of the second frame will return TRUE because the frame execution does not meet the specified deadline.



Setting a Timeout

A timeout specifies the maximum amount of time, in milliseconds, a frame can wait to begin execution. The timeout value is relative to the start of the iteration or relative to the end of the previous frame. If the execution of the subdiagram does not begin before the specified timeout value, the Timed Loop returns **Timeout** in the **Wakeup Reason** output of the Left Data node for that frame.

You can use [several types of NI-DAQmx timing sources](#), including digital edge counters and signals from task sources, to control timed structures and to specify a timeout. In the following example, the second frame of the Timed Sequence structure executes when the DAQmx Create Timing Source VI receives a pulse from an external device. The external device sends a pulse every 50 ms under ideal conditions, but the pulse timing can vary. The timeout value of 60 ms for the second frame indicates that the frame can wait 60 ms to start after the beginning of the iteration. If the second frame does not start before 60 ms, the structure continues to execute the rest of the iteration untimed and the second frame returns **Timeout** in the **Wakeup Reason** output of the Left Data node.



If a timeout occurs in a timed structure, the structure continues to execute the rest of the iteration untimed. The timed structure returns **Timeout** in the **Wakeup Reason** output of the remaining frames. The timing information provided by the remaining frames equals the timing information for the frame where the timeout occurred. If a Timed Loop must

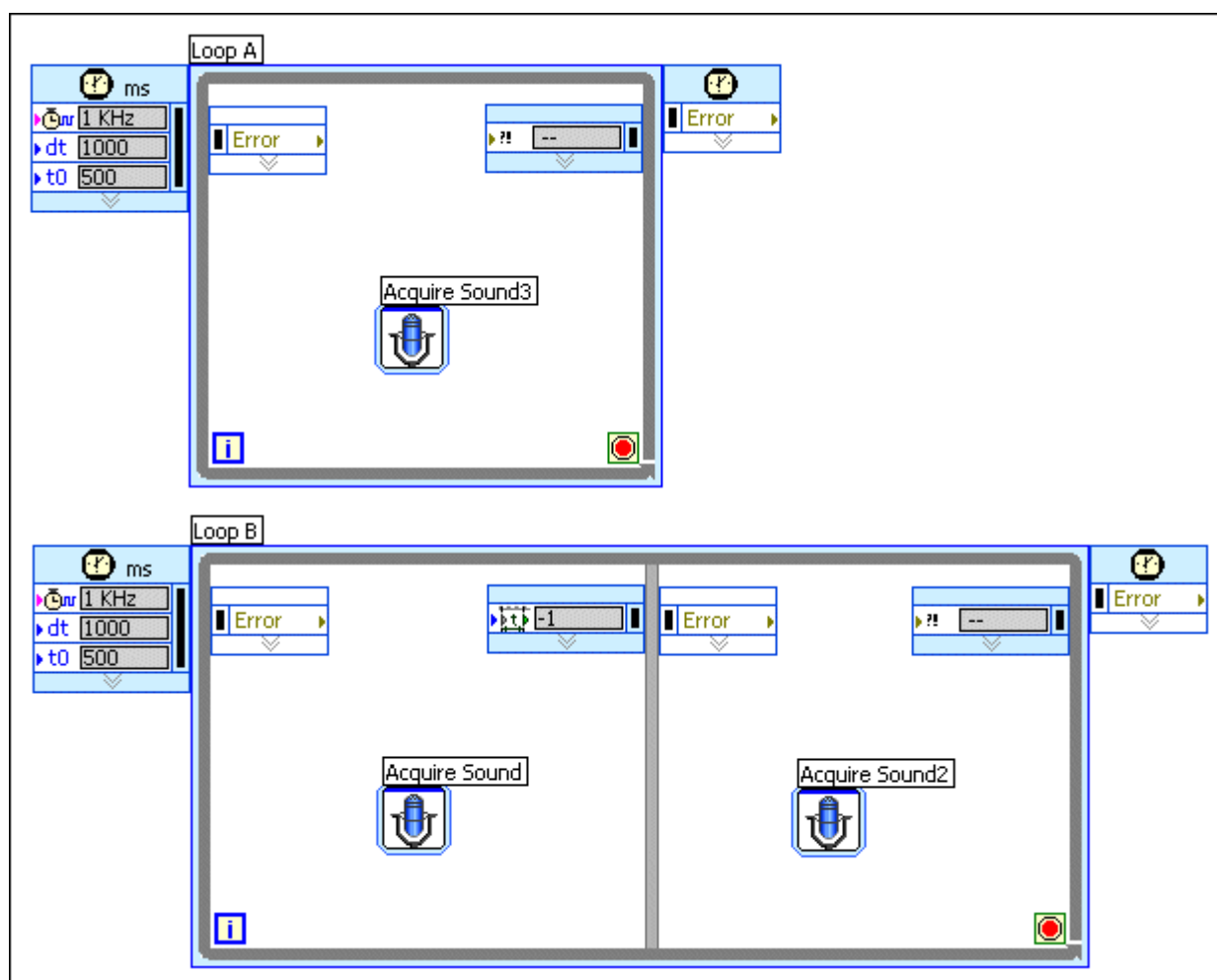
complete another iteration, the loop stops at the same frame where the timeout occurred and waits for the original timeout event.

The default timeout value for the first frame of a timed structure is -1 , which indicates to wait indefinitely for the start of the subdiagram, or frame. The default timeout value for other frames is 0 , which indicates no change from the timeout value of the previous frame.

Setting an Offset

An offset is the length of time, relative to the start of the timed structure, a structure waits to begin execution of the first subdiagram, or frame. The offset is defined in absolute units of the structure timing source.

You can align the phase of different timed structures using an offset and setting the same timing source for both structures. The Timed Loops in the following block diagram use the same 1 kHz timing source and have offset t_0 values of 500, meaning the Loops wait 500 ms after the timing source trigger to begin executing their iterations.



NOTE

Aligning the phase of two timed structures does not guarantee that the structures start at the same time. Use the Synchronize Timed Structure Starts VI to *synchronize the start of timed structures* on the block diagram.

In the last frame of a Timed Loop, you can use the Right Data node to *change the offset for subsequent iterations of the loop dynamically*. However, when you change the offset for subsequent iterations dynamically, you also must specify a mode by wiring a value to the **Mode** input of the Right Data node.



Note You cannot wire **No Change** to the **Mode** input if you change the offset using the right data node. You must select a mode value.

Refer to the Timed Loop Offset VI in `labview\examples\Structures\Timed Loop` directory for an example of setting the offset for a Timed Loop.

Assigning a Processor

By default, LabVIEW automatically assigns available processors to handle the execution of timed structures. However, you manually can load balance timed structures across multiple processors for further execution control.

Select **Manual** from the **Mode** pull-down menu in the **Processor Assignment** section of the Configure Timed Loop, Configure Timed Loop with Frames, Configure Timed Sequence, Configure Next Frame Timing, or Configure Next Iteration dialog box to manually configure a processor to handle a timed structure. You also must enter an integer between 0 and 255, where 0 represents the first available processor in the system, in the **Processor** field to specify which processor you want to handle the execution. If you attempt to enter a number out of the specified range, LabVIEW coerces it between 0 and 255. If you enter a number that exceeds the number of available processors, you generate a run-time error and the timed structure does not execute.

If you select **Automatic** from the **Mode** pull-down menu, LabVIEW automatically assigns a processor to the structure and the **Processor** value automatically sets to -2. If you select **No Change** from the **Mode** pull-down menu for a frame or subsequent iteration, LabVIEW uses the same processor that handled the previous frame or iteration and the **Processor** value automatically sets to -1.

You also can wire a value to the **Processor** input of the Input node of a timed structure or timed structure frame to set the processor assignment.

Refer to Optimizing RT Applications for Multiple-CPU Systems for more information about optimizing timed structure performance by manually assigning processors.

Setting a Secondary Timing Source

You can use a second timing source to control the subdiagrams, or frames, of a timed structure with a different clock source than the structure itself. Use the **Frame Timing Source** section of the Configure Timed Loop, Configure Timed Loop with Frames, or Configure Timed Sequence dialog boxes to select a secondary timing source.

The timed structures support the following secondary timing sources:

- **Use timing source of loop or sequence** —Use the same timing source of the timed structure to time the frames.
- **Use built-in source from below** —Use a separate built-in timing source to time the frames.
 - **1 kHz Clock <reset at structure start>** —A timing source similar to the 1 kHz clock that resets to 0 after every iteration of a timed structure.
 - **1 MHz Clock <reset at structure start>** —A timing source similar to the 1 MHz clock that resets to 0 after every iteration of a timed structure.

Absolute and Relative Time in Timed Structures

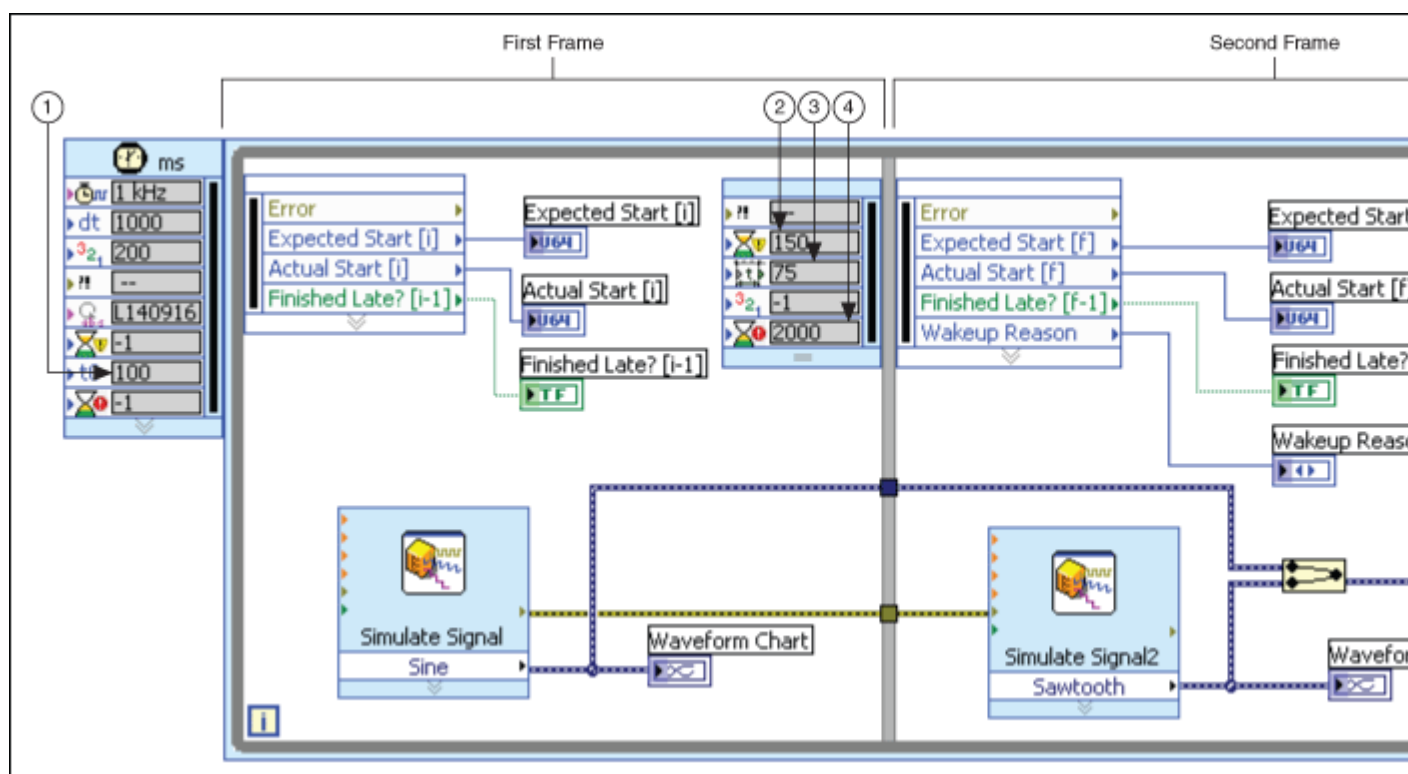
A designated *timing source* governs timed structure execution. However, not all timed structure configuration options refer to the absolute value of the timing source. In Timed Loops with frames and Timed Sequence structures with frames, some configuration options are relative to the timing of the previous frame or the current frame. Options that refer to the start or finish of a frame execution use relative time, while options that refer to the timing source use absolute time.

In timed structures, the offset is governed by absolute time, and outputs like **Expected Start**, **Expected End**, **Actual Start**, and **Actual End** are calculated in units of absolute time for frame and iteration timing. The start time, deadline, and timeout for each frame are governed by relative time. The start time and timeout are relative to the beginning of the previous frame execution, the deadline is relative to the beginning of the current frame.

In the following block diagram, a Timed Loop with frames governed by a 1 kHz clock has a period of 1000 ms and an offset of 100 ms. The offset is governed by absolute time, so the loop iterates when the timing source clock reaches 100 ms, 1100 ms, 2100 ms, 3100 ms, and so on, until the loop execution completes.



Note A Timed Loop that is late can use absolute or relative time, depending on the *mode of the Timed Loop*.



1	Offset	Absolute time—refers to loop timing source
2	Deadline	Relative time—refers to start of second frame
3	Start	Relative time—refers to start of first frame
4	Timeout	Relative time—refers to start of end frame

The second frame of the Timed Loop has a start time of 75 ms. Since the start time is relative to the beginning of the execution of the previous frame, the frame execution can start no earlier than 75 ms after the first frame begins to execute. According to absolute time maintained by the timing source, the start time for the second frame is 175 ms in the first iteration, since the second frame executes no earlier than 75 ms after the first frame begins at 100 ms on the timing source clock.

The frame also has a timeout input of 2000 ms. The timeout establishes a maximum wait for the start of one frame after the beginning of the execution of the previous frame. If the second frame does not begin to execute within 2000 ms of the beginning of the execution of the first frame, LabVIEW executes the rest of the iteration untimed and returns **Timeout** in the **Wakeup Reason** output on the Left Data node. If the second frame does begin to execute before the 2000 ms for the timeout passes, the deadline of 150 ms then places a limit on the amount of time the frame can spend executing. From the point in time when the second frame begins to execute, 150 ms can pass before the frame will output a **TRUE** value in the **Finished Late** indicator.

Setting the Priority of a Timed Structure (Real-Time, Windows)

The priority of a timed structure specifies when the structure executes on the block diagram relative to other timed structures on the block diagram. Use the priority setting of a timed structure to write applications with multiple tasks that can preempt each other in the same VI. The higher the priority of a timed structure, the earlier the structure executes relative to other timed structures. The value for the Priority input must be a positive integer between 1 and 65,535.

Each timed structure on the block diagram creates and runs in its own execution system that contains a single thread, so no parallel tasks can occur. A Timed Loop executes in the data flow of a block diagram ahead of any VI not configured to run at a time-critical priority.



NOTE

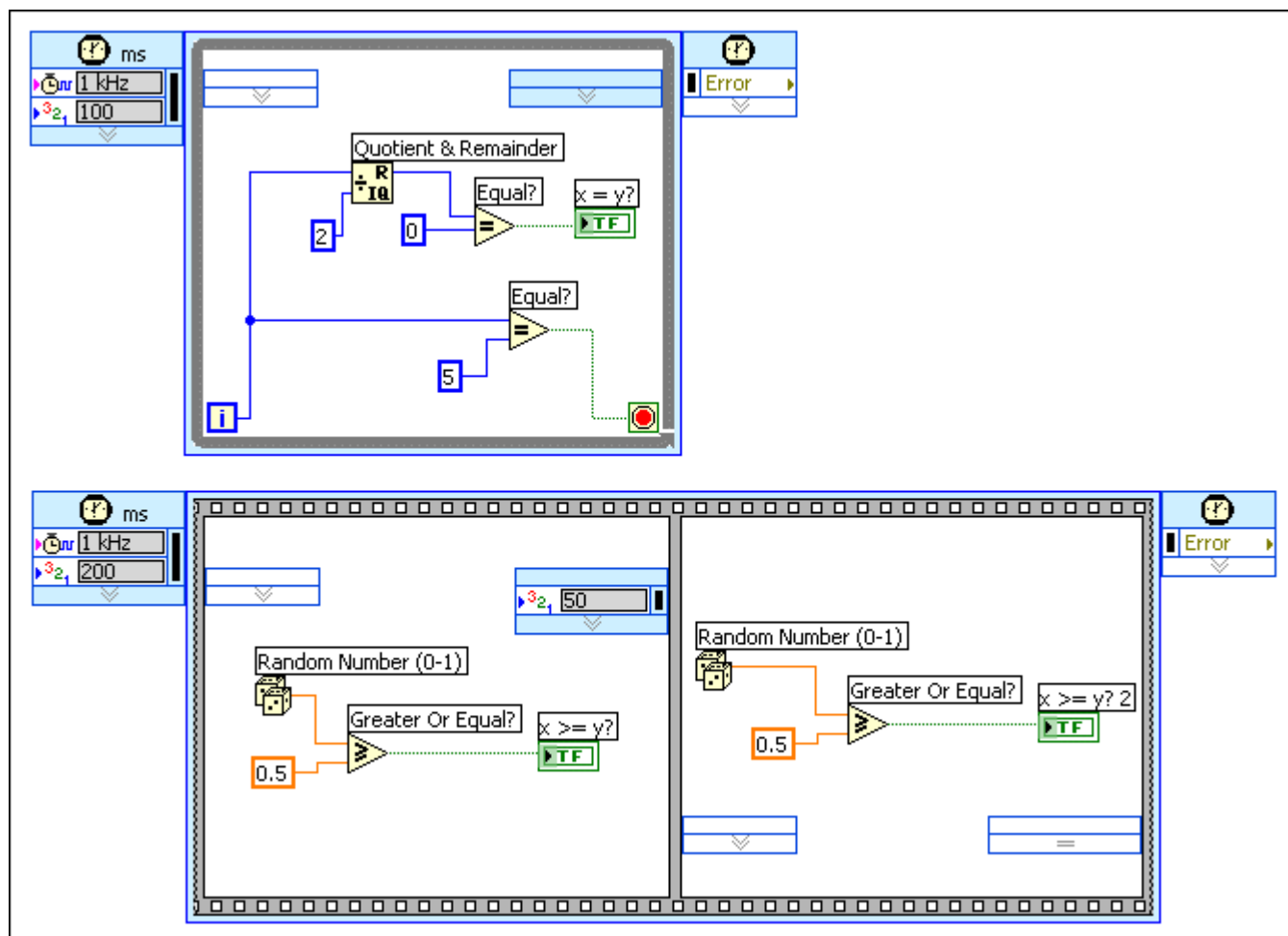
Adding a VI configured to run at time-critical priority and a Timed Loop on the same block diagram can lead to unexpected timing behavior.

You can assign a priority to each frame of a Timed Sequence structure or Timed Loop with frames. When you run a VI that contains timed structures, LabVIEW checks the priority of any frame ready to execute on the block diagram and first executes the frame with the highest priority.



Note (NI Linux Real-time) Do not set more than 32 timed structure priority levels in a VI. LabVIEW returns an error if you attempt to set more than 32 timed structure priority levels.

The following block diagram contains a Timed Loop and a Timed Sequence structure. The priority value of the first frame of the Timed Sequence structure (200) is higher than the priority value of the Timed Loop (100), so the first frame of the Timed Sequence structure executes first.



After the first frame of the Timed Sequence structure executes, LabVIEW compares the priority of other structures or frames that are ready to execute. The Timed Loop priority (100) is higher than the second frame of the Timed Sequence structure (50). In this example, the Timed Loop will execute completely before the second frame of the Timed Sequence structure executes.

You can set the priority of a timed structure by right-clicking the Input node and selecting **Configure Input Node**, then changing the **Priority** value. You also can *dynamically set the priority of subsequent iterations* of a Timed Loop by wiring a value to the Priority input of the Right Data node in the last frame. You can *dynamically set the priority of the next frame* of a timed structure by wiring a value to the Right Data node of the current frame. By default, the Right Data node of a frame does not display all of the available output terminals. You can resize the Right Data node or right-click the node and use the shortcut menu to *display hidden terminals*.



Note (NI Linux Real-Time) National Instruments recommends interactively changing timed structure priority levels. Dynamically changing timed structure priority levels can decrease performance.



[Timed Structures Home](#)

Selecting a Timing Source for a Timed Structure (Real-Time, Windows)

A timing source controls the execution of a timed structure. You can select from three types of timing sources:

- **Internal** —Built-in timing sources that you select using the configuration for the Input node of a timed structure
- **Software-Triggered** —Timing sources you create using the Create Software-Triggered Timing Source instance of the Create Timing Source VI
- **External** —Timing sources that you create using the Create Timing Source VI and the DAQmx - Data Acquisition VIs

Internal Timing Sources

Internal timing sources for controlling a timed structure include the 1 kHz clock of the operating system and the 1 MHz clock of a supported real-time (RT) target. Use the **Loop Timing Source** or **Sequence Timing Source** section of the Configure Timed Loop, Configure Timed Sequence, or Configure Timed Loop with Frames dialog boxes to select an internal timing source.

- **1 kHz Clock** —By default, a timed structure uses the 1 kHz clock of the operating system as the timing source. Using the 1 kHz clock, you can schedule a timed structure with millisecond resolution. All LabVIEW platforms that can run a timed structure support the 1 kHz timing source.
- **1 MHz Clock** —Supported targets can use the 1 MHz clock timing source to control a timed structure. Using the 1 MHz clock, you can schedule a timed structure with microsecond resolution. If the target does not support microsecond resolution due to processor or operating system limitations, the 1 MHz clock is not available.
- **1 kHz <absolute time>** —Selects the 1 kHz clock of the operating system. When you select this timing source, the timed structure uses timestamps for all input and output terminals related to start and finish times. You can use an absolute timing source to specify a specific date and time for each execution of the structure.
- **1 MHz <absolute time>** —Selects the 1 MHz clock supported on RT targets with Intel Pentium III or later-class processors, such as the NI PXI-817x and NI PXI-818x devices. When you select this timing source, the timed structure uses timestamps for all input and output terminals related to start and finish times. You can use an absolute timing source to specify a specific date and time for each execution of the structure.
- **1 kHz Clock <reset at structure start>** —A timing source similar to the 1 kHz clock that resets to 0 ticks when the timed structure starts.
- **1 MHz Clock <reset at structure start>** —A timing source similar to the 1 MHz clock that resets to 0 ticks when the timed structure starts.
- **Synchronize to Scan Engine** —Synchronizes the timed structure to the [NI Scan Engine](#). Using this timing source, the timed structure executes at the end of each scan. The period between iterations corresponds to the **Scan Period (μS)** setting you configure on the Scan Engine page.

Software-Triggered Timing Sources

You can create a software-triggered timing source to trigger a timed structure based on a software-defined event. Use the Create Timing Source VI to create a software-triggered timing source. Use the Fire Software-Triggered Timing Source VI to programmatically trigger a Timed Loop controlled by a software-triggered timing source.

You can use a software-triggered timing source as an RT-compatible event handler or to notify a consumer Timed Loop when new data becomes available in a producer-consumer application. You also can use software-triggered timing sources for discrete event simulation. Use the **number of ticks** input of the Fire Software-Triggered Timing Source VI to specify the number of ticks that elapse between each discrete event. The internal tick counter of the timed structure advances by **number of ticks** when the Fire Software-Triggered Timing Source VI executes. If the internal tick counter jumps past one or more Timed Loop periods, the Timed Loop regards those periods as missed periods. If you want to trigger multiple Timed Loop iterations with a single call to the Fire Software-Triggered Timing Source VI, you can remove the checkmark from the **Discard missed periods** checkbox in the Configure Timed Loop dialog box.

External Timing Sources

You can create external timing sources for controlling a timed structure with NI-DAQmx 7.2 or later. Use the DAQmx Create Timing Source VI to programmatically select an external timing source. You also can use several types of NI-DAQmx timing sources, including frequency, digital edge counters, digital change detection, and signals from task sources, to control timed structures. Use the DAQmx - Data Acquisition VIs to create the following types of NI-DAQmx timing sources to control a timed structure.

- **Frequency**—Creates a timing source that causes a timed structure to execute at a constant frequency.
- **Digital Edge Counter**—Creates a timing source that causes a timed structure to execute on rising or falling edges of a digital signal.
- **Digital Change Detection**—Creates a timing source that causes a timed structure to execute on rising or falling edges of one or more digital lines.
- **Signal from Task**—Creates a timing source that uses the signal you specify to determine when a timed structure executes.

Refer to the NI-DAQmx Help, available by selecting **Start»All Programs»National Instruments»NI-DAQ»NI-DAQmx Help**, for information about using NI-DAQmx VIs and functions to control timed structures.

You can *use a secondary timing source* to control the execution of the frames of a timed structure. For example, you can use a 1 kHz clock to control the iterations of a Timed Loop and use a 1 MHz clock to control the timing of the frames in every iteration.



Setting the Input Options of a Timed Structure Dynamically (Real-Time, Windows)

Use the Right Data node of a frame to dynamically set the input values for the next frame of a timed structure or to set the input values for subsequent iterations of a Timed Loop.

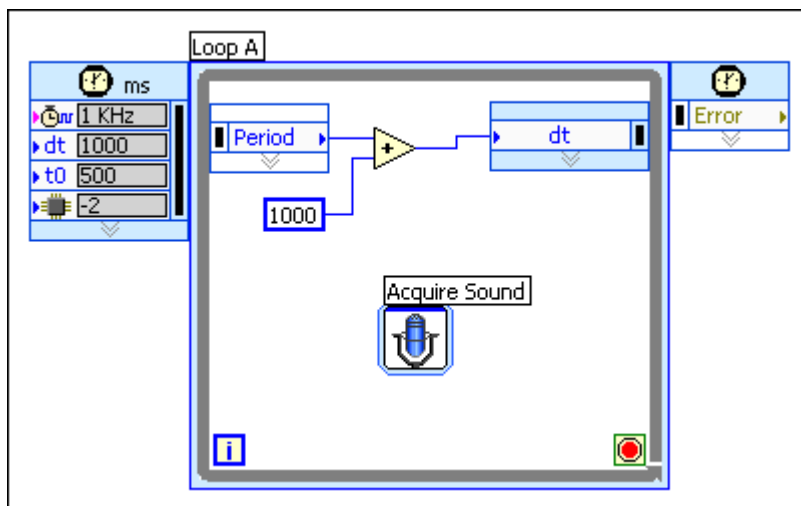


NOTE

(NI Linux Real-Time) National Instruments recommends interactively changing timed structure priority levels. Dynamically changing timed structure priority levels can decrease performance.

Configuring the Next Iteration of a Timed Loop

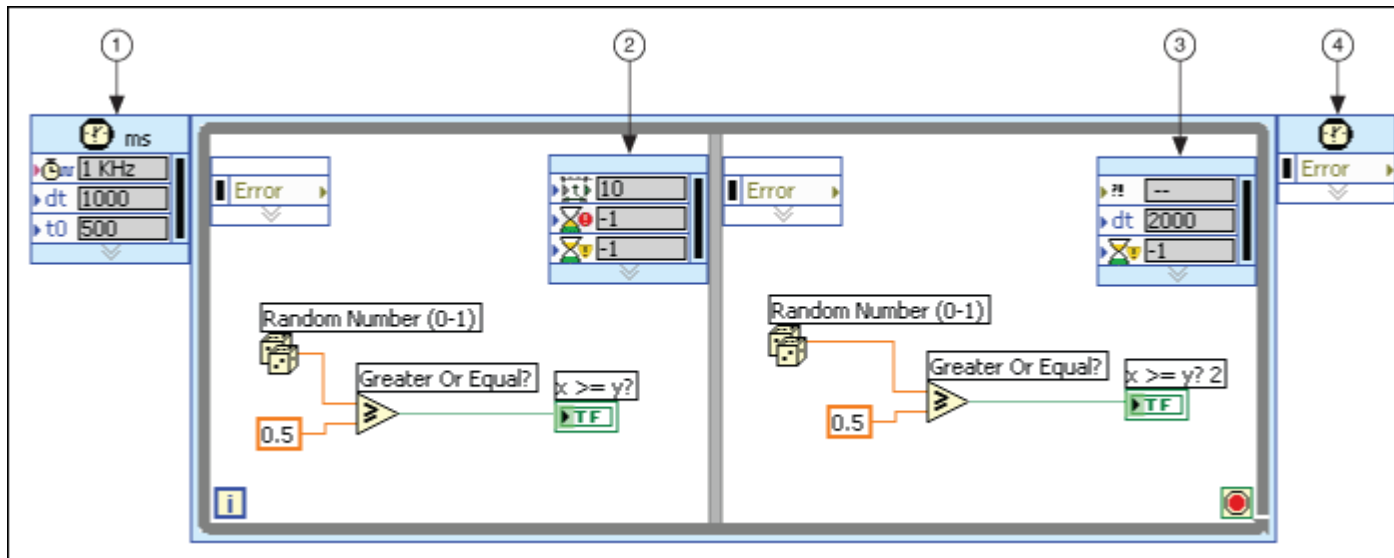
Use the Right Data node of a Timed Loop or the Right Data node of the last frame of a Timed Loop with frames to set the configuration options for the next iteration of the loop. The Timed Loop in the following block diagram uses the **Period** input of the Right Data node to configure the loop to run one second (1,000 ms) longer each time the loop iterates.



Note If you dynamically change the offset of a Timed Loop by wiring a value to the **Offset** input of the Right Data node, you also must specify a mode with the **Mode** input of the Right Data node.

Configuring the Next Frame of a Timed Structure

You also can use the Right Data node of a frame to change the configuration options of the next frame dynamically. The Timed Loop in the following block diagram includes two frames—A and B. The Input node (1) configures the initial execution of the Timed Loop and the options of the first frame. The Right Data node of the first frame configures the options of second frame. The Right Data node of the second frame configures the options of the first frame and the next loop iteration.

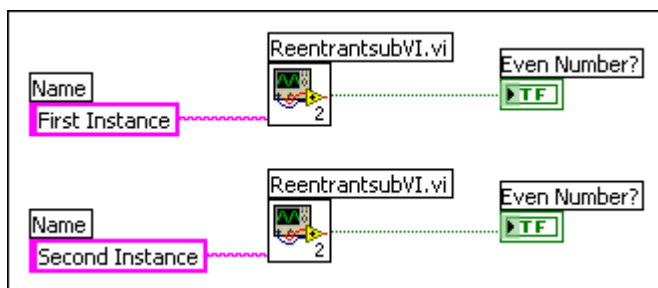


1	Input node
2	Right Data node
3	Right Data node
4	Output node

Naming Timed Structures (Real-Time, Windows)

By default, LabVIEW automatically uniquely identifies each timed structure you place on the block diagram with a name, which appears in the **Structure Name** text box of the Configure Timed Loop or Configure Timed Loop with Frames dialog box or the **Sequence name** text box of the Configure Timed Sequence dialog box. You can rename the timed structure by entering a new name in the text box. You can use the unique name of a timed structure with the VIs on the Timed structures and VIs palette to programmatically stop the structure and to synchronize a group of timed structures to use the same start time.

If a **reentrant VI** includes a timed structure and you use two or more instances of that reentrant VI as subVIs on a block diagram, you can programmatically change the name of the timed structure for each instance of the reentrant VI. To configure the reentrant VI, connect an input terminal on the connector pane to a string control wired to the **Name** input of the Timed Loop or Timed Sequence structure on the block diagram. On the block diagram where two or more instances of the reentrant VI are used as subVIs, wire unique string values to the **Name** input on the reentrant subVIs to uniquely identify each timed structure within each instance of the reentrant subVIs. For example, the following block diagram includes two instances of the same reentrant VI as subVIs. The **Name** string constants wired to the instances of the subVIs pass two different names to the Timed Loop in the block diagram of the reentrant VI each time the block diagram executes.



Synchronizing the Start of Timed Structures (Real-Time, Windows)

You can synchronize the start of timed structures running on a single computing device or on multiple distributed computing devices connected over Ethernet.

Synchronizing Timed Structures Running on a Single Computing Device

If you are using *I/O variables* to access I/O data inside of the timed structures you plan to synchronize, you can *synchronize timed structures to the NI Scan Engine*.

If you are not using I/O variables to access I/O inside of the timed structures you plan to synchronize, use the Synchronize Timed Structure Starts VI to synchronize the start of timed structures running on a single computing device. For example, if you have two timed structures on the block diagram, you can ensure that they execute on the same schedule relative to each other. You can configure the first timed structure to execute first and generate data. You then can configure the second timed structure to process that data when an iteration of the first structure finishes. You can synchronize the start of the timed structures to ensure that both structures use the same start time as the basis for their execution.

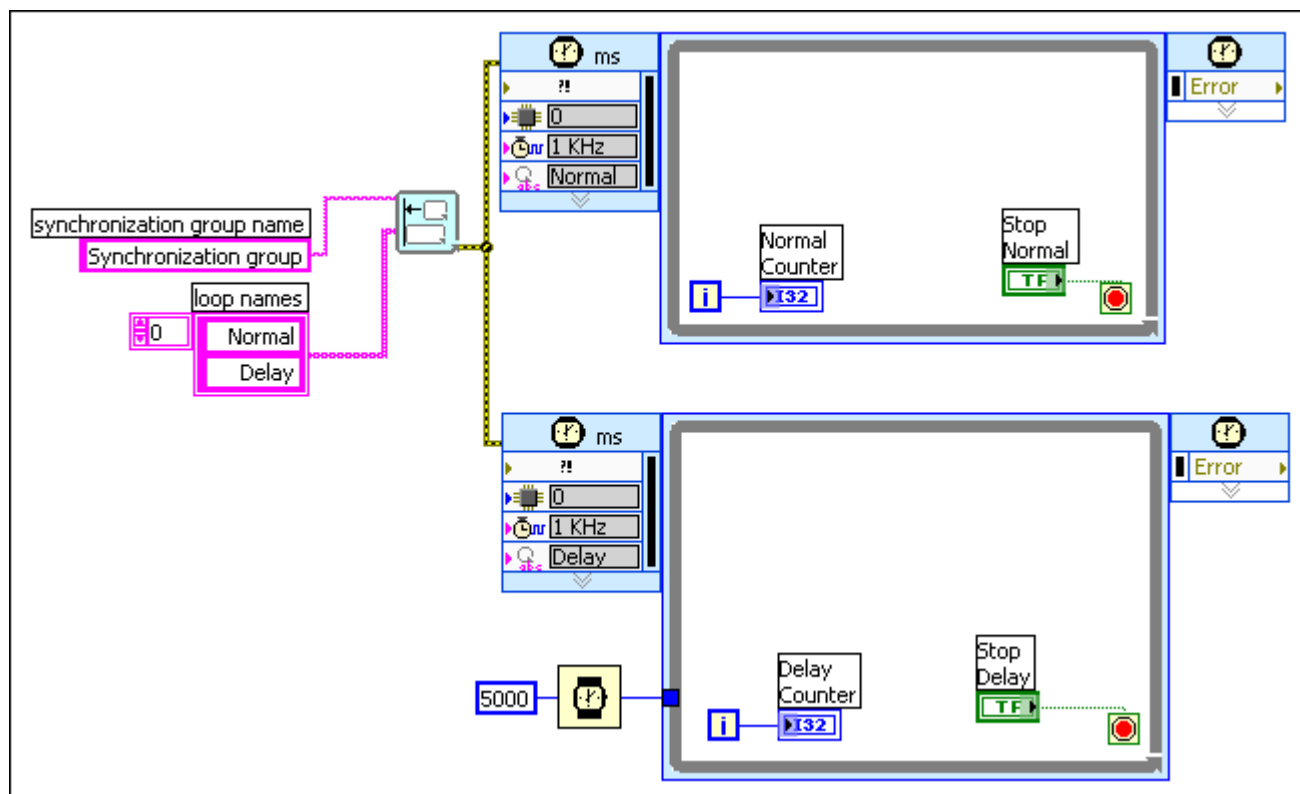


NOTE

National Instruments recommends that you use the Synchronize Timed Structure Starts VI only with timed structures configured to use either the **1 kHz Clock** timing source or the **1 MHz Clock** timing source. National Instruments does not recommend using this VI with absolute time timing sources, reset at structure start timing sources, the Synchronize to Scan Engine timing source, or any external timing source.

You can create synchronization groups to specify the structures on the block diagram that you want to synchronize. You can create a synchronization group by wiring a name for the group to the **synchronization group name** input and wiring an array of the *timed structure names* to the **timed structure names** input of the Synchronize Timed Structure Starts VI. The synchronization groups you create remain active until the VI completes execution.

The Synchronize Timed Structure Starts VI in the following block diagram creates a synchronization group and synchronizes two Timed Loops, Normal and Delay, to use the same start time.



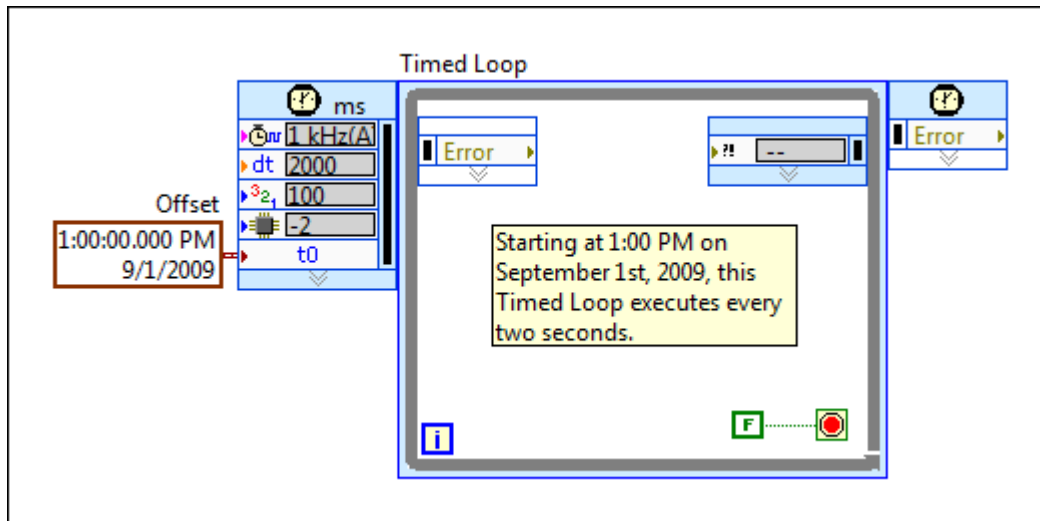
A timed structure cannot belong to more than one synchronization group. If you attempt to add a timed structure that already belongs to a previous group to a new synchronization group, LabVIEW removes the structure from the previous group and adds it to the new group. You can set the **replace (T)** input of the Synchronize Timed Structure Starts VI to FALSE to prevent the movement of a timed structure to the synchronization group if the structure already belongs to another synchronization group. If you set the **replace (T)** input to FALSE and then attempt to include a timed structure already included in another synchronization group, LabVIEW returns an error.

Synchronizing Timed Structures Running on Multiple Computing Devices

If you are using *I/O variables*, you can *synchronize timed structures to the NI Scan Engine* on each device that supports the NI Scan Engine. Because each instance of the NI Scan Engine starts executing at an absolute time that is a multiple of the scan period you configure, you can synchronize the phases of timed structures on multiple targets by setting the scan period on each target to a common value or a multiple of a common value.

If you are not using I/O variables, use absolute time timing sources to synchronize the start of timed structures running on one or more distributed computing devices connected over Ethernet. When you select either **1 kHz <absolute time>** or **1 MHz <absolute time>** as the timing source of a timed structure, you can specify the exact date and time to start the structure by wiring a LabVIEW timestamp control or constant to the **Offset** input of the timed structure. By wiring the same timestamp value to multiple timed structures, you can synchronize the start time of the structures.

The Timed Loop in the following block diagram uses the **1 KHz <absolute time>** timing source with an **Offset** configured to start at 1:00 p.m. on September 1, 2009 and a **period** configured to execute every two seconds.

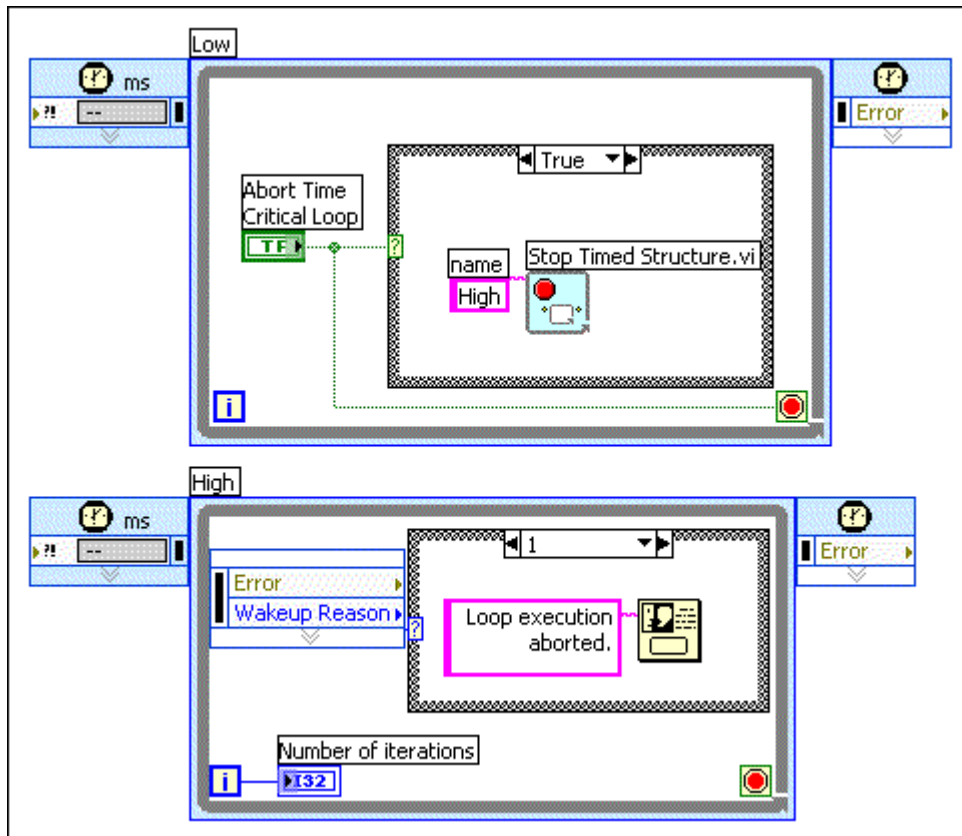


Note Both the NI Scan Engine and the absolute time timing sources rely on the system clock of the computing device on which the timed structure runs. Therefore, the precision to which you can synchronize timed structures depends on the degree of synchronization of the system clocks themselves. You might be able to improve clock synchronization by setting the system clock of each device to a common master clock. Refer to the National Instruments Web site for information about NI time synchronization software based on the IEEE 1588 specification.

[Timed Structures Home](#)

Aborting the Execution of a Timed Structure (Real-Time, Windows)

Use the Stop Timed Structure VI to abort the execution of a timed structure programmatically. Specify the name of the timed structure you want to abort by wiring the structure name in a string constant or control to the **name** input of the Stop Timed Structure VI. For example, in the following block diagram, the lower Timed Loop includes the Stop Timed Structure VI. The upper Timed Loop runs and displays the number of iterations it has completed. If the user clicks the **Abort Time Critical Loop** button on the front panel, the **Wakeup Reason** output of the Left Data node returns a value of **Aborted**, a dialog box appears, and when the user clicks **OK** in the dialog box, the VI stops running.



 [Timed Structures Home](#)

Events in LabVIEW

Types of Events

LabVIEW generates events from the following sources:

- User Interface—User interface events include mouse clicks, key presses, and so on.
- External I/O—External I/O events include hardware timers or triggers that signal when acquisition completes or when an error condition occurs.



NOTE

LabVIEW also supports ActiveX and .NET Framework events, which are external I/O events that you can generate programmatically.

- Block diagram—Block diagram events, or programmatically generated events, include [user events](#), which allow you to generate events with data you define, and other events that occur from the block diagram.

Each event is associated with a control on the front panel of the VI, the front panel window of the VI, a pane of the VI, or the LabVIEW application.



Note LabVIEW generates events for a VI only while that VI runs or when another running VI calls that VI as a subVI.

User Interface Events

With user interface events, you can design an application so LabVIEW generates an event each time a user performs a particular action on the front panel. Each time that event occurs on the front panel, the block diagram responds according to the code you write for that event. Compared to polling the user interface in a loop for interactions, user interface events reduce the CPU requirements of the program, simplify the block diagram code, and ensure that the block diagram can respond to all interactions the user makes. LabVIEW categorizes user interface events into two different *types of events*.

Programmatically Generated Events

With programmatically generated events, you can generate events from the block diagram without user actions on the front panel. Programmatically generated events communicate among different parts of the program that have no dataflow dependency. For example, one part of an application may run a continuous loop to monitor a temperature sensor and generate a user event when the sensor reaches an alarm level. In a different part of the same application, an Event structure executes to handle both the user event generated when the sensor reaches the alarm level and a user interface event that the Event structure handles to notify the user of the alarm condition. Handling both user interface events and programmatically generated events in the same Event structure allows you to implement more advanced architectures, such as queued state machines using events.



Note In general, LabVIEW generates user interface events only as a result of direct user interaction with the active front panel. However, you can use the Value (Signaling) property to generate a Value Change event programmatically.

Event-Driven Programming

In LabVIEW, data flow determines when block diagram elements execute. A node executes after receiving all required inputs, and then the node generates data to pass to the next node in the dataflow path. Event-driven programming allows you to influence this execution with an event, or an input that provides an asynchronous notification that something occurred. For example, if you want an event to occur when the user presses a specific key on the front panel, you can design an application to include a loop that waits until the key press event occurs, handles that event, and returns to wait for the next event. While the loop waits for the event to occur, other parts of the application can continue to run. How the application handles each event depends on the code written for that specific event. Some sections of the program may execute frequently because the events those sections handle occur frequently, and other sections of the program may not execute at all because the events never occur.

Frequently Asked Questions

How can I use events in LabVIEW?

How can I make sure my application responds to events?

How can I specify which events I want an application to monitor?

How can I monitor and control only certain events?

How can I create and generate an event that carries customized data?

How can I most effectively design an application to include events?

How can I view the events in an event queue while an application runs?

Determining Which Type of User Interface Events to Use

LabVIEW categorizes **user interface events** into two different types of events: notify and filter. The following table describes these types of events and when to use each type:

Event Type	Description	Example	Use Case
Notify	Notify events inform you that a user action occurred.	You can configure an Event structure to execute to handle a Value Change event when the user changes the value of a control.	Use a notify event if you only want to know that the user performed a particular action.
Filter	Filter events allow you to validate or change the event data before LabVIEW performs the default action associated with that event. You also can discard the event entirely to prevent the change from affecting the VI.	You can configure an Event structure to discard the Panel Close? event, preventing the user from interactively closing the front panel of the VI.	Use a filter event if you want to participate in the handling of the user action.



NOTE

Filter events have names that end with a question mark to help you distinguish them from notify events. LabVIEW generates an associated notify event after most filter events if no event case discarded the event. The notify event has the same name as the filter event but without the question mark.

Data Flow for Notify and Filter Events

For notify events, LabVIEW sends a copy of the event—in parallel—to each Event structure configured to handle the event.

For filter events, LabVIEW sends the event sequentially to each Event structure configured for the event. The order in which LabVIEW sends the filter event to each Event structure depends on the order in which you [register](#) the events. Each Event structure must complete the filter event case before LabVIEW can notify the next Event structure. If an Event structure case changes any of the event data, LabVIEW passes the changed data to subsequent Event structures in the chain. If an Event structure in the chain discards the event, LabVIEW does not pass the event to any Event structures remaining in the chain. LabVIEW completes processing the user action that triggered the event only after all configured Event structures handle the event without discarding it.

Choosing How the Event Structure Monitors For Events

LabVIEW can generate many different events. Use event registration to specify which events you want an Event structure to handle. When you register for an event, LabVIEW notifies you when that particular event occurs. In LabVIEW, you can register for an event statically or dynamically. With static registration, LabVIEW always queues the event and notifies you when the event occurs. For example, if you statically register for a Mouse Move event on a front panel control, LabVIEW queues this event every time the user moves the mouse over that control. With dynamic registration, you can design applications so that LabVIEW only queues the event and notifies you when the event occurs if that event occurs under the circumstances you specify. For example, you may not want to monitor the Mouse Move event until the user presses a certain key. You also can use dynamic registration to change which VIs or controls generate events and to handle events in a subVI rather than the VI where LabVIEW generates events.

Static Event Registration

LabVIEW automatically performs static registration when you run a VI that contains an Event structure. You can configure an Event structure to specify which [user interface events](#) on the front panel of a VI you want to handle in each Event structure case. Use the Edit Events dialog box to configure an Event structure to [handle a statically registered event](#).



NOTE

You cannot statically configure an Event structure to handle events for the front panel of a different VI.

Refer to `examples\Structures\Event Structure\Event Structure.lvproj` for examples of using static event registration.

Dynamic Event Registration

[Dynamically registering for events](#) allows you to control when LabVIEW generates events, which events LabVIEW generates, and where LabVIEW handles those events. Dynamic event registration provides the following benefits:

- You can control when LabVIEW generates events by registering for events during different parts of an application. For example, you may want to monitor only certain events under certain circumstances, like mouse movement over a graph after the

user clicks a button that plots data to the graph. With static registration, the Mouse Move event always occurs. With dynamic event registration, you can register for the Mouse Move event after the user clicks the button to plot data to the graph.

- At run time, you can change which VIs or controls generate events. To change the objects for which LabVIEW generates events, *modify* the registration information at run time.
- LabVIEW can generate events without direct user interaction with the front panel. With dynamic event registration, you can design an application to make programmatic changes, like updating the value of a control from the block diagram, and LabVIEW generates events for these changes. Static event registration requires user interaction for LabVIEW to generate events.
- You can handle events in a subVI rather than only in the VI where you generate the events, which provides modularity to the application. For example, in a DAQ application, you can create a subVI to handle all of the user interface events, a subVI to handle all *custom events*, and a subVI to handle all DAQ events for the entire application rather than handling all events in the VI where the events are generated.

Grouping Data in LabVIEW

Grouping Data Using Strings, Clusters, Arrays, and Collections

Use the following data types to group data in LabVIEW:

- **String** —Groups sequences of ASCII characters.
- **Cluster** —Groups data elements of mixed types.
- **Array** —Groups data elements of the same type in unsorted order. Includes the matrix data type, a 2D array you can use to make modeling of math problems easier:
 - **Matrix** —Groups rows or columns of real or complex scalar data for some math operations, such as linear algebra operations.
- **Collection** —Includes the following sub-types:
 - **Set** —Groups unique data elements of the same type in sorted order.
 - **Map** —Groups key-value pairs where all the keys are unique. The key and value can each be any data type.

Related Concepts

[Grouping Data with Strings](#)

[Grouping Data with Arrays and Clusters](#)

[Grouping Data with Collections](#)

Using Matrices

Grouping Data with Strings

A string is a sequence of displayable or non-displayable ASCII characters. Strings provide a platform-independent format for information and data. Some of the more common applications of strings include the following:

- Creating simple text messages.
- Controlling instruments by sending text commands to the instrument and returning data values in the form of either ASCII or binary strings which you then convert to numeric values.
- Storing numeric data to disk. To store numeric data in an ASCII file, you must first convert numeric data to strings before writing the data to a disk file.
- Instructing or prompting the user with dialog boxes.

On the front panel, strings appear as tables, text entry boxes, and labels. LabVIEW includes built-in VIs and functions you can use to manipulate strings, including formatting strings, parsing strings, and other editing.

Refer to the ASCII Codes topic for more information about ASCII codes and conversion functions.

Strings on the Front Panel

Use the [string controls and indicators](#) as text entry boxes and labels.

String Display Types

Right-click a string control or indicator on the front panel to select from the display types shown in the following table. You also can right-click a string control or indicator and select **Visible Items»Display Style** from the shortcut menu to display a glyph on the object that indicates the display type. The table also shows an example message in each display type.

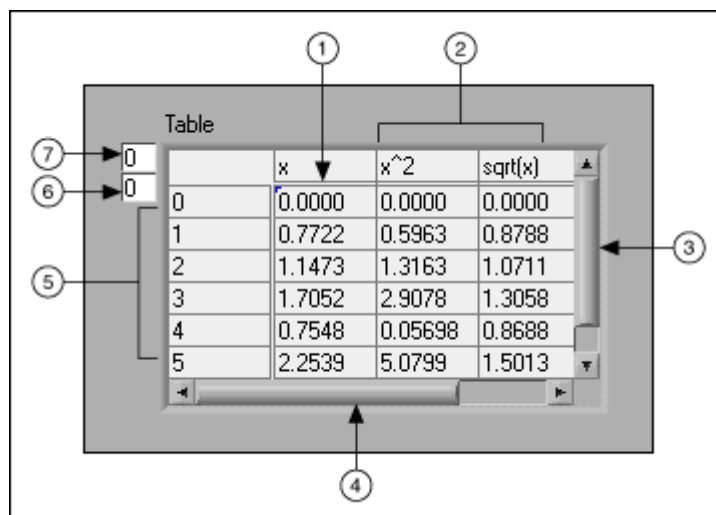
Type Glyph	Display Type	Description	Message
n	Normal Display	Displays printable characters using the font of the control. Non-displayable characters generally appear as boxes.	There are four display types. \ is a backs

Type Glyph	Display Type	Description	Message
\	<i>' Codes Display</i>	Displays backslash codes for all non-displayable characters.	There\sare\sfour\sdisplay\stypes.\n\\\sis\sas\s
P	<i>Password Display</i>	Displays an asterisk * for each character including spaces.	*****
*	<i>Hex Display</i>	Displays the ASCII value of each character in hex instead of the character itself.	5468 6572 6520 6172 6520 666F 7572 2064 6973 2074 7970 6573 2E0A 5C20 6973 2061 2062 6163 7368 2E

Tables

Use the table control to create a table on the front panel. Each cell in a table is a string, and each cell resides in a column and a row. Therefore, a table is a display for a 2D *array* of strings.

The following illustration shows a table and all its parts.



1	Cell indicated by index values
2	Column headings
3	Vertical scroll bar

4	Horizontal scroll bar
5	Row headings
6	Horizontal index
7	Vertical index

Editing, Formatting, and Parsing Strings

Use the String functions to edit strings in ways similar to the following:

- Search for, retrieve, and replace characters or substrings within a string.
- Change all text in a string to upper case or lower case.
- Find and retrieve matching patterns within a string.
- Retrieve a line from a string.
- Rotate and reverse text within a string.
- Concatenate two or more strings.
- Delete characters from a string.

Refer to the [LabVIEW Style Checklist](#) for more tips on minimizing memory usage when editing strings programmatically. Refer to the `labview\examples\Strings\Strings.lvproj` for examples of using the String functions to edit strings.

Formatting and Parsing Strings

To use data in another VI, function, or application, you often must convert the data to a string and then format the string in a way that the VI, function, or application can read. For example, Microsoft Excel expects strings that include delimiters, such as tabs, commas, or blank spaces. Excel uses the delimiter to segregate numbers or words into cells.

For example, to write a 1D array of numeric values to a spreadsheet using the Write to Text File function, you must format the array into a string and separate each numeric with a delimiter, such as a tab. You can use the Write Delimited Spreadsheet VI to format the array into a string and insert delimiters.

Use the String functions to perform tasks similar to the following:

- Concatenate two or more strings.
- Extract a subset of strings from a string.
- Convert data into strings.
- Format a string for use in a word processing or spreadsheet application.

Use the File I/O VIs and functions to save strings to text and spreadsheet files.

Format Specifiers

In many cases, you must enter one or more format specifiers in the **format string** parameter of a String function to format a string. A format specifier is a code that indicates how to convert numeric data to or from a string. LabVIEW uses conversion codes to determine the textual format of the parameter. For example, a format specifier of %x converts a hex integer to or from a string.

The Format Into String and Scan From String functions can use multiple format specifiers in the **format string** input, one for each input or output to the expandable function.

The Array To Spreadsheet String and Spreadsheet String To Array functions use only one format specifier in the **format string** input because these functions have only one input to convert. LabVIEW treats any extra specifiers you insert into these functions as literal strings with no special meaning.

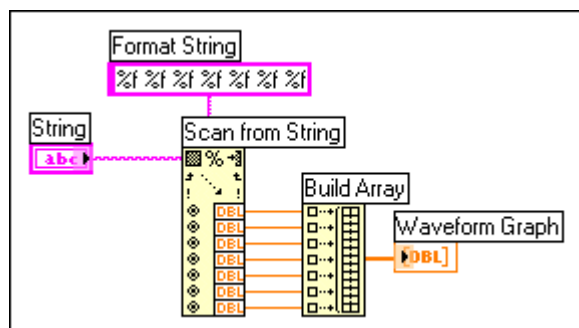
Numeric and String Data

Numeric data and string data differ because string data is ASCII characters and numeric data is not. Text and spreadsheet files accept strings only. To write numeric data to a text or spreadsheet file, you must first convert the numeric data to a string.

To add a set of numeric values to an existing string, convert the numeric data to a string and use the Concatenate Strings or another String function to add the new string to the existing string. Use the String/Number Conversion functions to convert numeric values to strings.

A string can include a set of numeric values you display in a graph or chart. For example, you can read a text file that includes a set of numeric values that you want to plot to a chart. However, those numeric values are in ASCII text, so you must read the numeric values as a string and then format the string into a set of numeric values before you plot the numeric values to a chart.

The following block diagram shows a string that includes a set of numeric values, *converts the string to numeric values*, builds an array of numeric values, and plots the numeric values to a graph.



Searching Strings

Use the String Functions to search strings for characters, words, phrases, or numbers with regular expressions. You can use regular expressions to search for literal strings in the text, such as the word "cat", or search for strings in the text that match a particular pattern, such as words that begin with the letter "c" and end with the letter "t" or substrings that start with a letter followed by one or more digits.

Use the Match Pattern function and the Search and Replace Pattern VI to search strings by entering literal strings or *special characters* in the **regular expression** input. Use the Search and Replace String function to *search for and replace text* in a string.

Use the Match Regular Expression function and the Search and Replace String function for more complex regular pattern matching using standard *Perl Compatible Regular Expression (PCRE) syntax*. Right-click the Search and Replace String function and select **Regular Expression** from the shortcut menu to configure the function for advanced regular expression searches and partial match substitution in the replacement string. *Resize* the Match Regular Expression function to view any partial matches found in the string. You can use the Match Pattern function or the Match Regular Expression function to search strings for regular expressions. The Match Regular Expression function gives you more options for matching strings but performs more slowly than the Match Pattern function.

Regular expression support is provided by the PCRE library package. Refer to the < National Instruments > _Legal Information directory for more information about the license under which the PCRE library package is redistributed.

Refer to the PCRE library package at www.pcre.org for more information about Perl Compatible Regular Expressions.

Grouping Data with Arrays and Clusters

Use the array and cluster controls and functions to group data. Arrays group data elements of the same type. Clusters group data elements of mixed types.

Arrays

An array consists of elements and dimensions. Elements are the data that make up the array. A dimension is the length, height, or depth of an array. An array can have one or more dimensions and as many as $(2^{31} - 1)$ elements per dimension, memory permitting.

You can build arrays of numeric, Boolean, path, string, waveform, and cluster data types. Consider using arrays when you work with a collection of similar data and when you perform repetitive computations. Arrays are ideal for storing data you collect from waveforms or data generated in loops, where each iteration of a loop produces one element of the array. You can *export data* from arrays to locations such as Microsoft Excel.

Right-click an array and select **Export** from the shortcut menu to view available export options.



NOTE

Array indexes in LabVIEW are zero-based. The index of the first element in the array, regardless of its dimension, is zero.

Restrictions

You cannot create arrays of arrays. However, you can use a multidimensional array or create an array of **clusters** where each cluster contains one or more arrays. Also, you cannot create an array of subpanel controls, tab controls, .NET controls, ActiveX controls, charts, multi-plot XY graphs, or multicolumn listboxes.

Indexes

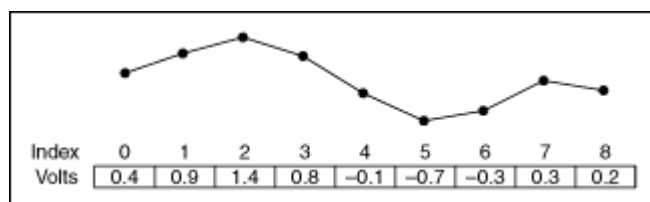
To locate a particular element in an array requires one index per dimension. In LabVIEW, indexes let you navigate through an array and retrieve elements, rows, columns, and pages from an array on the block diagram.

Examples of Arrays

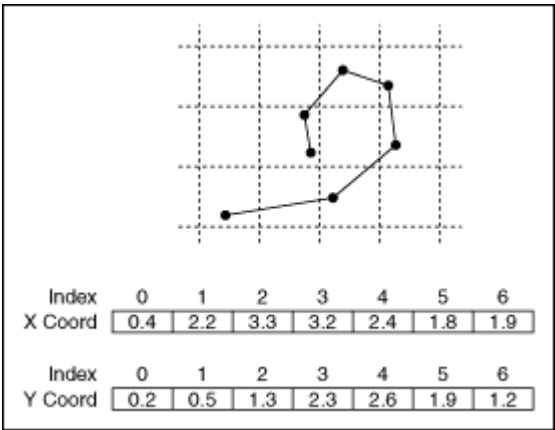
An example of a simple array is a text array that lists the eight planets of our solar system. LabVIEW represents this as a 1D array of strings with eight elements.

Array elements are ordered. An array uses an index so you can readily access any particular element. The index is zero-based, which means it is in the range 0 to $n - 1$, where n is the number of elements in the array. For example, $n = 8$ for the eight planets, so the index ranges from 0 to 7. Earth is the third planet, so it has an index of 2.

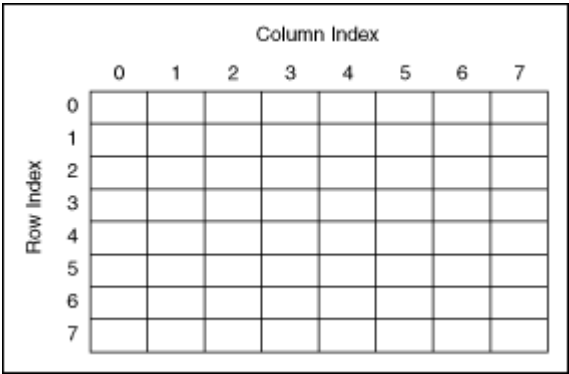
Another example of an array is a waveform represented as a numeric array in which each successive element is the voltage value at successive time intervals, as shown in the following illustration.



A more complex example of an array is a graph represented as an array of points where each point is a cluster containing a pair of numeric values that represent the X and Y coordinates, as shown in the following illustration.

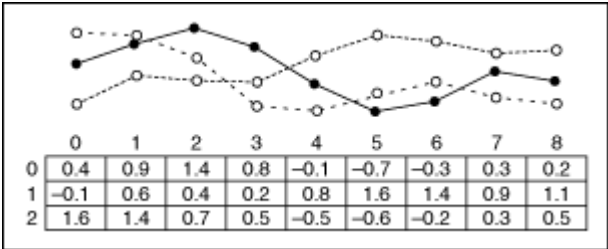


The previous examples use 1D arrays. A 2D array stores elements in a grid. It requires a column index and a row index to locate an element, both of which are zero-based. The following illustration shows an 8 column by 8 row 2D array, which contains $8 \times 8 = 64$ elements.



For example, a chessboard has eight columns and eight rows for a total of 64 positions. Each position can be empty or have one chess piece. You can represent a chessboard as a 2D array of strings. Each string is the name of the piece that occupies the corresponding location on the board or an empty string if the location is empty.

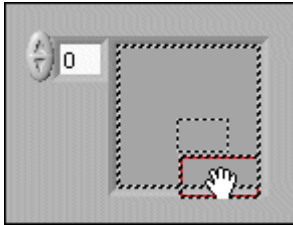
You can generalize the previous 1D array examples to two dimensions by adding a row to the array. The following illustration shows a collection of waveforms represented as a 2D array of numeric values. The row index selects the waveform, and the column index selects the point on the waveform.



Refer to the `labview\examples\Arrays\Arrays.lvproj` for examples of using arrays. Find related examples

Creating Array Controls, Indicators, and Constants

Create an array control or indicator on the front panel by adding an array shell to the front panel, as shown in the following front panel, and dragging a data object or element, which can be a numeric, Boolean, string, path, refnum, or cluster control or indicator, into the array shell.



The array shell automatically resizes to accommodate the new object.

To create an array constant on the block diagram, select an array constant on the Functions palette, place the array shell on the block diagram, and place a string constant, numeric constant, a Boolean constant, or cluster constant in the array shell. You can use an array constant to store constant data or as a basis for comparison with another array.

Creating Multidimensional Arrays

To add a multidimensional array to the front panel, right-click the index display and select **Add Dimension** from the shortcut menu. You also can resize the index display until you have as many dimensions as you want. To delete dimensions one at a time, right-click the index display and select **Remove Dimension** from the shortcut menu. You also can resize the index display to delete dimensions.

To display a particular element on the front panel, either type the index number in the index display or use the arrows on the index display to navigate to that number.

For example, a 2D array contains rows and columns. As shown in the following front panel, the upper display of the two boxes on the left is the row index and the lower display is the column index. The combined display to the right of the row and column displays shows the value at the specified position. The following front panel shows that the value at row 6, column 13, is 66.



1	Row index
2	Column index
3	Value at row, column

Rows and columns are zero-based, meaning the first column is column 0, the second column is column 1, and so on. Changing the index display for the following array to row 1, column 2 displays a value of 6.

0	1	2	3
4	5	6	7
8	9	10	11

If you try to display a column or row that is out of the range of the array dimensions, the array control appears dimmed to indicate that there is no value defined, and LabVIEW displays the default value of the data type. The default value of the data type depends on the data type of the array.

Use the Positioning tool to resize the array to show more than one row or column at a time.

You also can use the scroll bars of an array to navigate to a particular element. Right-click the array and select **Visible Items»Vertical Scrollbar** or **Visible Items»Horizontal Scrollbar** from the shortcut menu to display scroll bars for the array.



Note Array functions in LabVIEW access array data in row-major order. In a 2D array, row is the first, major index. Column is the last, minor index. In bigger multidimensional arrays, column remains the last index and the more major indexes are added to the front. Thus, vol is the first index, followed by page, row, and column. These names are index identifiers and have no other meaning.

Array Functions

Use the Array functions to create and manipulate arrays. For example, you can perform tasks similar to the following:

- Extract individual data elements from an array.
- Insert, delete, or replace data elements in an array.
- Split arrays.

Use the Build Array function to build an array programmatically. You also can [use a loop](#) to build an array.

Refer to the [LabVIEW Style Checklist](#) for more tips on minimizing memory usage when using Array functions in a loop.

Automatically Resizing Array Functions

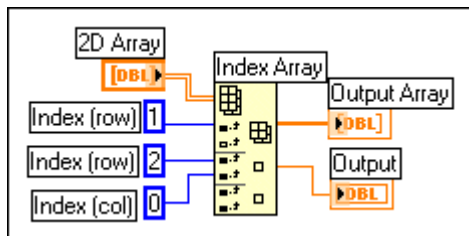
The Index Array, Replace Array Subset, Insert Into Array, Delete From Array, and Array Subset functions automatically resize to match the dimensions of the input array you wire. For example, if you wire a 1D array to one of these functions, the function shows a single

index input. If you wire a 2D array to the same function, it shows two index inputs—one for the row and one for the column.

You can access more than one element, or subarray (row, column, or page), with these functions by using the Positioning tool to manually resize the function. When you expand one of these functions, the functions expand in increments determined by the dimensions of the array wired to the function. If you wire a 1D array to one of these functions, the function expands by a single index input. If you wire a 2D array to the same function, the function expands by two index inputs—one for the row and one for the column.

The index inputs you wire determine the shape of the subarray you want to access or modify. For example, if the input to an Index Array function is a 2D array and you wire only the row input, you extract a complete 1D row of the array. If you wire only the column input, you extract a complete 1D column of the array. If you wire the row input and the column input, you extract a single element of the array. Each input group is independent and can access any portion of any dimension of the array.

The following block diagram uses the Index Array function to retrieve a row and an element from a 2D array.



To access multiple consecutive values in an array, expand the Index Array function, but do not wire values to the index inputs in each increment. For example, to retrieve the first, second, and third rows from a 2D array, expand the Index Array function by three increments and wire 1D array indicators to each subarray output.

Default Data in Arrays

Indexing beyond the bounds of an array produces the **default value** for the array element parameter. You can use the Array Size function to determine the size of the array.

You can index beyond the bounds of an array inadvertently by indexing an array past the last element **using a While Loop**, by supplying too large a value to the **index** input of an Index Array function, or by supplying an empty array to an Index Array function.

Clusters

Clusters group data elements of mixed types. An example of a cluster is the LabVIEW error cluster, which combines a Boolean value, a numeric value, and a string. A cluster is similar to a record or a struct in text-based programming languages.

Bundling several data elements into clusters eliminates wire clutter on the block diagram and reduces the number of connector pane terminals that subVIs need. The connector

pane has, at most, 28 terminals. If your front panel contains more than 28 controls and indicators that you want to pass to another VI, group some of them into a cluster and assign the cluster to a terminal on the connector pane.

Most clusters on the block diagram have a pink wire pattern and data type terminal. Error clusters have a dark yellow wire pattern and data type terminal. Clusters of numeric values, sometimes referred to as points, have a brown wire pattern and data type terminal. You can wire brown numeric clusters to Numeric functions, such as Add or Square Root, to perform the same operation simultaneously on all elements of the cluster.

Order of Cluster Elements

Although cluster and array elements are both ordered, you must unbundle all cluster elements at once using the Unbundle function. You can use the Unbundle By Name function to unbundle cluster elements by name. If you use the Unbundle by Name function, each cluster element must have a label. Clusters also differ from arrays in that they are a fixed size. Like an array, a cluster is either a control or an indicator. A cluster cannot contain a mixture of controls and indicators.

Cluster elements have a logical order unrelated to their position in the shell. The first object you place in the cluster is element 0, the second is element 1, and so on. If you delete an element, the order adjusts automatically. The cluster order determines the order in which the elements appear as terminals on the Bundle and Unbundle functions on the block diagram. You can view and [modify the cluster order](#) by right-clicking the cluster border and selecting **Reorder Controls In Cluster** from the shortcut menu.

To wire clusters to each other, both clusters must have the same number of elements. Corresponding elements, determined by the cluster order, must have compatible data types. For example, if a double-precision floating-point numeric value in one cluster corresponds in cluster order to a string in the another cluster, the wire on the block diagram appears broken and the VI does not run. If the numeric values are different representations, LabVIEW [coerces](#) them to the same representation.

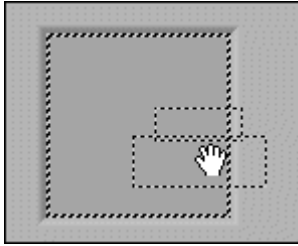
Cluster Functions

Use the Cluster, Class, & Variant VIs and functions to create and manipulate clusters. For example, you can perform tasks similar to the following:

- Extract individual data elements from a cluster.
- Add individual data elements to a cluster.
- Break a cluster out into its individual data elements.

Creating Cluster Controls, Indicators, and Constants

Create a cluster control or indicator on the front panel by adding a cluster shell to the front panel, as shown in the following front panel, and dragging a data object or element, which can be a numeric, Boolean, string, path, refnum, array, or cluster control or indicator, into the cluster shell.



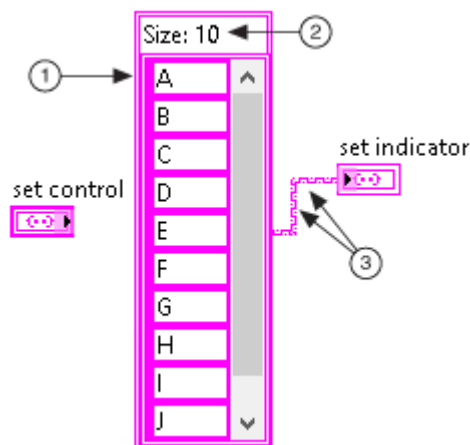
To create a cluster constant on the block diagram, select a cluster constant on the Functions palette, place the cluster shell on the block diagram, and place a string constant, numeric constant, a Boolean constant, or cluster constant in the cluster shell. You can use a cluster constant to store constant data or as a basis for comparison with another cluster.

Grouping Data with Collections

A collection is a container for aggregating homogeneous data. Each object in the collection is an element or entry of the collection. Use the collection data types to create high-performance data structures for storing and accessing data. LabVIEW includes the following collection data types: set and map.

Collection Sets

A set groups unique elements of the same data type in sorted order. Inserting a new element modifies the set only if that element does not already exist in the set.



①	Element
②	Size—Number of elements in the set
③	Wires of set—Wire color matches the data type color of the element

Because a set maintains elements in sorted order, a set allows faster search, insertion, modification, and removal operations on the data than unordered data structures, such as

arrays, even when the data size is large. A set has memory overhead to maintain the elements in sorted order, which arrays do not have.

Collection Maps

A map groups key-value pairs in sorted order where all the keys are unique. The key and value can each be any data type. Maps are also known as dictionaries because the key is used to look up a value. Inserting a key-value pair into a map either adds the pair to the map or replaces the existing value of the associated key with the new value in the map, depending on whether the key already exists in the map.

①	Key
②	Value
③	Entry—A pair of associated key and value
④	Size—Number of entries in the map
⑤	Wires of map—Wire color matches the data type color of the value

While variant attributes can also be used as a dictionary to look up a value, maps are more flexible than variant attributes in that the key and value can each be any LabVIEW data type. For variant attributes, the key must be a string and the value must be a variant. Maps are also less error prone than variant attributes. For maps, LabVIEW returns wiring errors at edit time because the value accepts a specific data type. For variant attributes, LabVIEW returns wiring errors at run time because the value is a variant type that accepts any data.

Use the Collection VIs and functions and the collection controls and indicators to create or manipulate collections.



NOTE

You cannot modify the embedded data in a collection interactively or programmatically. You can update collection data only as a whole through operations such as writing to the collection front panel terminal or right-clicking the collection and selecting **Data Operations»Copy Data** or **Data Operations»Paste Data**.

Refer to the following VIs or projects for an example of manipulating data using collections:

- labview\examples\Collections\Set Collection - Word Counting.vi
- labview\examples\Collections\Map Collection - Comparing Test Results.vi
- labview\examples\Collections\Map Collection - Word Counting.vi
- labview\examples\Design Patterns\Registration Map\Registration Map Usage.lvproj

Format Specifier Syntax

You use format specifiers to *format strings*, *convert a number into a string*, and *insert non-displayable characters in strings*.

For functions that produce a string as an output, such as Format Into String and Array To Spreadsheet String, a format specifier uses the following syntax elements. Double brackets [] enclose optional elements.

```
%[$][-][+][#][^][0][Width][.Precision|_|_SignificantDigits]
[ { Unit } ] [< Embedded information >] Conversion Code
```

where **Width** must be a number greater than zero and **.Precision** and **_SignificantDigits** must be a number greater than or equal to zero.

For functions that scan a string, such as Scan From String and Spreadsheet String To Array, a format specifier uses the following simplified syntax elements.

```
%[Width]Conversion Code
```

The Format Into String, Format Into File, Scan From String, and Scan From File functions can use multiple format specifiers in the **format string** input, one for each input or output of the expandable function.

Format Specifiers Syntax Elements

The following table displays the syntax elements for format specifiers. Refer to the [examples of format specifiers](#) for more information.

Table 1 :

Syntax Element	Description
%	Begins the format specifier.
\$ (optional)	When you use a formatting function, this modifier specifies the order in which to display variables. Include the digit that represents the order of the variable immediately before this modifier.
- (optional)	When you use a formatting function, this modifier left justifies the parameter rather than right justifies it within its width.
+ (optional)	When you use a formatting function, this modifier includes sign even when the number is positive.
^ (optional)	When you use a formatting function and the e or g conversion codes, this element formats the number in engineering notation, where the exponent is always a multiple of three.
# (optional)	When you use a formatting function, this modifier removes trailing zeros. If the number has no fractional part, this modifier also removes the description part.
0 (optional)	When you use a formatting function, use this modifier without the - modifier to pad any excess space to the left of a numeric parameter with zeros rather than with spaces to reach minimum width.
Width (optional)	When you use a scanning function, such as Scan From String, the Width element specifies the maximum character field width to use. LabVIEW scans the maximum number of characters when processing the parameter. When you use a formatting function, the Width element specifies the minimum character field width of the output. This width is not a maximum width. LabVIEW uses as many characters as necessary to format the parameter without truncating it. LabVIEW pads the field to the left or right of the parameter with spaces, depending on justification. If Width is missing or 0, the output is only as long as necessary to contain the converted input parameter.

Table 1 : (Continued)

Syntax Element	Description
.Precision or _Significant Digits (optional)	When you use a formatting function, . or _ controls the number of digits displayed. If you use ., LabVIEW uses the number that follows as a precision specifier for digits to the right of the decimal point. If you use _, LabVIEW uses the number that follows as the specified number of significant digits to use in the display format. .Precision—When you use it with floating-point notation, this element specifies the number of digits to the right of the decimal point. If . is not present, LabVIEW uses a precision of six digits. If . is 0, LabVIEW does not insert a precision. When you use it with string parameters, .Precision specifies the maximum width of the scanned field. LabVIEW truncates strings longer than this length. _Significant Digits—Displays the data by rounding to the number of digits you specify. LabVIEW rounds the data only for display purposes, which does not affect the original data. .Precision affects only the digits to the right of the decimal point, and _Significant Digits includes all non-spacing digits. For example, • 3.457 has 4 significant digits • 0.0012 has 2 significant digits • 123000 has 3 significant digits  NOTE You cannot use precision and significant digits together in a single format specifier. For single-precision, floating-point numbers, National Instruments recommends that you use values from 1 to 6 for _Significant Digits. For double-precision and extended-precision, floating-point numbers, National Instruments recommends that you use values from 1 through 13 for _Significant Digits.
{Unit} (optional)	Overrides the original unit of a VI when you use a function to convert a physical quantity (a value with an associated unit). You must use a compatible unit. You can use this syntax element only with the Format Into String function.
<Embedded Time Format> (optional)	Contains a time-specific format string for use with the T (absolute time) and t (relative time) conversion codes. Refer to <i>Format Codes for the Time Format String</i> for valid format strings. Only %W, %D, %H, %M, %S and %u apply to relative time.

Table 1 : (Continued)

Syntax Element	Description
Conversion Codes	Characters that specify how to scan or format a parameter. Use the following conversion codes for integers and fixed-point numbers: • x—Hexadecimal integer (for example, B8). • o—Octal integer (for example, 701). • b—Binary integer (for example, 1011). • d—Signed decimal integer. • u—Unsigned decimal integer.
	Use the following conversion codes for floating-point and fixed-point numbers: • f—Floating-point number with fractional format (for example, 12.345). • e—Floating-point number in scientific notation (for example, 1.234E1). • g—LabVIEW uses f or e depending on the exponent of the number. LabVIEW uses f if the exponent is greater than -4 or less than the precision specified. LabVIEW uses e if the exponent is less than -4 or greater than the precision specified. • p—Floating-point number in SI notation.
	Use the following conversion codes for a string: • s—String (for example, abc). When scanning, s matches only up to the next white-space character. A space matches one or more consecutive white-space characters. To scan a string that may contain white space, use the characters in set conversion code. Specify all the characters which the string may contain within the brackets, including space or other white-space characters. • []—Characters in set. [] matches a string that contains only the characters specified between the brackets. Character matches are case sensitive. The [] conversion code is useful only when you are scanning strings. To match the caret symbol (^) in a set, make sure it is not the first character after the bracket. The following examples demonstrate the use of the characters in set conversion code: ◦ %[aeiou]—Matches any string that contains only lowercase vowels. ◦ %[0-9a-zA-Z]—Matches a string that contains numbers, letters, or spaces. You can use a hyphen to specify ranges of characters in the set. ◦ %[^ , ;]—Matches any string of characters up to but not including the first comma or semicolon.

Table 1 : (Continued)

Syntax Element	Description
Localization Codes	Characters that determine if LabVIEW uses a decimal or comma to separate the whole number from the decimal part of the number. These codes control the decimal separator for numeric output. These codes do not cause any input or output to occur. They change the decimal separator for all further inputs and outputs until they find the next %;. • % , ; —Comma decimal separator. • % . ; —Period decimal separator. • % ; —System default separator. If you do not specify a separator, LabVIEW uses the system default separator.
Backslash (\) Codes	Characters that specify hex values, spacing, backspaces, and other formatting options.

Format Specifier Examples

The following table displays examples using the *syntax elements* for format specifiers.

Type	Argument(s)	Format String	Resulting String	Comments
Automatic Formatting (%g)	12.00	%#g	12	If you specify #, LabVIEW removes trailing zeros. If you specify g, LabVIEW chooses scientific notation or floating-point notation based on the number to format.
	12000000	%#g	1.2E+7	

Type	Argument(s)	Format String	Resulting String	Comments
Decimal (%d)	12.67	score= %d%%	score= 13%	When you specify %d, LabVIEW displays integers in base-10 decimal format. - For signed or unsigned integers of any bit size, LabVIEW displays the integer unmodified. - LabVIEW rounds floating-point values to signed 64-bit integers. NaN and values greater than the maximum value of a signed 64-bit integer are rounded to the maximum value of a signed 64-bit integer, which is 9223372036854775807. To record values greater than the maximum value of a signed 64-bit integer, you must explicitly cast the values to unsigned 64-bit integers using the To Unsigned Quad Integer function.
Floating-Point (%f)	12.67	Temp: %5.1f	Temp: 12.7	The 5 in the Format String section specifies a width of 5, and the 1 specifies the number of digits to the right of the decimal, or precision.
	12.67 N	%5.3f	12.670 N	Units are valid only if you use the Format Into String function. These are examples of physical quantity input. The second example shows how you can convert from one unit to another. The question mark indicates when the unit in the format specifier is in conflict with the input unit.
	12.67 N	%5.3{mN} f	12670.000 mN	
	12.67 N	%5.3{kg} f	12.670 ?kg	
Scientific Notation (%e)	12.67	%.3e	1.267E+1	Add ^ to change to engineering notation where

Type	Argument(s)	Format String	Resulting String	Comments
	12.67	%^ .3e	12.670E+0	the exponent is always a multiple of three.
SI Notation (%p)	12000000	%.2p	12.00M	A value of .2 specifies that you want a precision of 2 digits in the resulting string. A value of _2 specifies that you want two significant digits in the resulting string. %p adds the <i>SI prefixes</i> .
	12000000	%_2p	12M	
Hexadecimal (%x)	12	%02x	0C	A – indicates that you want the resulting string left justified. A 0 indicates that you want the resulting string padded with zeros. The number specifies the width of the resulting string. In other words, a value of –n pads the resulting string with spaces to the n width. b is base 2, o is base 8, and x is base 16.
Octal (%o)	12	%06o	000014	
Binary (%b)	12	%b	1100	
Relative Time (%t)	91.80	%.2t	01:31.80	LabVIEW formats elapsed time in terms of complete weeks %W, days %D, hours %H, minutes %M, seconds %S, and fractions of seconds %u. Refer to the Format Codes for the Time Format String for more time format codes.
	91.8	%<Hours:%H Minutes:%M Seconds:%S>t	Hours:00 Minutes:01 Seconds:31	
Absolute Time (%T)	00:00:00.000 AM 1/1/2001 (Universal Time)	%<%.3X %x>T	12:00:00.000 AM 01/01/2001	%T specifies absolute time. Any information you include with the < and > brackets indicates how you want to format the absolute time. This format, including the punctuation, changes based on the regional settings of the computer. The time changes based on the configured time zone for the computer. Refer to the Format Codes for the Time Format String for more time format codes.
	00:00:00.000 AM 1/1/2001 (Universal Time)	%<%Y.%m.%d>T	2001.01.01	
	00:00:00.000 AM 1/1/2001 (Universal Time)	%^<%.3X %x>T	06:00:00.000 AM 12/31/2000	Add ^ to format the time in Universal Time.
String (%s)	Smith John	Name: %s, %s.	Name: Smith, John.	Strings are valid only if you use the Format Into String function. In the second

Type	Argument(s)	Format String	Resulting String	Comments
	Hello, World	String: %10.6s	String: Hello,	example, the format string specifies to use at most six characters from the string Hello, World then pad with spaces so the total string length is equal to 10.
Variable Order (%)\$)	Smith John	%2\$s %1\$s	John Smith	In this example, Smith is the first variable and John is the second variable. If you use the \$ syntax element in the format string for a particular variable, subsequent syntax elements that are not preceded by a \$ syntax element apply to the next variable. If you use the \$ syntax element for a variable, use the \$ syntax element for any other variables in the string to improve readability.
Percent Character (%)%)	None	%%	%	%% formats a single %. This format provides a way to include an actual percent character in the string. It does not add a terminal to the Format Into String or Scan From String functions.



RELATED INFORMATION
Format Codes for the Time Format String

Graphs and Charts in LabVIEW

Graphs and Charts

After you acquire or generate data, use a graph or chart to display data in a graphical form.

Graphs and charts differ in the way they display and update data. VIs with a graph usually collect the data in an array and then plot the data to the graph. This process is similar to a spreadsheet that first stores the data then generates a plot of it. When the data is plotted, the graph discards the previously plotted data and displays only the new data. You typically use a graph with fast processes that acquire data continuously.

In contrast, a chart appends new data points to those points already in the display to create a history. On a chart, you can see the current reading or measurement in context with data previously acquired. When more data points are added than can be displayed on

the chart, the chart scrolls so that new points are added to the right side of the chart while old points disappear to the left. You typically use a chart with slow processes in which only a few data points per second are added to the plot.



RELATED INFORMATION

[Types of Graphs and Charts](#) on page 477

[Types of Graphs and Charts](#) on page 477

[Customizing Graphs and Charts](#) on page 493

[Configuring a Graph or Chart](#) on page 1196

[Configuring the X- and Y-Axes on a Graph or Chart](#) on page 1196

[Exporting Data and Images](#) on page 1182

Types of Graphs and Charts

LabVIEW includes the following types of graphs and charts:

- **Waveform Graphs and Charts** —Display data typically acquired at a constant rate.
- **XY Graphs** —Display data acquired at a non-constant rate and data for multivalued functions.
- **Intensity Graphs and Charts** —Display 3D data on a 2D plot by using color to display the values of the third dimension.
- **Digital Waveform Graphs** —Display data as pulses or groups of digital lines.
- **Mixed Signal Graphs** —Display data types accepted by waveform graphs, XY graphs, and digital waveform graphs. Also accept clusters that contain any combination of those data types.
- **2D Graphs** —Display 2D data on a 2D front panel plot.
- **3D Graphs** —Display 3D data on a 3D front panel plot.



NOTE

3D graph controls are available only in the LabVIEW Full and Professional Development Systems.

- **ActiveX 3D Graphs** —Display 3D data on a 3D plot in an [ActiveX object](#) on the front panel.



Note ActiveX 3D graph controls are supported only on Windows in the LabVIEW Full and Professional Development Systems.

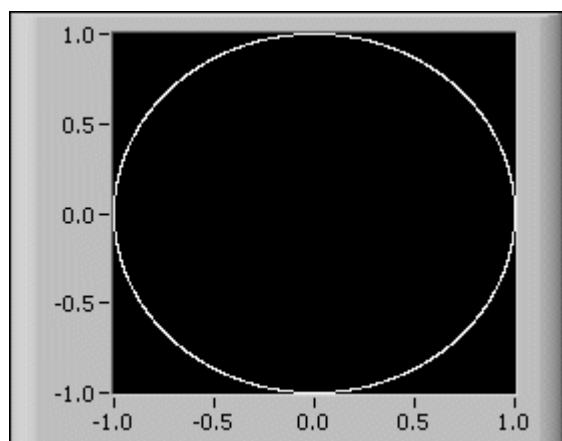
Refer to the `labview\examples\Controls and Indicators\Graphs and Charts` directory for examples of graphs and charts.

XY Graphs

The XY graph is a general-purpose, Cartesian graphing object that plots multivalued functions, such as circular shapes or waveforms with a varying time base. The XY graph displays any set of points, evenly sampled or not.

You also can **display** Nyquist planes, Nichols planes, S planes, and Z planes on the XY graph. Lines and labels on these planes are the same color as the Cartesian lines, and you cannot modify the plane label font.

The following front panel shows an example of an XY graph.



The XY graph can display plots containing any number of points. The XY graph also accepts several data types, which minimizes the extent to which you must manipulate data before you display it.

Displaying a Single Plot on XY Graphs

The XY graph accepts three data types for single-plot XY graphs. The XY graph accepts a cluster that contains an x array and a y array.

The XY graph also accepts an array of points, where a point is a cluster that contains an x value and a y value. The XY graph also accepts an array of complex data, in which the real part is plotted on the x-axis and the imaginary part is plotted on the y-axis.

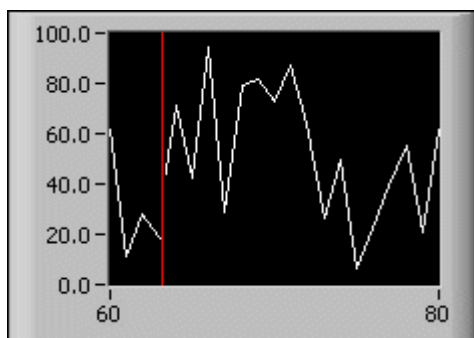
Displaying Multiple Plots on XY Graphs

The XY graph accepts three data types for displaying multiple plots. The XY graph accepts an array of plots, where a plot is a cluster that contains an x array and a y array.

The XY graph also accepts an array of clusters of plots, where a plot is an array of points. A point is a cluster that contains an x value and a y value. The XY graph also accepts an array of clusters of plots, where a plot is an array of complex data, in which the real part is plotted on the x-axis and the imaginary part is plotted on the y-axis.

Waveform Charts

The waveform chart is a special type of numeric indicator that displays one or more plots of data typically acquired at a constant rate. The following front panel shows an example of a waveform chart.



The waveform chart maintains a history of data, or buffer, from previous updates. Right-click the chart and select **Chart History Length** from the shortcut menu to configure the buffer. The default [chart history length](#) for a waveform chart is 1,024 data points. The frequency at which you send data to the chart determines how often the chart redraws.

Displaying a Single Plot on Waveform Charts

If you pass the chart a single value or multiple values at a time, LabVIEW interprets the data as points on the chart and increments the x index by one starting at $x = 0$. The chart treats these inputs as new data for a single plot.

The waveform chart accepts the [waveform data type](#), which carries the data, start time, and delta t of a waveform. Use the Build Waveform (Analog Waveform) function to plot time on the x-axis of the chart and automatically use the correct interval between markers on the x-scale of the chart. A waveform that specifies **t0** and a single-element Y array is useful for plotting data that is not evenly sampled because each data point has its own time stamp.

Refer to the `labview\examples\Controls and Indicators\Graphs and Charts\Waveform Graphs and Charts\Waveform Graphs and Charts.lvproj` for examples of the waveform chart.

Displaying Multiple Plots on Waveform Charts

To pass data for multiple plots to a waveform chart, you can bundle the data together into a cluster of scalar numeric values, where each numeric represents a single point for each of the plots.

If you want to pass multiple points per plot in a single update, wire an array of clusters of numeric values to the chart. Each numeric represents a single y value point for each of the plots.

You can use the [waveform data type](#) to create multiple plots on a waveform chart. Use the Build Waveform function to plot time on the x-axis of the chart and automatically use the correct interval between markers on the x-scale of the chart. A 1D array of waveforms that each specify **t0** and a single-element Y array is useful for plotting data that is not evenly sampled because each data point has its own time stamp.

If you cannot determine the number of plots you want to display until run time, or you want to pass multiple points for multiple plots in a single update, wire a 2D array of numeric values or waveforms to the chart. By default, the waveform chart treats each column in the array as a single plot. Wire a 2D array data type to the chart, right-click the chart, and select **Transpose Array** from the shortcut menu to treat each row in the array as a single plot.

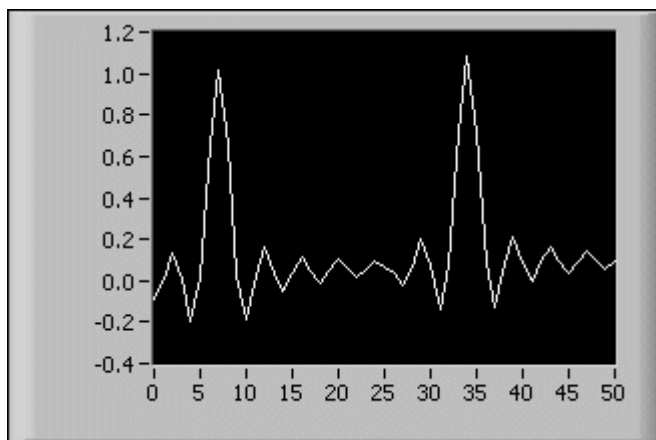
Refer to the `labview\examples\Controls and Indicators\Graphs and Charts\Waveform Graphs and Charts\Waveform Graphs and Charts.lvproj` for examples of the waveform chart.

Waveform Data Type

The [waveform data type](#) carries the data, start time, and delta t of a waveform. You can create a waveform using the Build Waveform function. Many of the VIs and functions you use to acquire or analyze waveforms accept and return waveform data by default. When you wire waveform data to a [waveform graph](#) or [chart](#), the graph or chart automatically plots a waveform based on the data, start time, and delta x of the waveform. When you wire an array of waveform data to a waveform graph or chart, the graph or chart automatically plots all waveforms.

Waveform Graphs

The waveform graph displays [one](#) or [more plots](#) of evenly sampled measurements. The waveform graph plots only single-valued functions, as in $y = fx$, with points evenly distributed along the x-axis, such as acquired time-varying waveforms. The following front panel shows an example of a waveform graph.



The waveform graph can display plots containing any number of points. The graph also accepts several data types, which minimizes the extent to which you must manipulate data before you display it.



Note Use the *digital waveform graph* to display digital data.

Displaying a Single Plot on Waveform Graphs

The waveform graph accepts several data types for single-plot waveform graphs. The graph accepts a single array of values, interprets the data as points on the graph, and increments the x index by one starting at $x = 0$. The graph accepts a cluster of an initial x value, a delta x, and an array of y data. The graph also accepts the *waveform data type*, which carries the data, start time, and delta t of a waveform.

The waveform graph also accepts the *dynamic data type*, which is for use with Express VIs. In addition to the data associated with a signal, the dynamic data type includes attributes that provide information about the signal, such as the name of the signal or the date and time the data was acquired. Attributes specify how the signal appears on the waveform graph. When the dynamic data type includes a single numeric value, the graph plots the single value and automatically formats the plot legend and x-scale time stamp. When the dynamic data type includes a single channel, the graph plots the whole waveform and automatically formats the plot legend and x-scale time stamp.

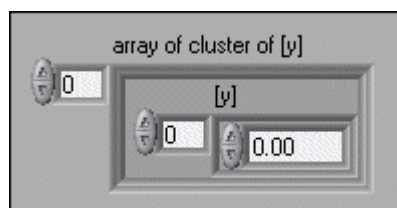
Refer to the Waveform Graph Data Type VI in the `labview\examples\Controls and Indicators\Graphs and Charts\Waveform Graphs and Charts` directory for examples of the data types that a waveform graph accepts.

Displaying Multiple Plots on Waveform Graphs

The waveform graph accepts several data types for displaying multiple plots. The waveform graph accepts a 2D array of values, where each row of the array is a single plot. The graph interprets the data as points on the graph and increments the x index by one, starting at $x = 0$. Wire a 2D array data type to the graph, right-click the graph, and select **Transpose Array** from the shortcut menu to handle each column of the array as a plot. This is particularly useful when you sample multiple channels from a DAQ device because the device can return the data as 2D arrays with each channel stored as a separate column.

The waveform graph also accepts a cluster of an initial x value, a delta x value, and a 2D array of y data. The graph interprets the y data as points on the graph and increments the x index by delta x, starting at the initial x value. This data type is useful for displaying multiple signals that are sampled at the same regular rate.

The waveform graph accepts a plot array where the array contains clusters. Each cluster contains a 1D array that contains the y data. The inner array describes the points in a plot, and the outer array has one cluster for each plot. The following front panel shows this array of the **y** cluster.



Use a plot array instead of a 2D array if the number of elements in each plot is different. For example, when you sample data from several channels using different time amounts from each channel, use this data structure instead of a 2D array because each row of a 2D array must have the same number of elements. The number of elements in the interior arrays of an array of clusters can vary.

The waveform graph accepts a cluster of an initial x value, a delta x value, and an array that contains clusters. Each cluster contains a 1D array that contains the y data. You use the Bundle function to bundle the arrays into clusters and you use the Build Array function to build the resulting clusters into an array. You also can use the Build Cluster Array function, which creates arrays of clusters that contain the inputs you specify.

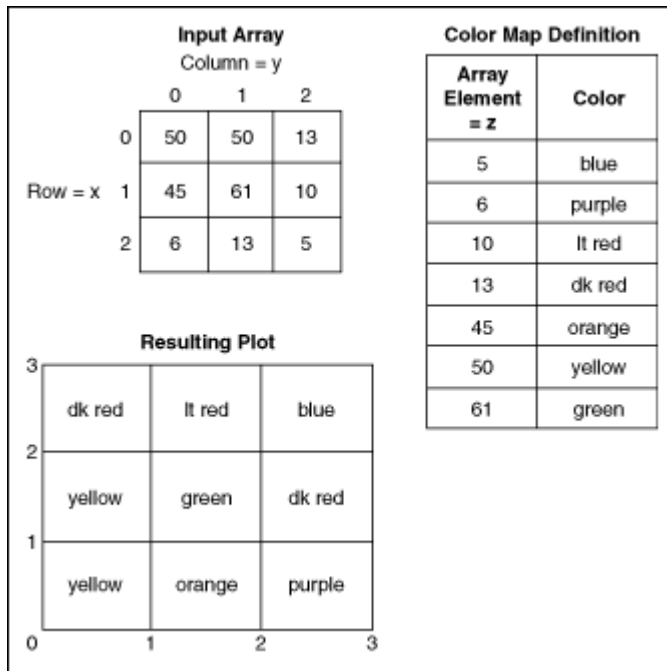
The waveform graph accepts an array of clusters of an x value, a delta x value, and an array of y data. This is the most general of the multiple-plot waveform graph data types because you can indicate a unique starting point and increment for the x-scale of each plot.

The waveform graph also accepts the *dynamic data type*, which is for use with Express VIs. In addition to the data associated with a signal, the dynamic data type includes attributes that provide information about the signal, such as the name of the signal or the date and time the data was acquired. Attributes specify how the signal appears on the waveform graph. When the dynamic data type includes multiple channels, the graph displays a plot for each channel and automatically formats the plot legend and x-scale time stamp.

Intensity Charts

Intensity Graphs and Charts

Use the intensity graph and chart to display 3D data on a 2D plot by placing blocks of color on a Cartesian plane. For example, you can use an intensity graph or chart to display patterned data, such as temperature patterns and terrain, where the magnitude represents altitude. The intensity graph and chart accept a 3D array of numbers. Each number in the array represents a specific color. The indexes of the elements in the 2D array set the plot locations for the colors. The following illustration shows the concept of the intensity chart operation.



The rows of the data pass into the display as new columns on the graph or chart. If you want rows to appear as rows on the display, wire a 2D array data type to the graph or chart, right-click the graph or chart, and select **Transpose Array** from the shortcut menu.

The array indexes correspond to the lower left vertex of the block of color. The block of color has a unit area, which is the area between the two points, as defined by the array indexes. The intensity graph or chart can display up to 256 discrete colors.

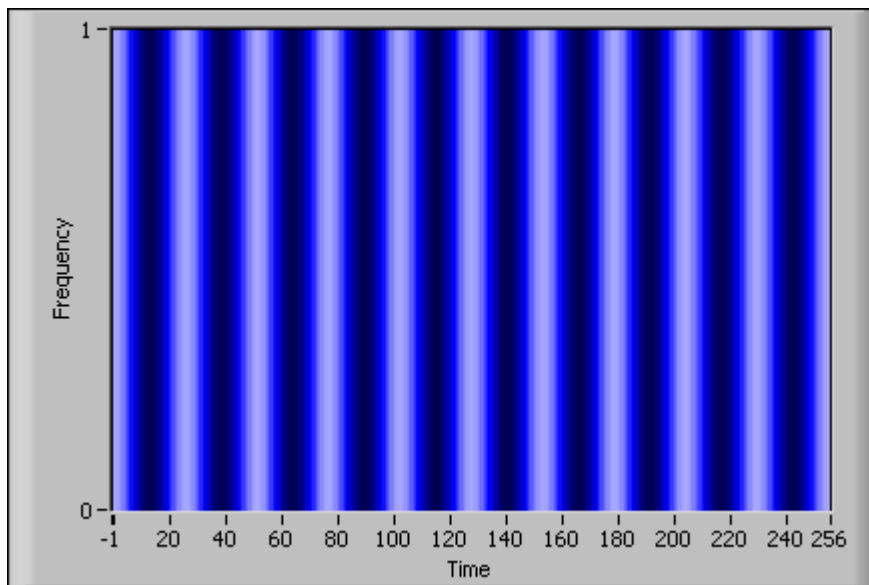
Refer to the `labview\examples\Controls and Indicators\Graphs and Charts\Intensity Graphs and Charts\Intensity Graphs and Charts.lvproj` for examples of intensity graphs and charts.

Find related examples

Intensity Charts

After you plot a block of data on an intensity chart, the origin of the Cartesian plane shifts to the right of the last data block. When the chart processes new data, the new data values appear to the right of the old data values. When a chart display is full, the oldest data values scroll off the left side of the chart. This behavior is similar to the behavior of a [strip chart](#).

The following front panel shows an example of an intensity chart.



The intensity chart shares many of the optional parts of the [waveform chart](#), including the [scale legend](#) and [graph palette](#), which you can show or hide by right-clicking the chart and selecting **Visible Items** from the shortcut menu. In addition, because the intensity chart includes color as a third dimension, a scale similar to a color ramp control defines the range and [mappings of values to colors](#).

Like the waveform chart, the intensity chart maintains a history of data, or buffer, from previous updates. Right-click the chart and select **Chart History Length** from the shortcut menu to configure the buffer. The default size for an intensity chart is 128 data points. The intensity chart display can be memory intensive.



Tip Unlike graphs, charts keep the history of data previously written. When a chart runs continuously, its history grows and requires additional memory space. This continues until the chart history is full, then LabVIEW stops taking more memory. LabVIEW does not automatically clear the chart history when the VI restarts. You can clear the chart history throughout the execution of the program. To do this, write empty arrays to the History Data attribute node for the chart.

Intensity Graphs

The intensity graph works the same as the [intensity chart](#), except it does not retain previous data values and does not include [update modes](#). Each time new data values pass to an intensity graph, the new data values replace old data values. Like other graphs, the intensity graph can have [cursors](#). Each cursor displays the x, y, and z values for a specified point on the graph.

Using Color Mapping with Intensity Graphs and Charts

An [intensity graph](#) or [chart](#) uses color to display 3D data on a 2D plot. When you set the color mapping for an intensity graph or chart, you configure the color scale of the graph or chart. The color scale consists of at least two arbitrary markers, each with a numeric value and a corresponding display color. The colors displayed on an intensity graph or chart

correspond to the numeric values associated with the specified colors. Color mapping is useful for visually indicating data ranges, such as when plot data exceeds a threshold value.

You can [set the color mapping interactively](#) for the intensity graph and chart the same way you define the colors for a [color ramp numeric control](#).

You can set the color mapping for the intensity graph and chart programmatically by using the Property Node in two ways. Typically, you specify the value-to-color mappings in the Property Node. For this method, specify the Z Scale:Marker Values property for the z-scale. This property consists of an array of clusters, in which each cluster contains a numeric limit value and the corresponding color to display for that value. When you specify the color mapping in this manner, you can specify an upper out-of-range color using the Z Scale:High Color property for the z-scale and a lower out-of-range color using the Z Scale:Low Color property for the z-scale. The intensity graph and chart are limited to a total of 254 colors, with the lower and upper out-of-range colors bringing the total to 256 colors. If you specify more than 254 colors, the intensity graph or chart creates the 254-color table by interpolating among the specified colors.

If you display a bitmap on the intensity graph, you specify a color table using the Color Table property. With this method, you can specify an array of up to 256 colors. Data passed to the chart are mapped to indexes in this color table based on the color scale of the intensity chart. If the color scale ranges from 0 to 100, a value of 0 in the data is mapped to index 1, and a value of 100 is mapped to index 254, with interior values interpolated between 1 and 254. Anything below 0 is mapped to the out-of-range below color (index 0), and anything above 100 is mapped to the out-of-range above color (index 255).



Note The colors you want the intensity graph or chart to display are limited to the exact colors and number of colors your video card can display. You also are limited by the number of colors allocated for your display.

Refer to the Modifying the Color Table VI in the labview\examples\Controls and Indicators\Graphs and Charts\Intensity Graphs and Charts for an example of color mapping.

Digital Waveform Graphs

Use the [digital waveform](#) graph to display digital data, especially when you work with timing diagrams or logic analyzers.

The digital waveform graph accepts the [digital waveform data type](#), the [digital data type](#), and an array of those data types as an input. By default, the digital waveform graph displays data as digital lines and buses in the plot area. [Customize the digital waveform graph](#) to display digital buses, digital lines, or a combination of digital buses and lines. If you wire an array of digital data where each array element represents a bus, the digital waveform graph plots each element of the array as a different line in the order that the array elements draw to the graph.

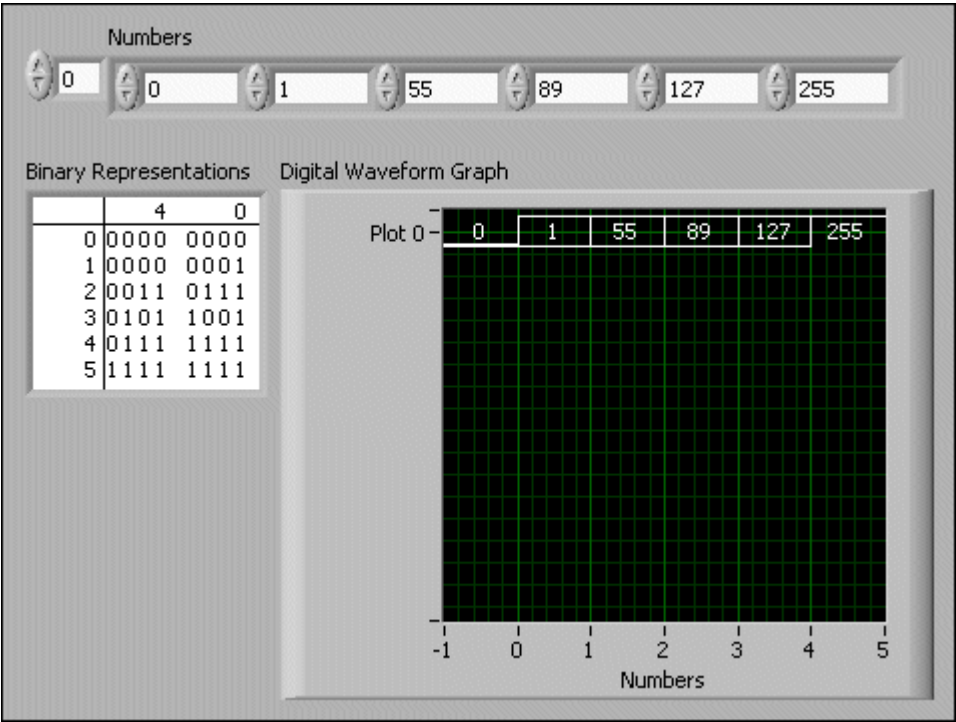
To expand and contract digital buses in the tree view of the plot legend, click the expand/contract symbol to the left of the digital bus. Expanding and contracting digital buses in the

tree view of the plot legend also expands and contracts the bus in the plot area of the graph. To expand and contract digital buses when the plot legend is in standard view, right-click the digital waveform graph and select **Y Scale»Expand Digital Buses** from the shortcut menu.



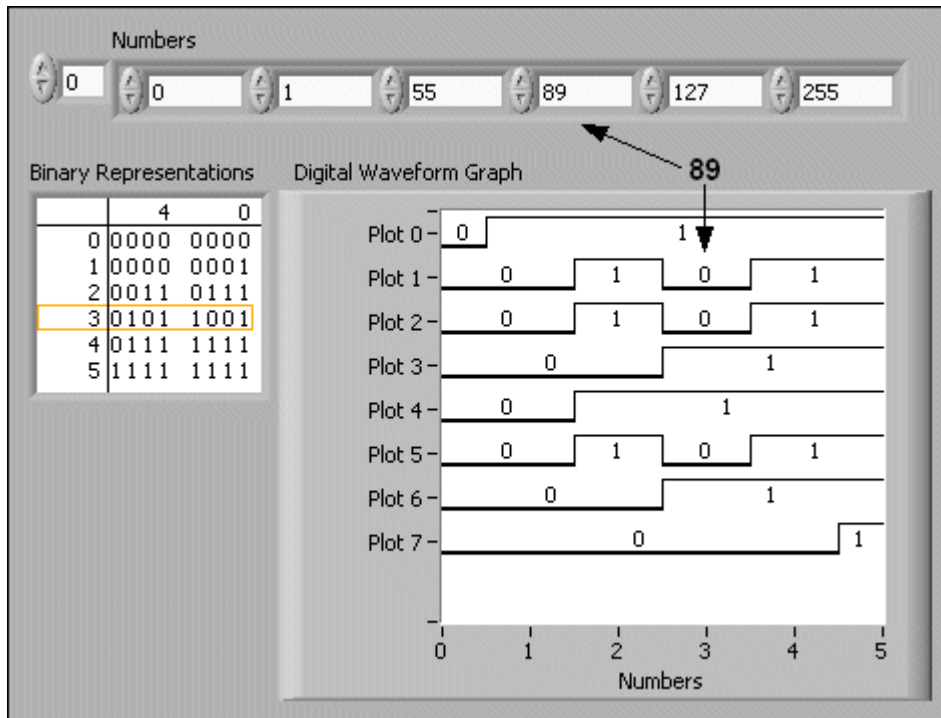
Note **Y Scale»Expand Digital Buses** is available only if you disable **Show Buses With Lines** and the plot legend is in standard view. To disable **Show Buses With Lines**, change the plot legend to the standard view, right-click the digital waveform graph, and select **Show Buses With Lines** from the shortcut menu to remove the checkmark next to the menu item.

The digital waveform graph in the following front panel plots digital data as a bus. The VI converts the numbers in the **Numbers** array to digital data and displays the binary representations of the numbers in the **Binary Representations** digital data indicator. In the digital graph, the number 0 appears without a top line to symbolize that all the bit values are zero. The number 255 appears without a bottom line to symbolize that all the bit values are 1.



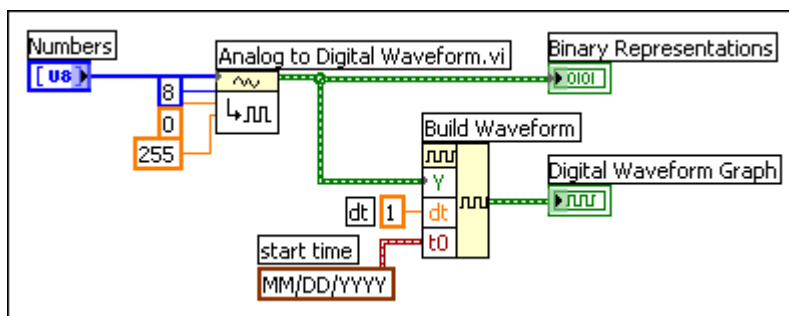
Right-click the y-scale and select **Expand Digital Buses** from the shortcut menu to plot each sample of digital data. Each plot represents a different bit in the digital pattern. You can [customize the appearance](#) of data plotted on a digital waveform graph.

The digital waveform graph in the following front panel displays the six numbers in the **Numbers** array.



The **Binary Representations** digital data indicator displays the binary representations of the numbers. Each column in the table represents a bit. For example, the number 89 requires 7 bits of memory (the 0 in column 7 indicates an unused bit). Point 3 on the digital waveform graph plots the 7 bits necessary to represent the number 89 and a value of 0 to represent the unused eighth bit on plot 7. Notice that the data is read from right to left.

The following VI converts an array of numbers to digital data and uses the Build Waveform function to assemble the start time, delta t, and the numbers entered in a [digital data control](#) and to display the digital data.



Refer to the labview\examples\Controls and Indicators\Graphs and Charts\Digital Waveform Graph\Digital Waveform Graph.lvproj for examples of the digital waveform graph.

Digital Waveform Data Type

The [digital waveform](#) data type carries start time, delta x, the data, and the attributes of a digital waveform. You can use the Build Digital Waveform function to create a digital waveform. When you wire digital waveform data to the [digital waveform graph](#), the graph

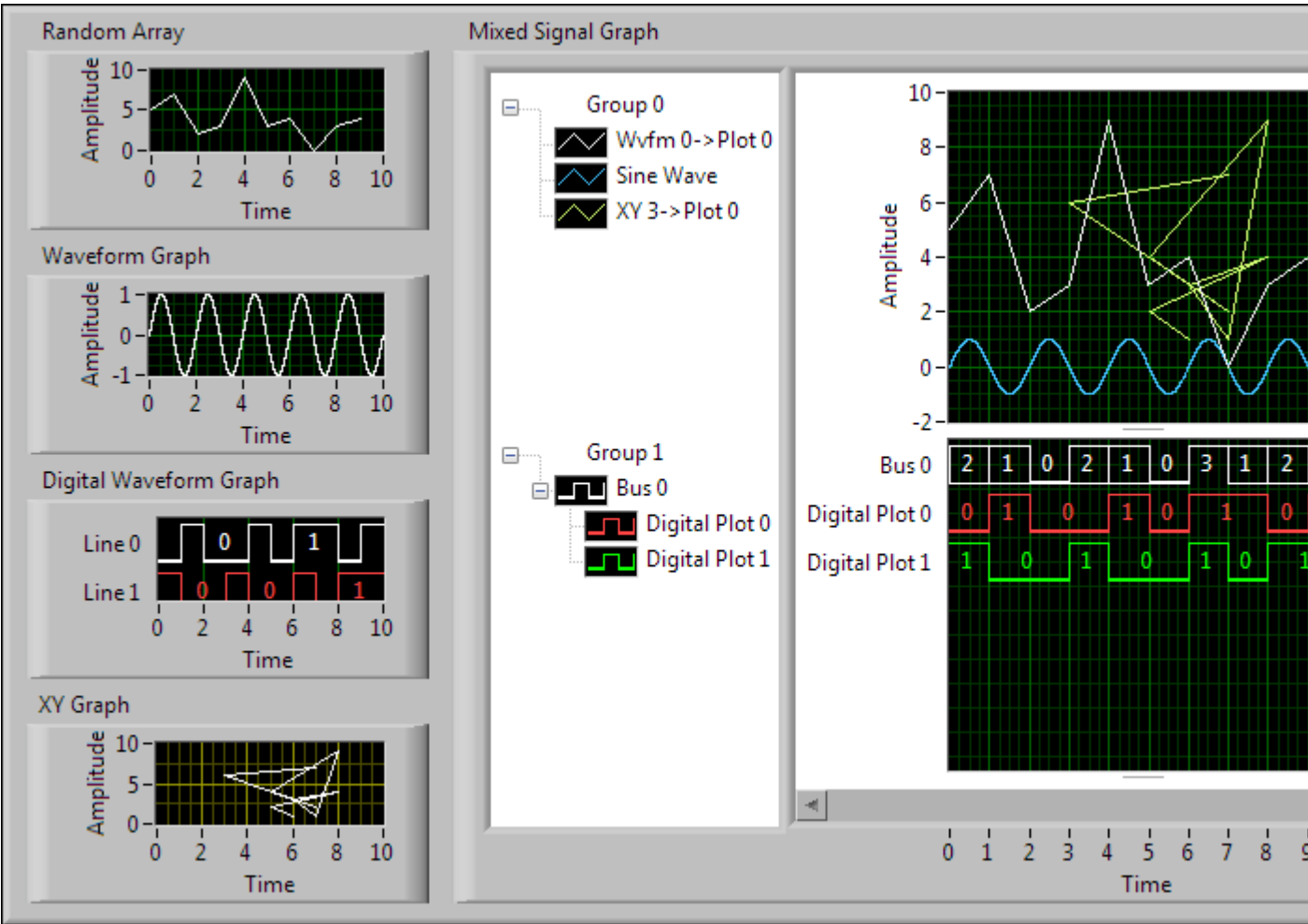
automatically plots a waveform based on the timing information and data of the digital waveform. Wire digital waveform data to a digital data indicator to view the samples and signals of a digital waveform.

Mixed Signal Graphs

The mixed signal graph can display both analog and digital data, and it accepts all data types accepted by *waveform graphs*, *XY graphs*, and *digital waveform graphs*.

A mixed signal graph may have multiple plot areas. A given plot area can display only digital or analog plots, not both. The plot area is where LabVIEW draws the data on the graph. The mixed signal graph automatically creates plot areas when necessary to accommodate analog and digital data. When you add multiple plot areas to a mixed signal graph, each plot area has its own y-scale. All of the plot areas share a common x-scale, allowing for the comparison of multiple signals of digital and analog data.

By default, the mixed signal graph displays digital data as digital lines and buses in the plot area. If you wire an array of digital data where each array element represents a bus, the mixed signal graph plots each column of digital data as a different line. The following front panel shows an example of a mixed signal graph.



You can [configure a plot](#) to change how a mixed signal graph displays numbers and plot lines. You also can use the Plot properties to get or set the format of numbers in the digital plots programmatically.

Displaying a Single Plot on Mixed Signal Graphs

The mixed signal graph accepts the same data types for single-plot mixed graphs as the [waveform graph](#), [XY graph](#), and [digital waveform graph](#).

Refer to the Mixed Signal Graph VI in the `labview\examples\Controls` and `Indicators\Graphs and Charts\Mixed Signal Graph` for examples of the data types that a mixed signal graph accepts.

Displaying Multiple Plots on Mixed Signal Graphs

The mixed signal graph accepts the same data types for displaying multiple plots as the [waveform graph](#), [XY graph](#), and [digital waveform graph](#).

Plot areas can accept only analog or only digital data. When you wire data to a mixed signal graph, LabVIEW automatically creates plot areas to accommodate combinations of analog and digital data. If there are multiple plot areas on the mixed signal graph, you can use the splitter bar between the plot areas to resize each plot area.

The plot legend on the mixed signal graph is comprised of tree controls and is displayed to the left of the graph plot areas. Each tree control represents one plot area. The plot area is labeled as **GroupX**, where X is the number corresponding to the order in which LabVIEW, or you, place the plot area on the graph. You can use the plot legend to [move plots from one plot area to another plot area](#). You can resize or hide the plot legend by moving the splitter bar that is between the plot area and the plot legend.

Refer to the Mixed Signal Graph VI in the `labview\examples\Controls` and `Indicators\Graphs and Charts\Mixed Signal Graph` for an example of displaying multiple plots on a mixed signal graph.

2D Graphs

A 2D graph uses x and y data to [plot points on the graph](#) and connect the points, forming a two-dimensional surface view of the data. With 2D graphs, you can visualize two-dimensional data on XY graphs because all 2D graphs are XY Graphs. Use the 2D graph properties to modify the way data appears in the 2D graphs.

When you add a 2D graph to the front panel, LabVIEW wires the graph on the block diagram to one of the helper VIs, depending on which 2D graph you select. The helper VIs convert the input data types into the generic data type the 2D graph accepts. LabVIEW includes the following types of 2D graphs:

- **Compass Plot** —Graphs vectors that emanate from the center of a compass graph.
- **Error Bar Plot** —Graphs the error bar at each point above and below the line graph.

- **Feather Plot** —Graphs vectors that emanate from equally spaced points along a horizontal axis.
- **XY Plot Matrix** —Graphs rows and columns of scatter graphs.

Refer to `labview\examples\Controls and Indicators\Graphs and Charts\Math Plots - 2D\Math Plots - 2D.lvproj` for examples of plotting data on a 2D graph.

Find related examples

3D Graphs

For many real-world data sets, such as temperature distribution on a surface, joint time-frequency analysis, and the motion of an airplane, you need to visualize data in three dimensions. With the 3D graphs, you can visualize three-dimensional data and alter the way that data appears by modifying the 3D graph properties.

LabVIEW includes the following types of 3D graphs:

- **Scatter** —Shows trends in statistics and the relationship between two sets of data.
- **Stem** —Displays an impulse response and organize data by its distribution.
- **Comet** —Creates an *animated graph* with a circle that follows the data points.
- **Surface** —Graphs data with a connecting surface.
- **Contour** —Graphs a plot with contour lines.
- **Mesh** —Graphs a mesh surface with open spaces.
- **Waterfall** —Graphs the surface of the data and the area on the y-axis below the data points.
- **Quiver** —Generates a plot of normal vectors.
- **Ribbon** —Generates a plot of parallel lines.
- **Bar** —Generates a plot of vertical bars.
- **Pie** —Generates a pie chart.
- **3D Surface Graph** —Draws a surface in 3D space.
- **3D Parametric Graph** —Draws a parametric surface in 3D space.
- **3D Line Graph** —Draws a line in 3D space.



Note 3D graph controls are only available in the LabVIEW Full and Professional Development Systems.

- **ActiveX 3D Surface Graph** —Draws a surface in 3D space using ActiveX technology.
- **ActiveX 3D Parametric Graph** —Draws a parametric surface in 3D space using ActiveX technology.
- **ActiveX 3D Curve Graph** —Draws a line in 3D space using ActiveX technology.

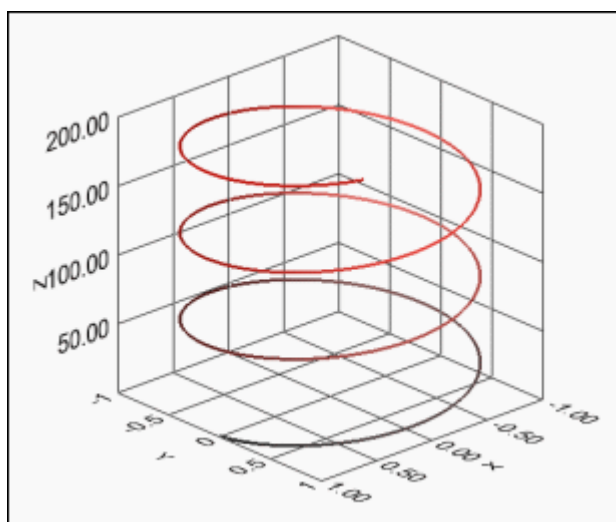


Note ActiveX 3D graph controls are supported only on Windows in the LabVIEW Full and Professional Development Systems.

Use the 3D Graphs, except for the 3D Surface, 3D Parametric, and 3D Curve graphs, in conjunction with the 3D Plot Properties dialog box to [plot graphs](#) with three dimensions. Refer to the `labview\examples\Controls and Indicators\Graphs and Charts\Math Plots - 3D` directory for examples of plotting data on a 3D graph.

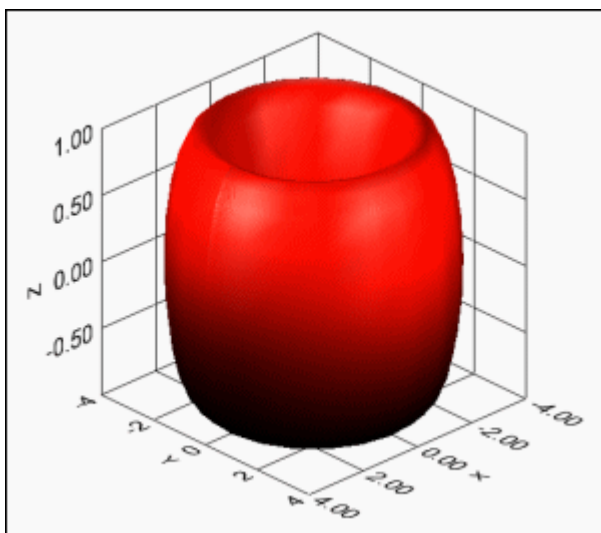
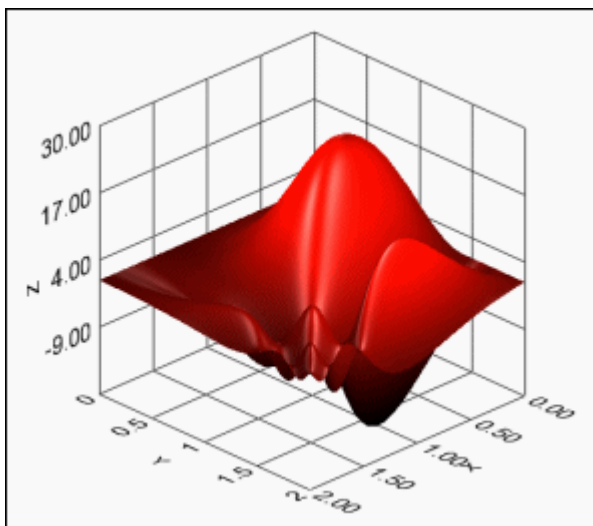
Find related examples

Use the 3D Surface, 3D Parametric, and 3D Curve graphs in conjunction with the 3D Graph Properties dialog box to plot curves and surfaces. A curve contains individual points on the graph, each point having an x, y, and z coordinate. The VI then connects these points with a line. A curve is ideal for visualizing the path of a moving object, such as the flight path of an airplane. The following illustration shows an example of a 3D Line Graph and is similar to the ActiveX 3D Curve Graph.



Note Use the 3D Graph Properties VIs to plot curves and surfaces on the ActiveX 3D graphs.

A surface plot uses x, y, and z data to plot points on the graph. The surface plot then connects these points, forming a three-dimensional surface view of the data. For example, you can use a surface plot for terrain mapping. A parametric plot is a surface plot that uses the parameters of a parametric function to determine the curves of the plot. You can use a parametric plot for graphing geometric solid objects. The following illustration shows examples of a 3D Surface Graph and a 3D Parametric Graph.



When you add a 3D graph to the front panel, LabVIEW wires the graph on the block diagram to one of the helper VIs, depending on which 3D graph you select. The helper VIs convert the input data types into the generic data type the 3D graph accepts.

The 3D graphs use graphical hardware acceleration in the render window, which can offer performance benefits. Right-click the 3D graph and select **Render Window** from the shortcut menu to view the 3D graph in the render window.

The ActiveX 3D graphs use ActiveX technology and VIs that handle 3D representation. When you select an ActiveX 3D graph, LabVIEW adds an ActiveX container to the front panel that contains a 3D graph control. LabVIEW also places a reference to the ActiveX 3D graph control on the block diagram. LabVIEW wires this reference to one of the three 3D Graph VIs. (Windows) The ActiveX 3D graph uses graphical hardware acceleration in the front panel window.

Refer to the `labview\examples\Controls and Indicators\Graphs and Charts\Waveform Graphs and Charts` directory for examples of plotting data on a 3D graph.

Customizing Graphs and Charts

Each graph and chart includes many options that you can use to customize appearance, convey more information, or highlight data. Although graphs and charts plot data differently, they have several common options that you access from the shortcut menu. However, some options are available only for a specific type of [graph](#) or [chart](#).

Using Multiple X- and Y-Scales

Use multiple scales on a graph or chart to display multiple plots that do not share a common x- or y-scale. Right-click the scale of the graph or chart and select **Duplicate Scale** from the shortcut menu to add multiple scales to the graph or chart. Right-click the plot in the plot legend and select the scale to associate with the plot from the **X scale** or **Y scale** shortcut menus.

You can [add multiple x- and y- scales to waveform](#), XY, intensity, and digital waveform graphs. Mixed signal graphs and all charts support only multiple y-scales.

Autoscaling

All graphs and charts, except for the 3D graphs, automatically enable autoscaling, which means they adjust their horizontal and vertical scales to fit the data you wire to them. By default, autoscaling is enabled for graphs and charts. However, autoscaling can slow performance. Right-click the graph or chart and select **X Scale»AutoScale X** or **Y Scale»AutoScale Y** from the shortcut menu to turn autoscaling on or off.



NOTE

For the Compass Plot, Error Bar Plot, Feather Plot, and XY Plot Matrix, select **Autoscale X** or **Autoscale Y** from the shortcut menu.

Use the Operating tool or the Labeling tool to change the horizontal or vertical scale directly.



Note LabVIEW does not include hidden plots when you autoscale the axes of a graph or chart. If you want to include the hidden plots when you autoscale, make the hidden plots transparent instead. Right-click the plot image in the plot legend and select **Color** from the shortcut menu to change the color of plots.

When a graph or chart scale resizes, other elements on the graph or chart move and resize. To disable this behavior so the plot area size stays fixed, right-click the graph or chart and select **Advanced»Auto Adjust Scales** from the shortcut menu. If you disable this behavior, the scales might clip or overlap each other.



Note The **Auto Adjust Scales** option does not apply to the Compass Plot, Error Bar Plot, Feather Plot, XY Plot Matrix, or the 3D graphs.

Formatting X- and Y-Scales

To configure the scales for the graph or chart, right-click the graph or chart and select **Properties** from the shortcut menu to display the **Graph Properties** dialog box or **Chart Properties** dialog box. Use the Display Format page of the **Properties** dialog box to specify how the scales of the x-axis and y-axis appear on graphs or charts, except for the 3D graphs. By default, the x-scale has a label of Time and the y-scale has a label of Amplitude. Use the **Format** page of the 3D Plot Properties or 3D Graph Properties dialog boxes to specify how the scales of the axes appear on the 3D graphs.

Use the **Display Format** or **Format** pages also to specify a numeric format for the scales of a graph or chart. Select **Advanced editing mode** to display the text options that let you enter format strings directly. Enter *format strings* to customize the appearance and numeric precision of the scales.

Click the **Scales** tab of the **Properties** dialog box, the **Axes** tab of the **3D Graph Properties** dialog box, or the **Axis** page of the **3D Plot Properties** dialog box to rename the scale and to format the appearance of the axis scale.

Customizing Graph and Chart Appearance

Customize the appearance of a graph or chart with the shortcut menu options. Right-click the graph or chart and select **Visible Items** from the shortcut menu to display or hide the following options:



Note These options are not available with the 3D graphs.

- **Plot Legend** —Defines the color and style of plots. Resize the legend to display multiple plots. This option does not apply to intensity graphs or charts.
- **Scale Legend** —Defines labels for scales and configures scale properties.
- **Graph Palette** —Allows you to move the cursor and zoom and pan the graph or chart while a VI runs.
- **Cursor Legend** —Displays a marker at a defined point coordinate. You can display multiple cursors on a graph. This option applies only to graphs.
- **X Scrollbar** —Scrolls through the data in the graph or chart. Use the scroll bar to view data that the graph or chart does not currently display.
- **X Scale and Y Scale** —Formats the x- and y-scales.
- **Digital Display** —Displays the numeric value of the chart. This option applies only to waveform charts.

Using Plot Legends

Use the plot legend of graphs and charts to view the plots LabVIEW draws on a plot area and to customize how a plot appears in the graph or chart. Customize a plot legend by using the Legend properties programmatically.



Note The *plot legend* of the Compass Plot, Error Bar Plot, and Feather Plot is different from the plot legends on the other graphs.

Adding and Displaying Plots in a Plot Legend

To add a plot to a plot legend, *use the Positioning tool*. Use the **Appearance** page of a graph or chart *Properties* dialog box to *specify the number of plots* in the plot legend of a graph or chart. You also can use the Legend:Number of Rows property to set the number of plots in the plot legend programmatically.

Choosing Which Plots to Display on a Graph or Chart

To choose whether to display a plot on a graph or chart, right-click the glyph in the plot legend and select **Plot Visible** from the shortcut menu. If there are multiple plots, you also can right-click the plot legend and select **Visible Items»Plot Visibility Checkbox** to display a checkbox next to each plot. Use the checkboxes to choose which plots you want to display in the plot area.

Using the Plot Legend to Customize the Plot Appearance

Customizing the plots that appear in the plot area of a graph or chart helps distinguish each plot from another. Use the plot legend to customize how each plot appears in the plot area of a graph or chart. Click a glyph in the plot legend and select from the following options in the shortcut menu to customize how the plot appears in the plot area of a graph or chart:

- **Plot Visible** —Sets whether to display the plot on the graph or chart.
- **Common Plots** —Provides options for the plot type. You can select a line, scatter, line and symbol, fill, comb, or bar plot type.
- **Color** —Displays the *color picker* so you can select the color of the plot.
- **Line Style** —Provides solid and dashed line styles.
- **Line Width** —Provides line widths.
- **Anti-Aliased** —Place a checkmark next to this item to make line plots appear smoother. However, anti-aliased lines can be computation intensive and can slow performance.
- **Bar Plots** —Provides bar plot options.
- **Fill Base Line** —Provides options for filling in space below the plot with the same color as the plot. If the graph or chart has multiple plots, you can select a plot at the bottom of the shortcut menu to fill the space between both plots.
- **Interpolation** —Provides interpolation options, including plotting only data points.
- **Point Style** —Provides point styles for the data points.
- **X Scale** —Sets the variable to associate with the x-axis.
- **Y Scale** —Sets the variable to associate with the y-axis.

You also can use the Plot properties to customize how a plot appears in the plot area programmatically.



Note The **Common Plots**, **Line Width**, **Anti-Aliased**, **Bar Plots**, **Fill Base Line**, **Interpolation**, and **Point Style** options are not available for digital waveform graphs.

Adding a Scroll Bar to a Plot Legend

You can *add a vertical or horizontal scroll bar* to the plot legend of graphs and charts. Use a scroll bar to view plots in a plot legend without exposing all the plots at any one time.

Using the Graph Palette

Use the graph palette to interact with a graph or chart while you run the VI. Right-click the graph or chart and select **Visible Items»Graph Palette** from the shortcut menu to display the graph palette, shown as follows.



Click a button in the graph palette to move *cursors*, *zoom*, or pan the display. Each button displays a green LED when you enable the button. The graph palette appears with the following buttons, in order from left to right:

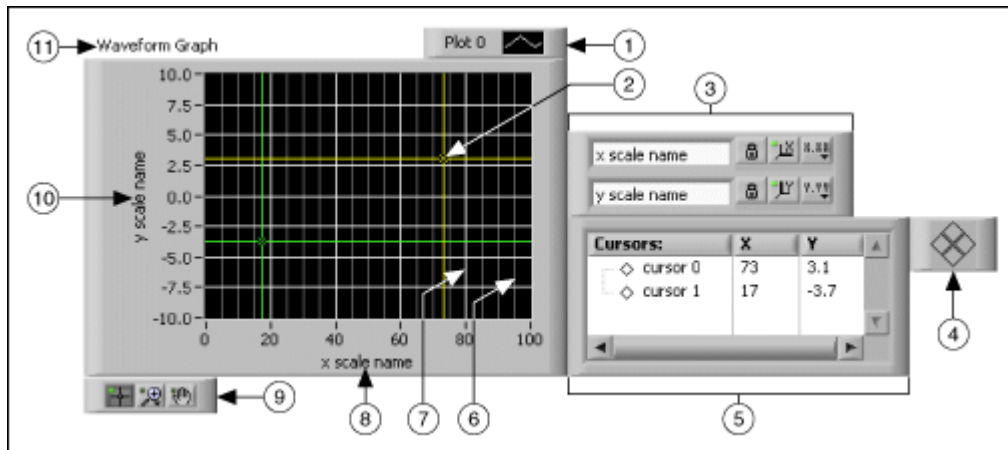
- **Cursor Movement Tool** —Moves the *cursor* on the display. This button applies only to graphs.
- **Zoom** —Zooms in and out of the display.
- **Panning Tool** —Picks up the plot and moves it around on the display.



Note The graph palette is not available for the 3D graphs.

Customizing Graphs

Each graph includes options that you can use to customize the graph to match your data display requirements. For example, you can modify the behavior and appearance of *graph cursors* or configure *graph scales*. The following illustration shows the elements of a graph.



1	Plot Legend
2	Cursor
3	Scale legend
4	Cursor mover
5	Cursor legend
6	Minor-grid mark
7	Grid mark
8	X-scale
9	Graph palette
10	Y-scale
11	Label



Note The plot legend for the Mixed Signal Graph and Digital Waveform Graph is on the left of the graph.

You add the plot legend, scale legend, cursor legend, graph palette, and label by right-clicking the graph, selecting **Visible Items** from the shortcut menu, and selecting the appropriate element.



Note The Plot Legend, Cursor, Scale Legend, Cursor mover, Cursor Legend, and Graph Palette do not apply to the 3D graphs.

Using Cursor Legends

Right-click the graph and select **Visible Items»Cursor Legend** from the shortcut menu to view the cursor legend. To *add a cursor to the graph*, right-click anywhere in the cursor legend, select **Create Cursor**, and select a cursor mode from the shortcut menu.

The cursor position is defined by the cursor mode. The cursor includes the following modes:

- **Free** —Moves the cursor freely within the plot area, regardless of plot positions.
- **Single-Plot** —Positions the cursor only on the plot that is associated with the cursor. You can move the cursor along the associated plot. Right-click the cursor legend row and select **Snap To** from the shortcut menu to associate one or all plots with the cursor.
- **Multi-Plot** —Positions the cursor only on a specific data point in the plot area. The multi-plot cursor reports values at the specified x-value for all of the plots with which the cursor is associated. You can position the cursor on any plot in the plot area. Right-click the cursor legend row and select **Snap To** from the shortcut menu to associate one or all plots with the cursor. This mode is valid only for mixed signal graphs.



Note You cannot change the mode of a cursor after you create it. You must *delete the cursor* and create another cursor.

Use a cursor on a graph to read the exact value of a point on a plot. The cursor value displays in the cursor legend.

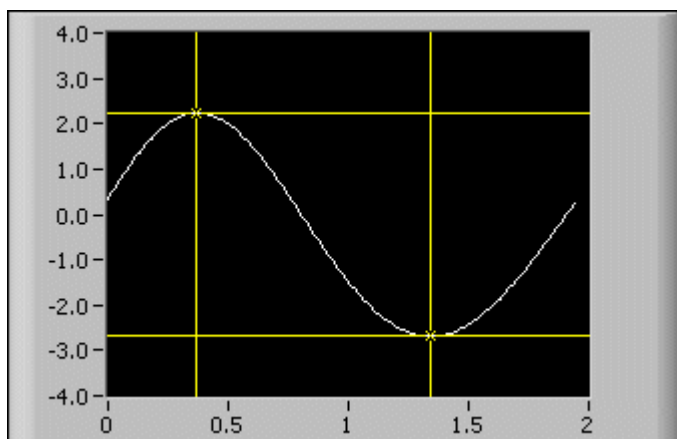


Note In a multi-plot graph or chart, the y value in the cursor legend may sometimes display in parentheses. This happens if there is no y value in the plot corresponding to the current x position of the cursor, and thus the y value has to be interpolated.



Tip Use an *annotation* instead of a cursor to highlight data points on the plot.

The following front panel shows an example of a graph with multiple cursors.



If you drag a graph cursor past the edge of the graph, the graph scrolls in the direction of the cursor. To disable this behavior, right-click the graph and select **Advanced»Cursors Scroll Graph** from the shortcut menu. If you disable this behavior, the scales do not update when you drag the cursor past the edge of the graph.



Note The **Cursors Scroll Graph** option does not apply to the Compass Plot, Error Bar Plot, Feather Plot, and XY Plot Matrix.

You can customize the appearance of the cursor in several ways. You can label the cursor on the plot, specify the color of the cursor, and specify line, point, and cursor style. Right-click the cursor legend row and select items from the shortcut menu to [customize the cursor](#).

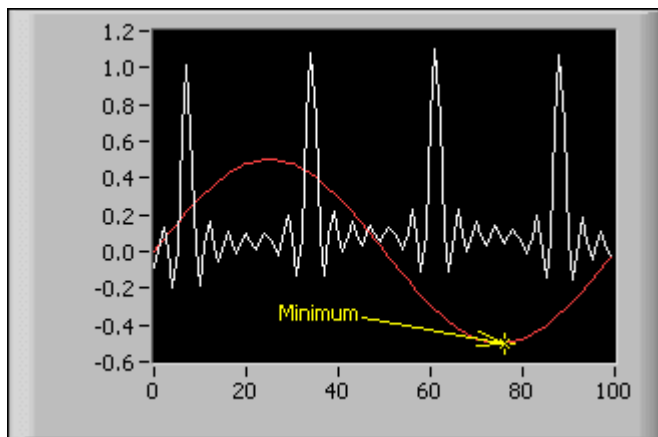
Using Graph Annotations

Use annotations on a graph to highlight data points in the plot area. The annotation includes a label and an arrow that identifies the annotation and data point. A graph can have any number of annotations.



Note You cannot use annotations on the 3D graphs.

The following front panel shows an example of a graph with annotations.



Right-click the graph and select **Data Operations»Create Annotation** from the shortcut menu to display the Create Annotation dialog box. Use the **Create Annotation** dialog box to specify the annotation name and how the annotation snaps to plots in the plot area.

Use the **Lock Style** pull-down menu in the Create Annotation dialog box to specify how the annotation snaps to plots in the plot area. The **Lock Style** component includes the following options:

- **Free** —Allows you to move the annotation anywhere in the plot area. LabVIEW does not snap the annotation to any plots in the plot area.
- **Snap to All Plots** —Allows you to move the annotation to the nearest data point along any plot in the plot area.
- **Snap to One Plot** —Allows you to move the annotation only along the specified plot.

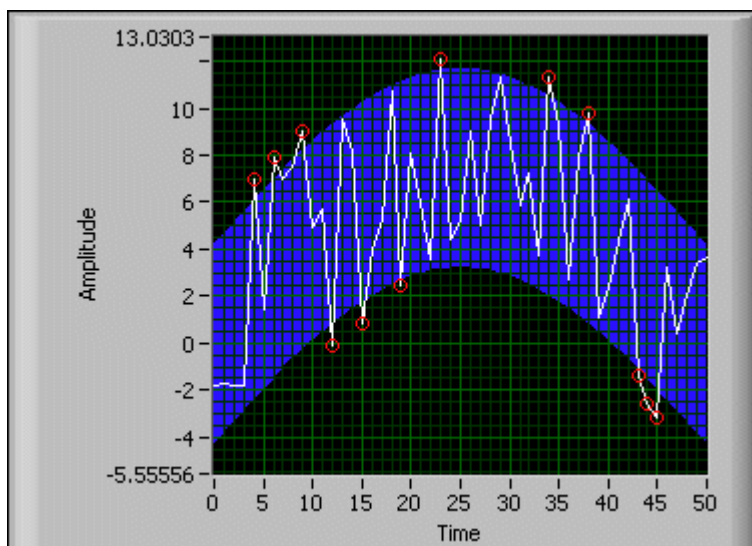
You can customize the behavior and appearance of the annotation in several ways. You can hide or show the annotation name or arrow in the plot area, specify the color of the annotation, and specify line, point, and annotation style. Right-click the annotation and select options from the shortcut menu to [customize the annotation](#).

To delete the annotation, right-click the annotation and select **Delete Annotation** from the shortcut menu. Right-click the graph and select **Data Operations»Delete All Annotations** from the shortcut menu to delete all annotations in the plot area.

Drawing to the Graph Plot Area

Customize the graph plot area of intensity, mixed signal, and waveform graphs by drawing a background, foreground, or middle image in the plot area. The canvas for drawing plot images has a coordinate system in which the origin (0,0) is always located at the top, leftmost corner of the graph plot area. You can use the Plot Images properties of the IntensityGraph, MixedSignalGraph, and WaveformGraph classes to create custom grids in the background, data envelopes in the middle, or use shapes to annotate data points in the foreground of the graph plot area.

The following front panel displays an example of how to create circles to annotate data points and an envelope to designate a defined tolerance for the data.



The background image resides behind the plot data and the grid lines. Use the Plot Images:Back property to set this image for mixed signal and waveform graphs.

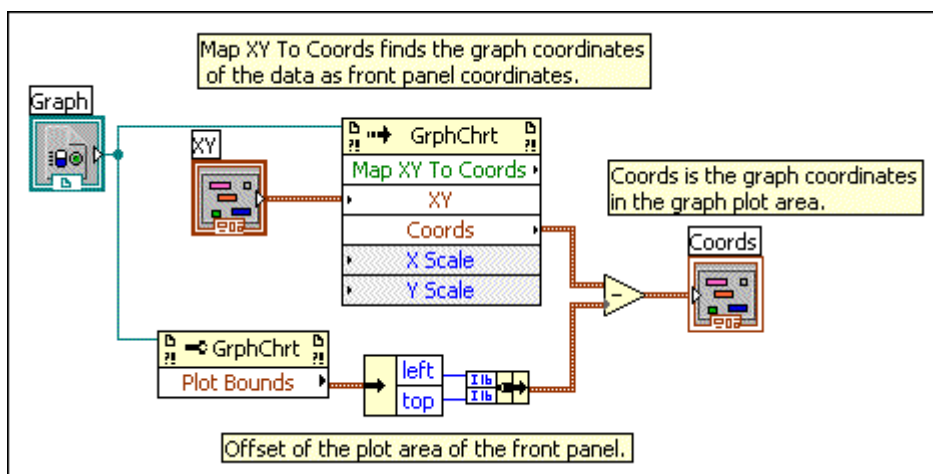
The foreground image resides in front of the plot data and the grid lines. Use the Plot Images:Front property to set this image for intensity, mixed signal, and waveform graphs.

The middle image resides between the grid lines and the plot data. Use the Plot Images:Middle property to set this image for mixed signal and waveform graphs.



Note You can draw images only in the foreground of an intensity graph.

When you use the Plot Images properties, LabVIEW draws the image beginning at the origin of the graph plot area. If you want to use the Plot Images properties to draw an image that uses the graph data points as references rather than the plot area origin (0,0), use the Map XY To Coordinates method to [map the graph coordinates of the data points](#) to coordinates on the front panel. The Map XY To Coordinates method allows you to account for the offset of the plot area of the graph from the origin of the front panel to find the correct location in which to draw. The following block diagram displays one option you can use to find the correct location in which to draw.





Note Using *splitter bars* on the front panel creates multiple panes. You then account for the offset of the plot area of the graph from the origin of the owning pane on the front panel.

Refer to the WaveformGraph properties, the IntensityGraph properties, and the MixedSignalGraph properties for more information about the Plot Images properties each graph supports.

If you want to remove images, you must wire an empty image to the appropriate property, or you must make the image *transparent*. Also, when you resize or copy the graph with custom annotations or envelopes, you must rerun the VI to redraw the image.

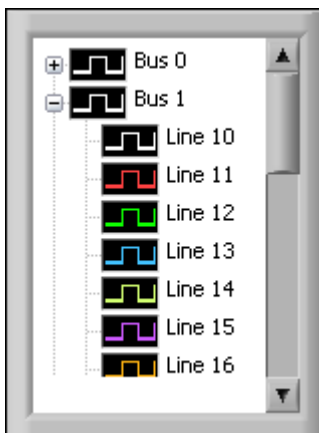
Refer to the `labview\examples\Controls and Indicators\Graphs and Charts\Graphs - General\Graphs - General.lvproj` for examples of using the Plot Images properties and the Map XY To Coordinates method.

Customizing Digital Waveform Graphs

Customize the digital waveform graph to display one of two views of the plot legend, set the type of data displayed in the plot area, and alter the appearance of the digital lines and buses in the plot area.

Customizing Plot Legend Views

The digital waveform graph provides two views of the plot legend. The following plot legend in tree view is the default view that allows you to view digital lines and buses and to expand and contract digital buses.

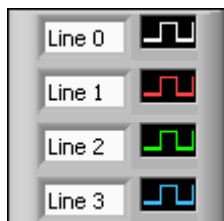


To expand and contract buses in the tree view of the plot legend, click the expand and contract symbol to the left of the bus. Scroll bars appear automatically when a bus or line name exceeds the size of the plot legend and when the amount of lines, buses, or lines and buses exceeds the size of the plot legend.

You can *customize* line thickness, set transitions type and location, format graph labels, and specify names for digital lines and buses that appear on digital waveform graphs. Use

the Group Digital Signals Express VI to group digital waveforms in the order you want to view them in the digital waveform graph.

The following plot legend is the standard view of the plot legend and allows you to view digital lines or buses.



To change the tree view of the plot legend to the standard view of the plot legend, right-click the digital waveform graph and select **Advanced»Change Plot Legend to Standard View** from the shortcut menu. To change back to the tree view of the plot legend, right-click the digital waveform graph and select **Advanced»Change Plot Legend to Tree View** from the shortcut menu.



Note LabVIEW deletes any *allocated lines* when you change from the standard view of the plot legend to the tree view of the plot legend, including any *custom line settings*. Additionally, you cannot allocate lines in the plot legend when the tree view is enabled.

To disable the plot legend regardless of the selected view, right-click the digital waveform graph and select **Advanced»Disable Plot Legend** from the shortcut menu to place a checkmark next to the menu item. You also can use the Legend:Disable property to disable the plot legend programmatically.

To resize the plot legend, move the Positioning tool over the legend border. When resizing handles appear at the corners of the legend, drag a handle up, down, left, or right and release the mouse button when the legend is the desired size.

Customizing the Data in the Plot Area

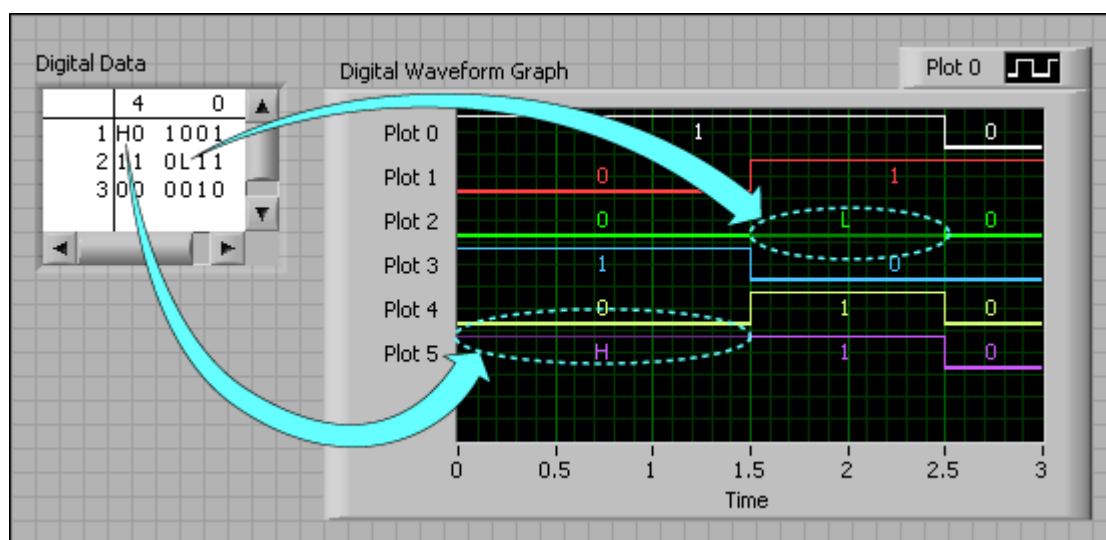
By default, the digital waveform graph displays both digital lines and buses in the plot area. *Customize the plot area* of the digital waveform graph to display only the data you want to view.

If a digital waveform graph includes digital data in both drive and compare logic states, by default the compare data appears darker on the plot than the drive data. If you do not want to darken compare data, right-click the plot and select **Advanced»Darken Compare Data** from the shortcut menu to remove the checkmark. You also can use the Darken Compare Data property to darken compare data programmatically.



Note This feature primarily applies to users generating digital I/O signals. Compared or expected response data refers to the marked highs and lows in a binary data stream. LabVIEW darkens these values to distinguish highs and lows from the rest of the drive or stimulus data.

The following front panel shows darkened compare data.



Customizing 2D Graphs

You can customize the appearance, format, plot styles, scales, and cursors of the Compass Plot, Error Bar Plot, Feather Plot, and XY Plot Matrix when you right-click a **2D graph** and select **Properties** from the shortcut menu to display the Properties dialog box.

Use the plot legend to the right of the graph to customize the plots for the Compass Plot, Error Bar Plot, and Feather Plot. Click the plot name to edit the name. Change the appearance of a plot by clicking the image on the right of the plot name to display the following options:

- **Line style** —Sets the line style of the plot.
- **Line width** —Sets the width of the plot.
- **Size** —Sets the arrow size for the Compass and Feather graphs and the error bar size for the Error Bar graph.
- **Color** —Sets the color of the plot line.



Note To close the dialog box, click the checkmark to make changes or the red x to cancel changes.

Refer to the labview\examples\Controls and Indicators\Graphs and Charts\Math Plots - 2D directory for examples of plotting data on a 2D graph.

Find related examples

Customizing 3D Graphs

You can customize the Appearance of all **3D graphs** when you right-click a 3D graph and select **Properties** from the shortcut menu to display the Properties dialog box.

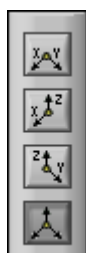
For the 3D graphs, except the 3D Line, 3D Parametric, and 3D Surface graphs, right-click the graph and select **3D Plot Properties** from the shortcut menu to display the 3D Plot Properties dialog box. Use this dialog box to customize the graph, lighting, axes, value pairs, format, and cursors.

For the 3D Line, 3D Parametric, and 3D Surface graphs, right-click the graph and select **3D Graph Properties** from the shortcut menu to display the 3D Graph Properties dialog box. Use the **3D Graph Properties** dialog box to format the properties of a 3D graph. Manipulate the controls on each page of the dialog box to change the appearance of the 3D graph.

(Windows) Because the ActiveX 3D graphs use **ActiveX technology** and VIs that handle 3D representation, you set options for the ActiveX 3D graphs differently than you set options for other graphs. While creating an application, use the **ActiveX Control Property Browser** to set properties for an ActiveX 3D graph. Right-click the ActiveX 3D graph and select **Property Browser** from the shortcut menu to display the ActiveX Control Property Browser. If you want to allow users to change common properties at run time or you need to set a property programmatically, use the 3D Graph Properties VIs.

For the Bar and Pie graphs, use the color palette to the right of the graphs to customize each bar or pie slice. To change the name of a bar or pie slice, highlight the name and enter new text. To change the color of a bar or pie slice, click the color to the right of the name and select a new color.

For the 3D graphs, use the following projection palette near the bottom-right corner of the graph to customize the plane projection view:



The projection palette contains the following buttons:

- **X-Y plane projection** —Displays the plot projection on the X-Y plane.
- **X-Z plane projection** —Displays the plot projection on the X-Z plane.
- **Y-Z plane projection** —Displays the plot projection on the Y-Z plane.
- **Default projection** —Displays the plot on the 3D plane.



Note The projection palette does not appear with the Bar, 3D Line, 3D Parametric, and 3D Surface graphs.

Refer to the `labview\examples\Controls and Indicators\Graphs and Charts\Math Plots - 3D` directory for examples of plotting data on a 3D graph.

Find related examples

Refer to the `labview\examples\Controls and Indicators\Graphs and Charts\Math Plots - 3D` directory for examples of plotting data on a 3D graph.

Find related examples

Customizing Charts

Unlike the graph, which displays new data that overwrites any stored data, the chart updates periodically and maintains a history of the data previously stored.

You can customize the chart to match your data display requirements. Options available for all charts include a scroll bar, the [scale legend](#), the [graph palette](#), a digital display, and representation of scales with respect to time. You also can modify the behavior of [chart history length](#), [update modes](#), and plot displays.

Refer to the Chart Data Types and Update Modes VI in the `labview\examples\Controls and Indicators\Graphs and Charts\Waveform Graphs and Charts` directory for examples of different kinds of charts and the data types they accept.

Configuring Chart History Length

LabVIEW stores data points already added to the chart in a buffer, or the chart history. Right-click the chart and select **Chart History Length** from the shortcut menu to configure the history buffer.



Tip Large chart history values can be memory intensive. Unlike graphs, charts keep the history of data previously written. When a chart runs continuously, its history grows and requires additional memory space. This continues until the chart history is full, then LabVIEW stops taking more memory. LabVIEW does not automatically clear the chart history when the VI restarts. You can clear the chart history throughout the execution of the program. To do this, write empty arrays to the History Data attribute node for the chart.

View previously collected data using the chart scroll bar. Right-click the chart and select **Visible Items»X Scrollbar** from the shortcut menu to display a scroll bar.

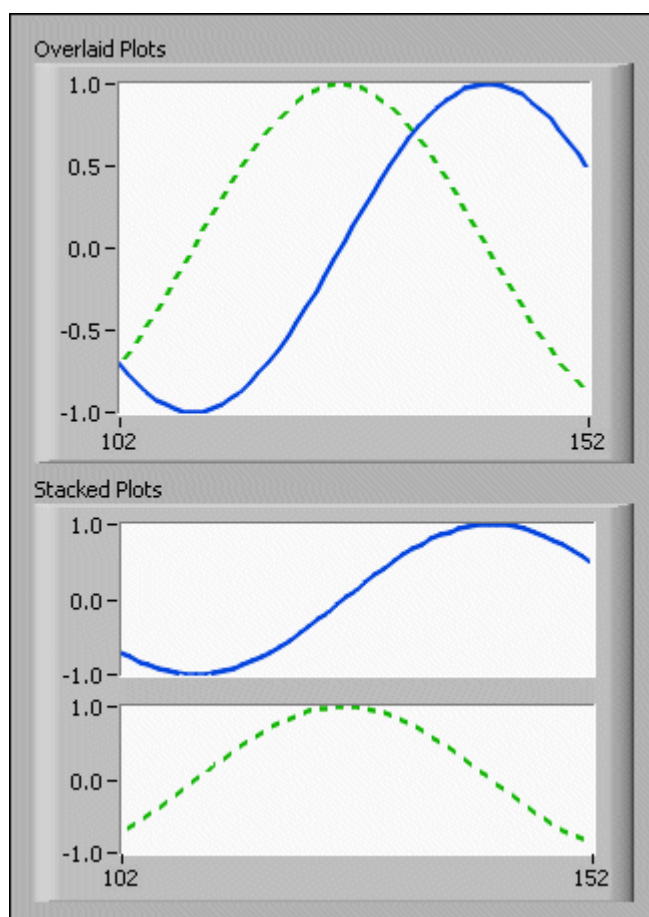
Configuring Chart Update Modes

Configure how the chart updates to display new data. Right-click the chart and select **Advanced»Update Mode** from the shortcut menu to set the chart update mode. The chart uses the following modes to display data:

- **Strip Chart** —Shows running data continuously scrolling from left to right across the chart with old data on the left and new data on the right. A strip chart is similar to a paper tape strip chart recorder. **Strip Chart** is the default update mode.
- **Scope Chart** —Shows one item of data, such as a pulse or wave, scrolling partway across the chart from left to right. For each new value, the chart plots the value to the right of the last value. When the plot reaches the right border of the plotting area, LabVIEW erases the plot and begins plotting again from the left border. The retracing display of a scope chart is similar to an oscilloscope.
- **Sweep Chart** —Works similarly to a scope chart except it shows the old data on the right and the new data on the left separated by a vertical line. LabVIEW does not erase the plot in a sweep chart when the plot reaches the right border of the plotting area. A sweep chart is similar to an EKG display.

Using Overlaid and Stacked Plots

You can display multiple plots on a waveform chart by using a single vertical scale, called overlaid plots, or by using multiple vertical scales, called stacked plots. The following front panel shows examples of overlaid plots and stacked plots.



Right-click the chart and select **Stack Plots** from the shortcut menu to view the chart plots as multiple vertical scales. Right-click the chart and select **Overlay Plots** to view the chart plots as a single vertical scale.

Exporting Data from Graphs, Charts, Tables, and Arrays

You can [export data](#) from graphs, charts, tables, and arrays and then import the data to the clipboard, Microsoft Excel, or DIAdem.



Note You must have Microsoft Excel installed to export data to Excel. You must have DIAdem installed to export data to DIAdem. Refer to the National Instruments Web site at ni.com/diadem to learn more about DIAdem and to download the latest version of DIAdem.

For graphs and charts, you can export only data that is visible in the front panel window. For arrays and tables, you can select the cells, columns, and rows that you want to export. Before you export data, you can use techniques to [customize how data displays](#).

The following information describes some caveats and recommendations to consider when you export data:

- You can export an entire graph with multiple plots. You also can export a single plot from a graph.
- The data you export from a graph or a chart displays in two columns for each individual plot. The first column in a pair represents the x values of the plot, and the second column in a pair represents the y values of the plot.
- The data you export from [intensity graphs](#) displays in a single column for each plot. A column represents a plot and an individual cell represents a z value at an x/y location on the plot.
- You can export 1D or 2D [string arrays or numeric arrays](#).
- You also can export arrays of more than two dimensions. However, for arrays of more than two dimensions, you can export only the dimensions visible in the front panel window.

Exporting Images of Graphs, Charts, and Tables

You can [export images](#) of graphs, charts, tables, picture controls, digital data, and digital waveform controls and indicators into presentations, email, text documents, and so on. When you export a simplified image, LabVIEW exports only the control or indicator, digital display, plot legend, and index display and does not export scroll bars, the scale legend, the graph palette, or the cursor legend. To export an image, right-click the graph or chart, and select **Export»Export Simplified Image**.



Note The **Export Simplified Image** menu item is not available for intensity graphs, intensity charts, or 3D graphs. To export an image of a 3D graph, except for the 3D Surface Graph, 3D Parametric Graph, and 3D Line Graph, right-click the 3D graph and select **Export Image»Export to File**.

You can export images into the following formats:

(Windows) .emf, .bmp, and .eps files

(macOS) .pict, .bmp, and .eps files

(Linux) .bmp and .eps files

Save the exported image to the clipboard or to disk.

Using Anti-Aliased Line Plots

You can improve the appearance of line plots in a [graph or chart](#) by using anti-aliased lines. When you enable anti-aliased line drawing, line plots appear smoother. Anti-aliased line drawing does not alter line widths, line styles, point styles, and so on.



NOTE

Anti-aliased line drawing is not available on digital waveform graphs.

Right-click the plot legend and select **Anti-Aliased** from the shortcut menu to enable anti-aliased line plots. If the plot legend is not visible, right-click the graph or chart and select **Visible Items»Plot Legend** from the shortcut menu.



Note Because anti-aliased line drawing can be computation intensive, using anti-aliased line plots can slow performance.

Using Scale Legends

Use the scale legend to label scales and configure scale properties on a [graph or chart](#). Right-click the graph or chart and select **Visible Items»Scale Legend** from the shortcut menu to view the scale legend.

Use the Operating tool to click the **Scale Lock** button, shown as follows, to toggle autoscaling for each scale.



Use the **Autoscale** button, shown as follows, to adjust the scale to reflect the data you wire to the graph or chart.



Use the **Scale Format** button, shown as follows, to configure the format, precision, and mapping mode, set the visibility of the scale and scale label, and specify the grid color.



You also can use the scale legend to [add multiple y-scales](#) to the graph or chart. Use the Positioning tool to expand the scale legend to add a y-scale to the graph or chart. Click anywhere in the scale legend row to activate the scale.

Dynamically Formatting Graphs

Wire [dynamic data](#) or waveform data to a graph to automatically format the plot legend and x-scale time stamp for the graph. For example, if you configure the Simulate Signal Express VI to generate a sine wave and to use absolute time and wire the output of the Simulate Signal Express VI to a waveform graph, the plot legend of the graph automatically updates the plot label to Sine, and the x-scale displays the time and date when you run the VI.

To ignore the plot legend label the dynamic or waveform data includes, right-click the graph and select **Ignore Attributes** from the shortcut menu. Selecting **Ignore Attributes** allows you to change the plot legend label programmatically.

To ignore the time stamp configuration the dynamic or waveform data includes, right-click the graph and select **Ignore Time Stamp** from the shortcut menu.



NOTE

If the data you display in the graph contains a time stamp formatted for absolute time, select **Ignore Time Stamp** to ignore the time stamp.

Using Smooth Updates

When LabVIEW updates an object with smooth updates off, it erases the contents of the object and draws the new value, which results in a noticeable flicker. Using smooth updates avoids the flicker that erasing and drawing causes. Right-click the graph and select **Advanced»Smooth Updates** from the shortcut menu to use an offscreen buffer to minimize the flicker. Using smooth updates can slow performance depending on the computer and video system you use.

Using the 2D Picture Control

The 2D picture control includes a set of drawing instructions for displaying pictures that can contain lines, circles, text, and other types of graphic shapes. Because you have pixel-level control over the 2D picture control, you can create nearly any graphics object. Use the 2D picture control and the graphics VIs instead of a graphics application to create, modify, and view graphics in LabVIEW.



NOTE

You also can use the 3D picture control to display graphical representations of objects in [3D scenes](#). You can set characteristics of the 3D scene such as the style and location of a light source and how a user-controlled camera interacts with the 3D scene.

The 2D picture control has a pixel-based coordinate system in which the origin (0, 0) is the pixel located at the top left corner of the control. The horizontal x component of a coordinate increases toward the right, and the vertical y coordinate increases toward the bottom.

If a picture is too large for the 2D picture control that displays it, LabVIEW crops the picture so you can see only the pixels that fit within the display region of the control. Use the Positioning tool to resize the control and run the VI again to view the entire picture. You also can display vertical and horizontal scroll bars for a 2D picture control so you can view the pixels that do not fit within the display region of the control. Right-click the control and select **Visible Items»Scrollbar** from the shortcut menu to display scroll bars for the control.

You can use [VI Server](#) properties in the Picture class to modify the 2D picture control programmatically, such as to change the size of the 2D picture control or the picture area in the control.

When you place a 2D picture control on the front panel, it appears as a blank rectangular area, and a corresponding terminal, shown as follows, appears on the block diagram.



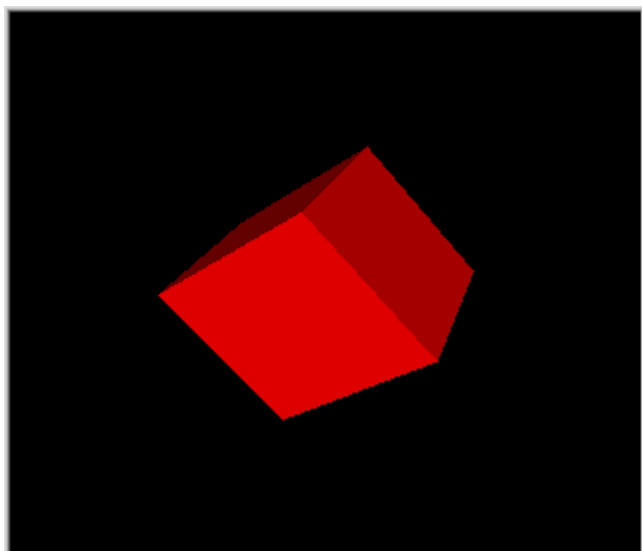
To display an image in a 2D picture control, you must write an image to the control programmatically. You cannot copy an image to the clipboard and paste it into the 2D picture control. You can [use the Picture Functions VIs](#) to specify a set of drawing instructions. Each VI takes a set of inputs that describes a drawing instruction. Based on these inputs, the VI creates a compact description of these specifications that you pass to the 2D picture control for display.

By default, the 2D picture control clears its display before drawing an image. If you want the 2D picture control to draw the new image over the previous image, right-click the control and remove the checkmark next to the **Erase First** shortcut menu item.

Using the 3D Picture Control

The 3D picture control displays graphical representations of [3D objects](#). A 3D scene is a 3D object or a collection of 3D objects that you can view in the 3D picture control or in a separate scene window. As you design a 3D scene, you can generate multiple 3D objects and specify their [orientation](#), [appearance](#), and [relationship to other objects](#) within the 3D scene. You can set characteristics of the 3D scene such as the style and location of a [light source](#) and [how a user-controlled camera interacts](#) with the 3D scene.

The following image shows a 3D picture control that contains a simple 3D scene.



Creating 3D Objects

The objects that appear in a 3D scene are graphical representations of physical objects. A 3D object is a set of vertices that exists in 3D space and any connections that may or may not exist between those vertices. When you [create a 3D scene](#), you create one or more 3D objects to comprise the 3D scene. Use the Create Object VI to create 3D objects.

After you create a 3D object, you must define how that object appears in the 3D scene. You can set the following drawable attributes for a 3D object:

- **Geometries** —A geometry specifies the shape of a 3D object. The Geometries VIs, such as Create Box and Create Sphere, define the representation of simple geometric objects. You also can use [meshes](#) to create more geometrically complex 3D objects.
- **Text** —If an object does not have a defined geometry, you can [apply text](#) to the object to create a text object. Like objects with defined geometries, the text appears when you render the 3D scene. Use the Create Text VI to create a new text and optionally to configure attributes of the text such as color and font. You also can use the SceneText properties and SceneText methods to configure attributes of the text.

LabVIEW draws new objects with a default size and color and displays the objects in the center of the 3D scene. However, you can use [transformations](#) and [textures](#) to change the size, orientation, location, or surface appearance of a 3D object in a 3D scene. You also can use the SceneObject properties and SceneObject methods to configure a 3D object.



NOTE

You cannot apply [texture](#) to a text object.

Applying Transformations

A transformation changes the orientation or appearance of a 3D object according to a mathematical rule. Use the Transformations VIs to change the size, orientation, or location

of an object in a 3D scene. You can apply the following three common transformations to 3D objects:

- **Translation** —Translating a 3D object moves the object to a new location in the 3D scene. To specify the new location for the 3D object, you define a vector along which to translate the object.
- **Rotation** —Rotating a 3D object changes the orientation of the 3D object. To specify the new orientation for the 3D object, you define an axis around which the object rotates and an angle around the axis by which the rotation occurs.
- **Scaling** —Scaling a 3D object changes the size of the 3D object. To specify the new scale for the 3D object, you set the factors in the x, y, and z planes of the 3D scene by which you want to change the size of the object.



Note To prevent 3D objects from appearing distorted, avoid scaling objects beyond the clipping planes. The default value of the near clipping plane, which defines the plane nearest the viewer, is 0 units. The default value of the far clipping plane, which defines the clipping plane farthest from the viewer, is 10000 units. You can use the Projection:Perspective method to set values for clipping planes programmatically.

For each transformation, you can perform an absolute transformation, which transforms the 3D object from the initial position of the object, or a relative transformation, which transforms the 3D object from the current position of the object. For example, use the Set Translation VI to translate a 3D object absolutely. Use the Translate Object VI to translate a 3D object relatively.

Refer to the Order of Transformations VI in the `labview\examples\Graphics and Sound\3D Picture Control` directory for an example of how to apply transformations to 3D objects.

Applying Textures

You can change the surface appearance of a 3D object by *wrapping a 2D image* around the 3D object. This process is called applying a texture. For example, you can turn a sphere into a planet by applying a 2D image of the planet surface to the sphere. The image of the planet surface is the texture. Use the Specials:Texturing property to enable texturing for each 3D object to which you want to apply a texture. Then, use the Texture VI to create a new texture. Use the SceneTexture properties and SceneTexture methods to configure the texture.

Refer to the 3D Model of Solar System VI in the `labview\examples\Graphics and Sound\3D Picture Control` directory for an example of applying texture to 3D objects.

Creating Relationships between 3D Objects

You can *create relationships between objects* in a 3D scene. Objects act as either parents or children. If you apply changes to a parent object, the child objects reflect those changes.

Use object relationships in large 3D scenes to program more efficiently and for 3D scenes that involve related movement. For example, in a replica of the solar system, a change in the revolution speed of the Earth affects the revolution speed of the moon. Create the moon as a child of the Earth to avoid configuring transformation changes for both objects.

Refer to the 3D Model of Solar System VI in the `labview\examples\Graphics and Sound\3D Picture Control` directory for an example of creating relationships between 3D objects.

Configuring Light Sources

LabVIEW can render a visible 3D scene with or without a configured light source. You can [configure a light source](#) to achieve visual effects such as sunlight or spotlights in a visible 3D scene. You can create three types of lights in a 3D scene: [directional](#) lights, [positional](#) lights, and [spotlights](#). Use the New Light VI to create a new light source. You also can use the SceneLight properties to configure additional aspects of light sources, such as color, attenuation, and intensity.

Directional Lights

Use a directional light to create a light source that appears infinitely distant from objects in a 3D scene, mimicking the effect of sunlight. Because the position of a directional light is at infinity, you define only a direction for the light. Use the Direction property to configure the light direction.

Positional Lights

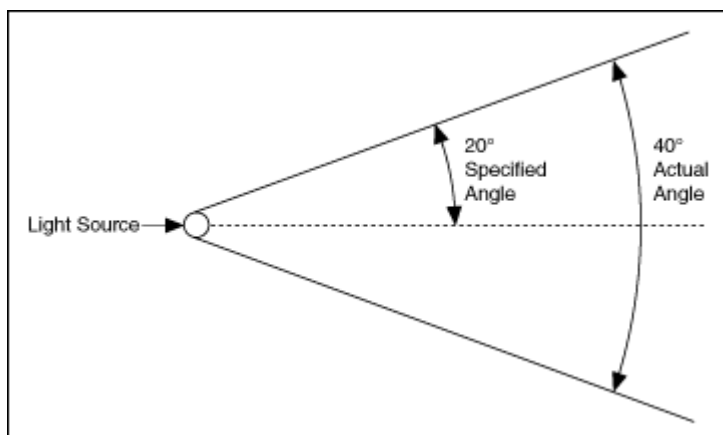
Use a positional light if you want to configure a specific position, or origin, for the light. For a positional light, you specify x, y, and z coordinates for the position of the light source and coordinates for the direction of the light. The x, y, and z coordinates that specify the position of the light are relative to the object in the scene that owns that light, where the position of the owning object represents the coordinates 0, 0, 0. For example, if you want to position a light source directly beneath a 3D object, you can set the position of the light source to 1 object coordinate beneath the object by using the coordinates 0, -1, 0. Another 3D object located at 0, -3, 0 relative to the first object appears beneath the positional light, and assuming the direction of the light is toward the first object, the light source does not affect the second object.

To add a positional light to a 3D scene, attach that light to an object in the 3D scene by wiring the object refnum and the light refnum to the Light:Add Light method. The object then owns the light. Use the Direction property to configure the light direction, and the Position property to configure the light position.

Spotlights

Use a spotlight if you want to specify both a position and a direction for a light. A spotlight is a positional light with a defined direction and angle. Set up the light as a positional light, but use the Cutoff property to define the angle of a spotlight. The value you wire to this property is equivalent to half the angle of the actual spotlight. For example, if you want the

spotlight to emit a 40 degree angle of light, set this property to 20. This property accepts values 0 to 90 and 180, where 180 represents no cutoff.



Refer to the Using Meshes VI in the labview\examples\Graphics and Sound\3D Picture Control directory for an example of configuring light sources.

Viewing a 3D Scene

You can [configure how a user-controlled camera](#) interacts with a 3D scene, configure a separate scene window in which to display the 3D scene, and configure a 3D scene to update continuously to display live data.

Configuring a Scene Camera

Right-click the 3D picture control and select **Camera Controller** from the shortcut menu to specify how the camera interacts with the 3D scene. The Operating tool acts as the camera that you move about the 3D scene. You also can use the SceneGraphDisplay properties or SceneWindow properties to configure the scene camera programmatically.

Configuring a Scene Window

You can [configure a separate scene window](#) in which to display the 3D scene when the VI runs. Use the Setup Window VI to configure a new scene window. You also can use the SceneWindow properties and SceneWindow methods to configure attributes such as the projection of the 3D scene and the size and active state of the scene window.

Live Data

You can configure a 3D scene to update continuously to display live, or continuously changing, 3D data. Enclose the 3D scene in a loop and using one of the following approaches:

- (Recommended) Use the Redraw method on the 3D picture reference inside the loop or wherever you want to redraw the scene.

- Place the 3D picture control terminal or local variable wherever you want to refresh the scene.

Refer to the 3D Model of Solar System VI in the `labview\examples\Graphics and Sound\3D Picture Control` directory for an example of configuring a scene camera and a scene window.

Displaying Data on a 3D Model

You can map real-world data onto a 3D model using free sensors and NI-DAQmx channels. Use the Sensor Mapping Express VI to wire an array of data values that represent sensors you configure in the **Configure Sensors** dialog box. For example, if you have a NI-DAQmx task that includes two channels and you also place two free sensors on a 3D model, LabVIEW expects the data values input as `daqsensor1`, `daqsensor2`, `freesensor1`, and `freesensor2`. LabVIEW allocates space for every channel in a NI-DAQmx task if you place at least one NI-DAQmx channel on the 3D model. Refer to the NI-DAQmx Help for more information about using NI-DAQmx channels.



Note You can create a NI-DAQmx task using the DAQ Assistant Express VI. However, the Sensor Mapping Express VI cannot retrieve the channels from the DAQ Assistant Express VI. You can wire the data to the **Sensor Values** input of the Sensor Mapping Express VI. Map the channels onto the 3D model by right-clicking the 3D model to place the sensors.

Wire a 3D picture control to the **Scene Ref Output** of the Express VI. The data you wire to the Express VI appears on the 3D model. The sensors you place on the 3D model affect the color of the 3D model at the sensor point according to the data value the sensor represents.

Using Meshes to Draw 3D Objects

The objects that appear in a **3D scene** are graphical representations of physical objects. A 3D object is a set of vertices that exist in 3D space and the connections made, or not made, between those vertices. The Geometries VIs such as Create Box and Create Sphere define the representation of simple geometric objects for you. If you want to create an object that is more geometrically complex, or if you want to define specific values for the **vertices**, **normals**, **indices**, **colors**, and **texture mapping** that define the appearance of a 3D object, you can configure a mesh.



NOTE

Using meshes is an advanced 3D graphics design technique. Prior experience with designing 3D graphics may help you use this feature.

A mesh is a collection of attributes that, when combined in a specific way, create a 3D surface. The attributes of a mesh include vertices, normals, colors, texture coordinates and the points, lines, triangles, or polygons that delineate or connect the vertices. When you define a mesh, you define arrays of values that represent the vertices, normals, colors, and texture coordinates of the mesh. You also define an array of **indices** that determines where

in the mesh to apply the values from each array. Finally, you define binding modes for the values in each of the arrays. To understand meshes, you must be familiar with the following terms:

- **Vertex** —A point in 3D space that creates a corner point of the mesh. An x, y, and z coordinate define each vertex.
- **Primitive** —An independent, visible face of a mesh. For example, if a mesh consists of three separate triangles, each triangle is a primitive of the mesh. Points, lines, triangles, quadrilaterals, and other polygons all can be primitives.
- **Normal** —A vector that points in a direction perpendicular to the surface of the mesh. You can define a normal for each vertex or for each polygon in a mesh. The normals of a mesh define the orientation of the surface of the mesh relative to light sources.

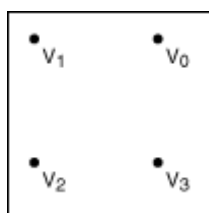
In LabVIEW, you can use the Create Mesh VI to create a reference to a new mesh. You then can use the SceneMesh properties and methods to configure the mesh programmatically. The following section provides additional information about the attributes of a mesh.

Draw Mode

The draw mode you select for a mesh specifies the way in which you want to draw or connect the vertices of the mesh. You can select from the following draw modes.

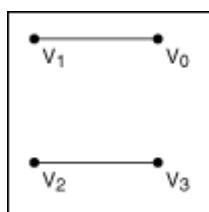
Points

Draws a point over each vertex. This mode does not connect the separate vertices, but the points appear in 3D space.



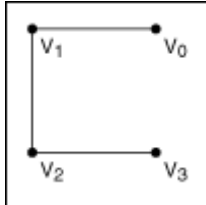
Lines

Draws non-contiguous line segments between pairs of vertices. For example, LabVIEW draws a mesh with four vertices with two lines: $V_0 V_1$ and $V_2 V_3$. If the total number of vertices is odd, LabVIEW ignores the final vertex.



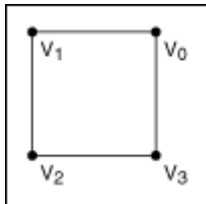
Line Strip

Draws contiguous line segments between the vertices, starting at V_0 and ending at V_{n-1} , where n is the total number of vertices. This mode connects all the vertices but does not create a closed polygon.



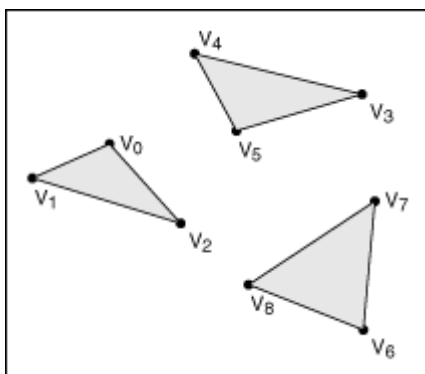
Line Loop

Draws contiguous line segments between the vertices, starting and ending at V_0 . This mode creates a closed polygon, but does not fill the space enclosed by the line segments.



Triangles

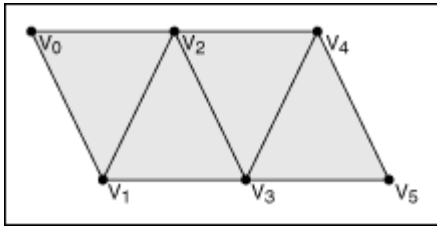
Draws a series of non-contiguous triangles that connect the vertices in groups of three, and fills the space enclosed by each triangle. For example, LabVIEW draws a mesh with six vertices with two triangles: $V_0 V_1 V_2$ and $V_3 V_4 V_5$. This mode does not connect the triangles to each other. If the total number of vertices is not an exact multiple of three, LabVIEW ignores the final one or two vertices.



Triangle Strip

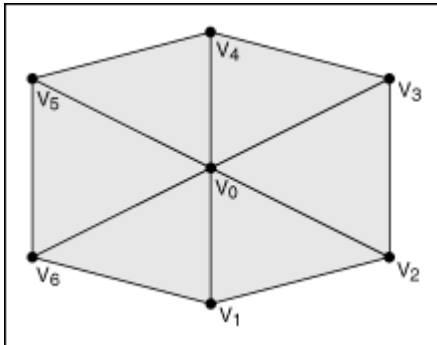
Draws a series of contiguous, filled triangles. LabVIEW connects the vertices so that the orientation of the triangles forms a solid strip, creating a portion of the object surface. For

example, the first triangle connects the vertices $V_0 V_1 V_2$, the second triangle connects the vertices $V_2 V_1 V_3$, the third triangle connects the vertices $V_2 V_3 V_4$, and so on.



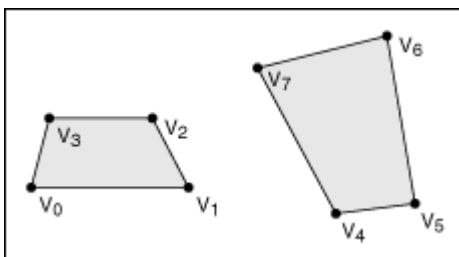
Triangle Fan

Draws a series of contiguous, filled triangles. The first vertex of each triangle is the same, and LabVIEW increments the second and third vertices by one for each subsequent triangle. This causes the triangles to create a fan around the central point of V_0 . For example, the first triangle connects $V_0 V_1 V_2$, the second triangle connects $V_0 V_2 V_3$, and so on.



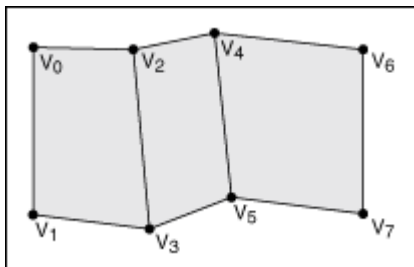
Quads

Draws a series of non-contiguous quadrilateral polygons that connect the vertices in groups of four, and fills the space enclosed by each quadrilateral. For example, LabVIEW draws a mesh with eight vertices with two quadrilaterals: $V_0 V_1 V_2 V_3$ and $V_4 V_5 V_6 V_7$. If the total number of vertices is not an exact multiple of four, LabVIEW ignores the final one, two, or three vertices.



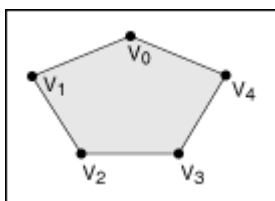
Quad Strip

Draws a series of contiguous quadrilateral polygons, and fills the space enclosed by each quadrilateral. For example, LabVIEW draws a mesh with eight vertices with three quadrilaterals: $V_0 V_1 V_3 V_2$, $V_2 V_3 V_5 V_4$, and $V_4 V_5 V_7 V_6$. If the total number of vertices is odd, LabVIEW ignores the last vertex.



Polygon

Draws a single, filled polygon to connect all the vertices, starting at V_0 and ending at V_{n-1} , where n is the total number of vertices. The total number of vertices must be at least three, or LabVIEW does not draw anything. Also, the polygon cannot intersect itself.



Vertex Array

The vertex array of a mesh defines the locations of each of the vertices in the mesh. Each element of the array represents one vertex in the mesh. LabVIEW indexes the array according to the values you specify in the [Indices](#) array. The vertex array is an array of clusters, where each cluster contains the x, y, and z values that define the position of one vertex in 3D space.

Indices

Indices is an array of integers that define how LabVIEW indexes the various arrays that create the mesh. For example, if the element at position 3 of the indices array is 0, then the elements at position 3 of the color array, [vertex array](#), [normal array](#), and [texture coordinate array](#) are all indexed to 0. Use the indices array to change the appearance of the mesh by changing the indices of the elements of four different arrays without changing the order of the elements in those arrays individually.

Color Binding Mode

The color binding mode you select for a mesh determines the way in which LabVIEW binds the elements of the **color array** to the mesh. You can select from the following color binding modes:

- **Overall** —Binds the color at the first index in the color array to all the primitives of the mesh.
- **Per Primitive** —Different colors bind to each primitive in the mesh. The **draw mode** of the mesh defines the primitives. For example, this method binds the color at the first index in the color array to the first primitive defined in the mesh, and so on.
- **Per Vertex** —Binds the colors in the color array to the vertices of the mesh according to the **indices** you define. If you select a draw mode that renders a solid polygon or polygons, the color at one vertex blends with the color at the next vertex as you move between the two.
- **Binding Off** —No colors bind to the mesh. LabVIEW ignores the color array.

Color Array

The color array of a mesh defines the RGBA color values to apply at each index of the indices array. The **color binding mode** you specify determines how LabVIEW applies the colors in this array. The color array is an array of clusters, where each cluster contains the red, green, blue, and alpha values that define one color.

Normal Binding Mode

The normal binding mode you select determines how LabVIEW binds the elements of the **normal array** to the mesh. The normal binding mode determines how light reflects off the surface of the mesh. You can select from the following normal binding modes:

- **Overall** —Binds the normal at the first index of the normal array to all the primitives of the mesh.
- **Per Primitive** —Different normals bind to each primitive in the mesh. The Draw Mode of the mesh defines the primitives. For example, this method binds the normal at the first index of the normal array to the first primitive defined in the mesh, and so on.
- **Per Vertex** —Binds the normals in the normal array to the vertices of the mesh according to the **indices** you define. You must define a normal for every vertex in the mesh.
- **Binding Off** —Turns off normal binding. LabVIEW ignores the normal array.

Normal Array

The normal array of a mesh defines the origin of each normal in the mesh. Each element of the array represents one normal in the mesh. The **normal binding mode** you specify determines how LabVIEW applies the normals in this array. The normal array is an array of

clusters, where each cluster contains the x, y, and z values that define the origin of one normal.

Texture Coordinate Array

The texture coordinate array of a mesh defines how a *texture* maps to the surface of the mesh. Each element of the array represents one texture coordinate. You must define a texture coordinate for each vertex of the mesh. The texture coordinate array is an array of clusters, where each cluster contains the s and t values that define one coordinate in the texture. If you do not enable texturing, LabVIEW ignores this array. Use the SceneObject property Specials:Texturing to enable texturing on an object in a 3D scene.

Refer to the Using Meshes VI in the `labview\examples\Graphics and Sound\3D Picture Control` directory for examples of how changing the attributes of a mesh affects the appearance of the mesh.

File I/O Operations in LabVIEW

File I/O

File I/O operations pass data to and from files. Use the File I/O VIs and functions on the File I/O palette to handle all aspects of file I/O, including the following:

- Opening and closing data files.
- Reading data from and writing data to files.
- Reading from and writing to spreadsheet-formatted files.
- Moving and renaming files and directories.
- Changing file characteristics.
- Creating, modifying, and reading a configuration file.

You can open, read or write, and close a file using a single VI or function. You also can use a function to control each step in the process.

Refer to the `labview\examples\File IO` directory for examples of file I/O operations.

Find related examples

Basics of File I/O

A typical file I/O operation involves the following process.

1. Create or open a file. After the file opens, a unique identifier called a refnum represents the file.
2. Use a VI or function from the File I/O palette to read from or write to the file.
3. Close the file.

File I/O VIs and some File I/O functions, such as the Read from Text File and Write to Text File functions, can perform all three steps for typical file I/O operations. The VIs and functions designed for multiple operations might not be as efficient as the functions that perform individual operations.



RELATED INFORMATION

[Refnum Controls and Indicators](#) on page 160

[Saving Memory using Disk Streaming](#) on page 524

[Improving Efficiency by Disabling File Buffering for Binary File I/O Functions](#) on page 525

[Determining Which File Format to Use](#) on page 526

[File Paths](#) on page 527

Extending Data File Support using DataPlugins (Windows)

DataPlugins provide a unified method to access data with LabVIEW, regardless of the file format. A DataPlugin is encapsulated code that reads and interprets the data contained in a specific file format by mapping the data to the TDM data model. The TDM data model arranges the data in three hierarchical levels: file, channel groups, and channels.

Downloading and Registering DataPlugins

National Instruments provides DataPlugins for some data formats. You can view and download these DataPlugins from the National Instruments website at ni.com/dataplugins.

After you download a DataPlugin, you must register it on the local computer before you can access the corresponding data file. Double-click the DataPlugin file to manually register the DataPlugin. You also can use the Register DataPlugins VI to programmatically register the DataPlugin.

Creating DataPlugins

If you cannot find the DataPlugin for a specific data format, you can create one by using the LabVIEW DataPlugin SDK. The SDK also enables you to create an installer in order to install the new DataPlugin on any Windows computer. Refer to the LabVIEW DataPlugin SDK User Manual for the following information:

- LabVIEW DataPlugin Types
- Creating DataPlugins with LabVIEW
- Channel Length Limitation
- Supported Data Types and Conversions
- Name Conventions
- Supported LabVIEW Versions

Refer to the Read ATML File VI in the `labview\examples\File IO\DataPlugins\ATML\DataPlugins - ATML.lvproj` file for an example of using a DataPlugin to read information from an ATML 2.0 file.

Saving Memory using Disk Streaming

Use disk streaming to save memory resources by reducing the number of times a function interacts with the operating system to open and close a file. Disk streaming refers to keeping files open while you perform multiple read or write operations within a loop.

Avoid wiring a path control or a constant to a function or VI that reads from or writes to a file, such as the Write to Text File function or the Read from Binary File function, as doing so adds the overhead of opening and closing the file each time the function or VI executes. Instead, implement disk streaming operations to eliminate this overhead.

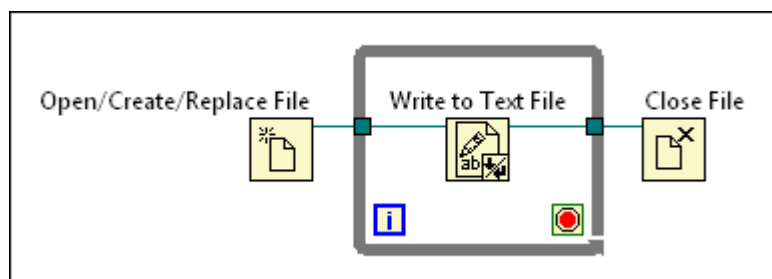
When to Use Disk Streaming

Consider the following information when determining whether a disk streaming operation is appropriate for your application:

- Disk streaming is ideal in lengthy data acquisition operations where speed is critical.
- Use disk streaming to write data continuously to a file while acquisition is still in progress.

Implementing a Disk Streaming Operation

To create a typical disk streaming operation, place the Open/Create/Replace File function and the Close File function outside the loop as in the following block diagram.



In a disk streaming design pattern, a VI continuously writes to a file within the loop without the overhead associated with opening and closing the file in each iteration.



TIP

For best results, avoid running other VIs and functions, such as analysis VIs and functions, until you complete the acquisition.

Improving Efficiency by Disabling File Buffering for Binary File I/O Functions

During each file I/O operation, LabVIEW takes a few milliseconds to call the operating system (OS) and request a data transfer to and from a file on a disk. These milliseconds accumulate every time LabVIEW calls the OS. To avoid these timely repetitions, most OS file systems provide a buffer to temporarily hold each piece of data waiting to be read or written to a file. When the buffer is full, the OS performs a single file I/O operation.

The process described in the previous paragraph is known as buffering, and LabVIEW enables it by default. Buffering reduces the number of times the OS must access the disk and expend processing time.

When to Disable Buffering

In some cases, you can achieve faster data streaming rates if you disable buffering. For example, you may use a Redundant Array of Independent Disks (RAID) to increase processing speed. A RAID is a set of hard disks acting as a single disk that the OS can simultaneously access, taking less time to read or write data. If you access a RAID with buffering enabled, LabVIEW can take more time to copy data to the OS than the OS takes to actually write the data to the disk. You can disable buffering with the **disable buffering** input of the Open/Create/Replace File function to avoid these data copies and force the OS to send data directly to disk.

However, if you disable buffering, make sure the following conditions are true:

- **You must make the size of the data in the file a multiple of the sector size, indicated in bytes, of the disk that contains or may eventually contain the file.** A sector is a subdivision of disk space that stores a fixed amount of data, typically 512 bytes. Use the **sector size (bytes)** output of the Get Volume Info function to determine the sector size of the disk. For LabVIEW to save data to a disk, the data can span multiple sectors but must fill each sector completely. A 512-byte sector requires 512 bytes of data. If the data is not a multiple of the sector size, you must pad the data with filler data and delete the filler data before LabVIEW reads back the file.
- **The data in the file must be aligned to a multiple of the alignment the disk requires.** LabVIEW aligns the data, and you cannot change that alignment. If the data does not meet the alignment requirement, LabVIEW returns an error and you must enable buffering and reopen the file.

Related Information

Buffering Data

Determining Which File Format to Use

The VIs on the File I/O palette you use depend on the format of the files. You can read data from or write data to files in the following formats:

- **Text** —If you want to make your data available to other applications, such as Microsoft Notepad, use text files because they are the most common and the most portable. In LabVIEW, the following types of files belong to text files:
 - Configuration files
 - Excel files



NOTE

Although you cannot open an Excel file .xlsx directly in a text editor, the file is a zipped file consisting of multiple XML files. Each XML file is in text file format.

 - LabVIEW measurement files
 - Spreadsheet files
 - XML files
- **Binary** —If you need to perform random access file reads or writes or if speed and compact disk space are crucial, use binary files because they are more efficient than text files in disk space and in speed. In LabVIEW, the following types of files belong to binary files:
 - TDM/TDMS files
 - Waveforms
 - Zip files
- **Datalog** —If you want to manipulate complex records of data or different data types in LabVIEW, use datalog files because they are the best way to store data if you intend to access the data only from LabVIEW and you need to store complex data structures.

(Windows) If you want to use LabVIEW to access data in a file format other than those listed above, you can use DataPlugins. DataPlugins provide a unified method to access data with LabVIEW, regardless of the file format.



RELATED INFORMATION

[Text Files](#) on page 535

[Configuration Files](#) on page 529

[Spreadsheet Files](#) on page 532

[LabVIEW Measurement Files](#) on page 878

[Spreadsheet Files](#) on page 532

[XML Files](#) on page 537

[Binary Files](#) on page 528

[TDM/TDMS Files](#) on page 883

[Waveforms](#) on page 560

[Zip Files](#) on page 543

[Datalog Files](#) on page 530

[Extending Data File Support using DataPlugins \(Windows\)](#) on page 523

File Paths

A [file path control](#), shown as follows, is a LabVIEW data type that identifies the location of a file on disk.



A file path describes the volume that contains the file, the directories between the top-level of the file system and the file, and the name of the file. Enter or display a path using the standard syntax for a given platform with the path control or indicator.

The following table includes information about the different types of file paths you can use in LabVIEW.

Type	Description	Usage
absolute path	An absolute path describes the location of a file or directory starting from the top level of the file system. For example, E:\Temp\test.txt.	Use an absolute path to point to the same location in a file system, regardless of the current working directory.

Type	Description	Usage
relative path	A relative path describes the location of a file or directory relative to an arbitrary location in the file system. For example, ..\test.txt.	Use a relative path to point to a location that might change with the current working directory. Use relative paths in VIs to avoid having to rework the paths if you build an application or run the VI on a different computer.
symbolic path or pseudopath	A symbolic path or pseudopath is a predefined short form of a path that you can use in LabVIEW. For example, the <vilib> symbolic path represents the <vi.lib> directory in the labview directory. At run time, LabVIEW interprets the <vilib> symbolic path as an absolute path, such as C:\Program Files (x86)\National Instruments\LabVIEW 2009\vi.lib or D:\Program Files\National Instruments\LabVIEW 2009\vi.lib, depending on where LabVIEW is installed.	Use a symbolic path to point to a location that changes with the execution environment, such as where on disk LabVIEW is installed or which user is logged in.

Binary Files

Storing binary data, such as an integer, uses a fixed number of bytes on disk. For example, storing any number from 0 to 4 billion in binary format, such as 1, 1,000, or 1,000,000, takes up 4 bytes for each number.

Use binary files to save numeric data and to access specific numbers from a file or randomly access numbers from a file. Binary files are machine readable only, unlike text files, which are human readable. Binary files are the most compact and fastest format for storing data. You can use multiple data types in binary files, but it is uncommon.

Binary files are more efficient because they use less disk space and because you do not need to convert data to and from a text representation when you store and retrieve data. A binary file can represent 256 values in 1 byte of disk space. Often, binary files contain a byte-for-byte image of the data as it was *stored in memory*, except for cases like extended and complex numeric values. When the file contains a byte-for-byte image of the data as it was stored in memory, reading the file is faster because conversion is not necessary.



NOTE
Text and binary files are both known as byte stream files, which means they store data as a sequence of characters or bytes.

Use the File I/O VIs and functions to *read from* and *write to* binary files. Consider using the binary file functions if you want to read numeric data from or write numeric data to a file or if you want to create text files for use on multiple operating systems.

Refer to `Simple Binary File.lvproj` in the `labview\examples\File IO\Binary\Simple Binary` directory for examples of reading and writing an array of double-precision floating-point values from and to a binary file, respectively.

Related Information

Creating Binary Files

Reading from Binary Files

Writing to Binary Files

Creating Binary Files

Use the Write to Binary File function to create a binary file. You can wire any data type to the **data** input of the Write to Binary File function. Use the Read from Binary File function to specify the data type of the data in the file you want to read by wiring a control or constant of that type to the **data type** input of the Read from Binary File function. You can use the Write to Binary File and Read from Binary File functions to read from and write to text files created on a different operating system.

Refer to `Simple Binary File.lvproj` in the `labview\examples\File IO\Binary\Simple Binary` directory for examples of using the Read from Binary File function and the Write to Binary File function.

Configuration Files

Use the Configuration File VIs to read and create Windows configuration settings `.ini` files and to write platform-specific data, such as paths, in a platform-independent format so that you can use the files these VIs generate across multiple platforms. The Configuration File VIs use the *configuration settings file format*. While you can use the Configuration File VIs on any platform to read and write files created by the VIs, you cannot use the Configuration File VIs to create or modify configuration files in an macOS or Linux format.



NOTE

You can use the Configuration File VIs for Windows configuration settings files only in the ANSI format.

Refer to the `labview\examples\File IO\Configuration (INI)\Configuration (INI) File.lvproj` for examples of using the Configuration File VIs.



Note The standard extension for *Windows configuration settings* files is `.ini`, but the Configuration File VIs work with files with any extension, provided the content is in the correct format.

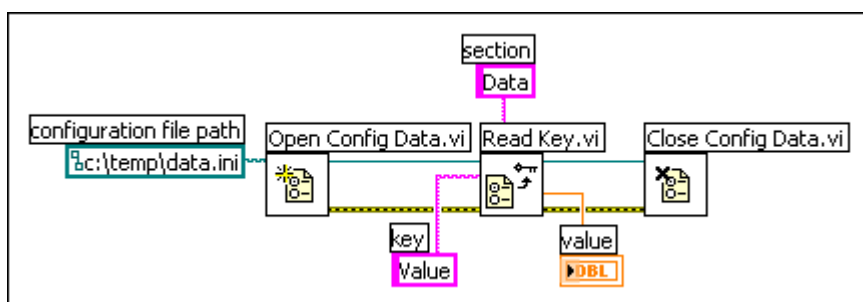
A standard Windows configuration settings file is a specific format for storing data in a text file. You can programmatically access data within the `.ini` file easily because it follows a specific format.

For example, consider a configuration settings file with the following contents:

```
[Data]
```

```
Value=7.2
```

You can use the Configuration File VIs to read this data, as shown in the following block diagram. This VI uses the Read Key VI to read the **key** named Value from the **section** called Data. This VI works regardless of how the file changes, provided the file remains in the Windows configuration settings file format.



Related Information

[Windows Configuration Settings File Format](#)

[Using Configuration File VIs](#)

Datalog Files

Use datalog files to access and manipulate data only in LabVIEW and to store complex data structures quickly and easily.

A datalog file stores data as a sequence of identically structured records, similar to a spreadsheet, where each row represents a record. Each record in a datalog file must have the same data types associated with it. LabVIEW writes each record to the file as a cluster containing the data to store. However, the components of a datalog record can be any data type, which you determine when you create the file.

For example, you can create a datalog whose record data type is a cluster of a string and a number. Then, each record of the datalog is a cluster of a string and a number. However, the first record could be ("abc",1), while the second record could be ("xyz",7).

You may eventually decide to change the record data type of your datalog file. In doing so, any VIs you created to manipulate those records will have to be updated to read the new data type. However, once you update your VIs, they cannot read any files created with the old record data type.

Using datalog files requires little manipulation, which makes writing and reading much faster. It also simplifies data retrieval because you can read the original blocks of data back as a record without having to read all records that precede it in the file. Random access is fast and easy with datalog files because all you need to access the record is the record number. LabVIEW sequentially assigns the record number to each record when it creates the datalog file.

You can [access datalog files](#) from the front panel and from the block diagram.

LabVIEW writes a record to a datalog file each time the associated VI runs. You cannot overwrite a record after LabVIEW writes it to a datalog file. When you read a datalog file, you can read one or more records at a time.

Another way to create a datalog file is by using [front panel data logging](#), which records data for use in other VIs and in reports.

Refer to the `labview\examples\File IO\Datalog\Datalog File.lvproj` for examples of reading and writing datalog files.

Related Information

[Creating Datalog Files](#)

[Reading from Datalog Files](#)

[Writing to Datalog Files](#)

[Logging Front Panel Data](#)

Creating Datalog Files

You can create and read datalog files by enabling [front panel data logging](#) or by using the Datalog functions to acquire data and write the data to a file.

You do not have to format the data in a datalog file. However, when you [write](#) or [read](#) datalog files, you must specify the data type. For example, if you acquire a temperature reading with the time and date the temperature was recorded, you write the data to a datalog file and specify the data as a cluster of one number and two strings.

If you read a file that includes a temperature reading with the time and date the temperature was recorded, you specify that you want to read a cluster of one number and two strings.

Datalog Type

The data type of each record in the datalog file can be of any type. However, the data type varies depending on how you logged the data to the datalog file.

For a datalog file you [created using the Write Datalog function](#), the data type matches the data LabVIEW wrote to the datalog file.

For a datalog file you [created using front panel data logging](#), the data type is a cluster of two clusters. The first cluster contains a time stamp, and the second cluster contains the front panel data. The time stamp includes a 32-bit unsigned integer that represents seconds and a 16-bit unsigned integer that represents milliseconds elapsed from LabVIEW system time. The front panel data cluster contains the data types that correspond to all front panel controls and indicators in their respective tabbing order.

Spreadsheet Files

Spreadsheet files organize 1D or 2D arrays of data in tabular form, or in rows and columns. Each cell of the table contains an element of the 1D or 2D array.

Text-Based Spreadsheet Files

Text-based spreadsheet files are a subset of text files. To write data to a spreadsheet file, you must format the data as a spreadsheet string, which is a string that includes delimiters, such as tabs or commas. After you read data from a spreadsheet file, the data is a set of spreadsheet strings. You need to format the strings as the data type you expect from this spreadsheet file. The File I/O VIs and functions support text-based spreadsheet files with delimited string.

Binary-Based Spreadsheet Files

Binary-based spreadsheet files contain binary data rather than text. Many binary spreadsheet file formats exist. To read binary data from a third-party spreadsheet file in LabVIEW, use DataPlugins.

To save dynamic data you acquire to a Microsoft Excel file `.xlsx`, you can use the Write To Measurement File Express VI. This Express VI uses the Office Open XML file format, which conforms to the international standard ISO/IEC 29500:2008, to create the Excel file. Refer to

the International Organization for Standardization website at www.iso.org for information about the ISO/IEC 29500:2008 standard.

**NOTE**

If your application uses the Excel file format, you cannot save the application for a previous version of LabVIEW. You must modify the application to use another file format before saving it for a previous version.

To write data to an Excel file, you do not need to install Microsoft Excel. To view the data in the Excel file, you must use a spreadsheet viewer that can read the Office Open XML file format.

Each spreadsheet viewer has a maximum value for the number of rows and columns of data you can view. Errors might occur if the amount of data you write to an Excel file exceeds the maximum value. Refer to the documentation for each spreadsheet viewer for information about the maximum value you can view in an Excel file.

After you write data to an Excel file, you can import the Excel file to other National Instruments software applications, such as DIAdem 2010 or later. Refer to the documentation for DIAdem for information about importing Excel files.

Refer to the Write Excel File VI in the `labview\examples\File IO\Spreadsheet\Excel` folder for an example of using the Write To Measurement File Express VI to write data to an Excel file.

Related Information

[DataPlugins](#)

[Creating Spreadsheet Files](#)

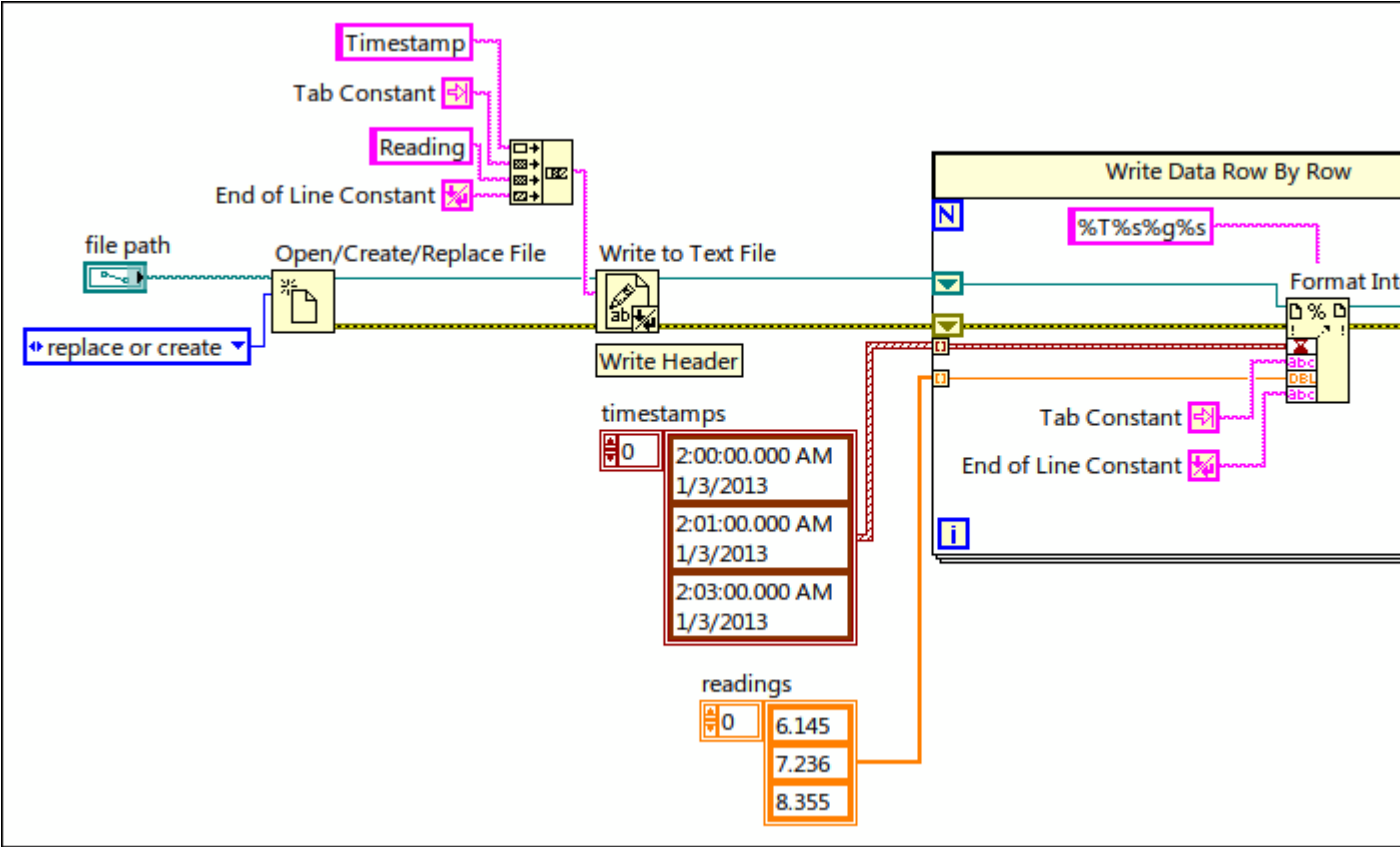
[Reading from Spreadsheet Files](#)

[Writing to Spreadsheet Files](#)

Creating Spreadsheet Files

Text-based spreadsheet files are a subset of text files. To write data to a spreadsheet file, you must format string data as a spreadsheet string, which is a string that includes delimiters, such as tabs.

The following VI writes different data types to a text file. The Write to Text File function writes a header containing string data and delimiters. The Format Into File function formats timestamp data, numeric data, and delimiters into a spreadsheet string and writes the string to the file.



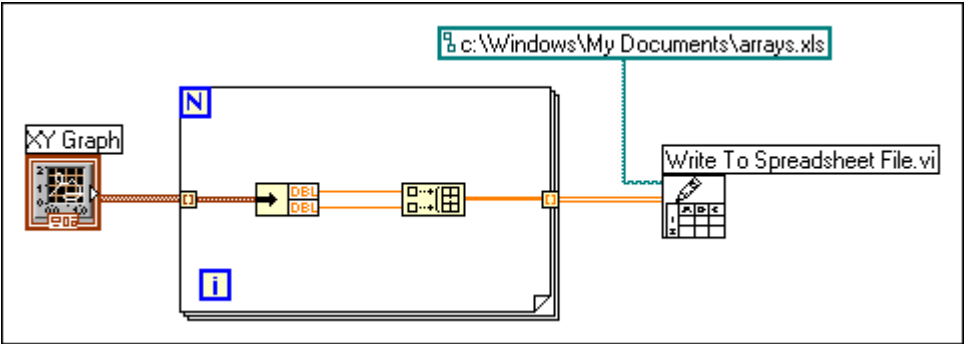
Use the Write Delimited Spreadsheet VI or the Array To Spreadsheet String function to convert a set of numbers from a graph, a chart, or an acquisition into a spreadsheet string.



NOTE

For Microsoft Excel files (.xlsx), use the Write to Measurement File Express VI to save the dynamic data you acquire to an Excel file.

The following VI acquires data from a graph and writes it to a spreadsheet file.



Reading text from a word processing application might result in errors because word processing applications format text with different fonts, colors, styles, and sizes that the File I/O VIs cannot process.

If you want to write numbers and text to a spreadsheet or word processing application, use the String functions and the Array functions to format the data and to combine the strings. Then write the data to a file.



RELATED INFORMATION

[Text Files](#) on page 535

[Grouping Data with Strings](#) on page 456

[Spreadsheet Files](#) on page 532

[Grouping Data with Strings](#) on page 456

[Grouping Data with Arrays and Clusters](#) on page 460

[Arrays](#) on page 460

[Clusters](#) on page 465

Text Files

Use text format files for your data to make it available to other users or applications, if disk space and file I/O speed are not crucial, if you do not need to perform random access reads or writes, and if numeric precision is not important.

Text files are the easiest format to use and to share. Almost any computer can read from or write to a text file. A variety of text-based programs can read text-based files. Most instrument control applications use text strings.

Store data in text files when you want to access it from another application, such as a word processing or spreadsheet application. To store data in text format, use the String functions to convert all data to text strings. Text files can contain information of different data types.

Text files typically take up more memory than binary and datalog files if the data is not originally in text form, such as graph or chart data, because the ASCII representation of data usually is larger than the data itself. For example, you can store the number -123.4567 in 4 bytes as a single-precision floating-point number. However, its ASCII representation takes 9 bytes, one for each character.

In addition, it is difficult to randomly access numeric data in text files. Although each character in a string takes up exactly 1 byte of space, the space required to express a number as text typically is not fixed. To find the ninth number in a text file, LabVIEW must first read and convert the preceding eight numbers.

You might lose precision if you store numeric data in text files. Computers store numeric data as binary data, and typically you write numeric data to a text file in decimal notation. A loss of precision might occur when you write the data to the text file. Loss of precision is not an issue with binary files.

Use the File I/O VIs and functions to [read from](#) or [write to](#) text files and to [read from](#) or [write to](#) spreadsheet files.

Refer to the following VIs for examples of using file I/O with text files:

- labview\examples\File IO\Text (ASCII)\Text (ASCII) Files.lvproj
- labview\examples\File IO\Spreadsheet\Tab-Delimited Data\Tab-Delimited Data.lvproj

Related Information

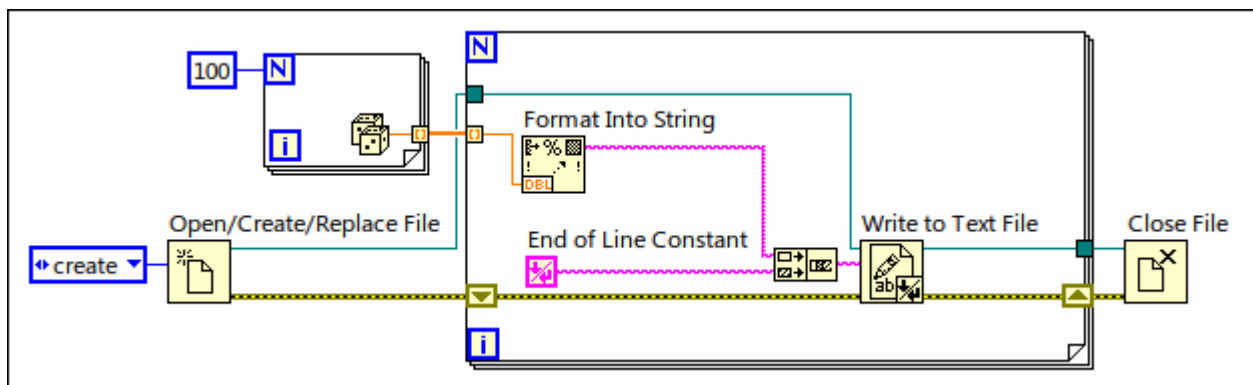
[Creating Text Files](#)

[Reading from Text Files](#)

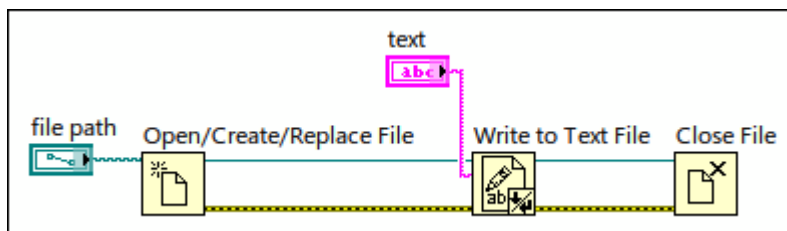
[Writing to Text Files](#)

Creating Text Files

To write data to a text file, you must convert your data to a string. Use the String functions to convert your data to a string. The following VI converts an array of randomly generated numbers to strings and writes the strings to a text file.



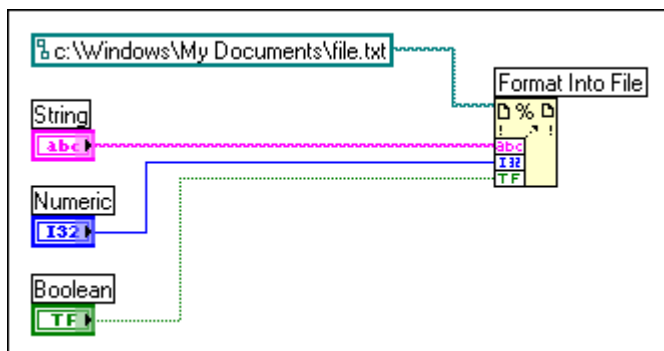
Writing text to text files requires no formatting because most word processing applications that read text do not require formatted text. To write a text string to a text file, use the Write to Text File function, which automatically opens and closes the file as the following VI demonstrates.



Formatting and Writing Data to Files

Use the Format Into File function to format string, numeric, path, and Boolean data as text and to write the formatted text to a file.

The following VI acquires multiple data types and writes them to a file.



Often you can use this function instead of separately formatting the string with the Format Into String function and writing the resulting string with the Write to Text File function.

Scanning Data from Files

Use the Scan From File function to scan text in a file for string, numeric, path, and Boolean values and then convert the text into a data type. Often you can use this function instead of reading data from a file with the Read from Binary File function or Read from Text File function and scanning the resulting string with the Scan From String function.



RELATED INFORMATION

[Grouping Data with Strings](#) on page 456

XML Files

Extensible Markup Language (XML) is a formatting standard that uses tags to describe data. Unlike an HTML tag, an XML tag does not tell a browser how to format a piece of data. Instead, an XML tag identifies a piece of data.

For example, suppose you are a bookseller who sells books on the Web. You want to classify each book in your library by the following criteria:

- Type of book (fiction or nonfiction)
- Title
- Author
- Publisher
- Price
- Genre(s)

- Synopsis
- Number of pages

You can create an XML file for each book. The XML file for a book titled Touring Germany's Great Cathedrals would be similar to the following:

```
<nonfiction>

<Title>Touring Germany's Great Cathedrals<Title>

<Author>Tony Walters<Author>

<Publisher>Douglas Drive Publishing<Publisher>

<PriceUS>$29.99<PriceUS>

<Genre>Travel<Genre>

<Genre>Architecture<Genre>

<Genre>History<Genre>

<Synopsis>This book fully illustrates twelve of Germany's
most inspiring cathedrals with full-color photographs, scaled
cross-sections, and time lines of their construction.<
Synopsis>

<Pages>224<Pages>

</nonfiction>
```

Similarly, you can classify LabVIEW data by name, value, and type. You can represent a string control for a user name in XML as follows:

```
<String>

<Name>User Name<Name>

<Value>Reggie Harmon<Value>

</String>
```

Related Information

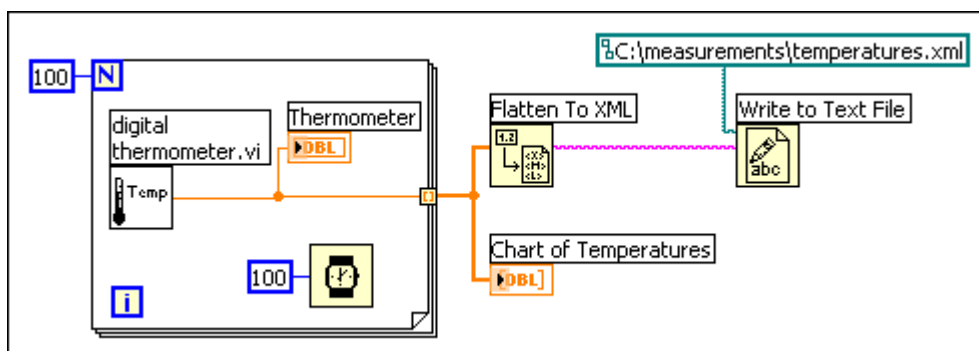
[Converting Data to and from XML](#)

[Parsing XML Files in LabVIEW](#)

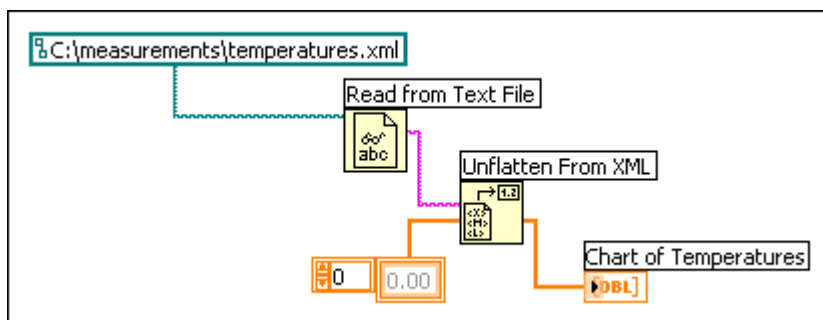
Converting Data to and from XML

Converting LabVIEW data to XML formats the data so that when you save the data to a file, you easily can identify the value(s), name(s), and type of the data from the tags that describe the data. For example, if you convert an array of temperature values to XML and save that data to a text file, you easily can identify the temperature values by locating the `<Value>` tag that identifies each temperature.

Use the Flatten To XML function to convert a LabVIEW data type to XML format. The following block diagram generates 100 simulated temperatures, plots the array of temperatures to a chart, converts the array of numbers to the XML format, and writes the XML data to the `temperatures.xml` file.



Use the Unflatten From XML function to convert a data type in the XML format into LabVIEW data. The following block diagram reads the 100 temperatures in the `temperatures.xml` file and plots the array of temperatures to a chart.



NOTE

Although you can flatten and unflatten LabVIEW variant data to and from XML, LabVIEW returns an error if the variant data is an element of a cluster or is an array of variants.

Refer to the `labview\examples\File IO\XML\Flatten and Unflatten XML\Flatten and Unflatten XML.lvproj` for examples of converting to and from the XML format.

LabVIEW XML Schema

LabVIEW converts data to an established XML schema. Currently, you cannot create customized schemas, and you cannot control how LabVIEW tags each piece of data. Also, you cannot convert entire VIs or functions to XML.

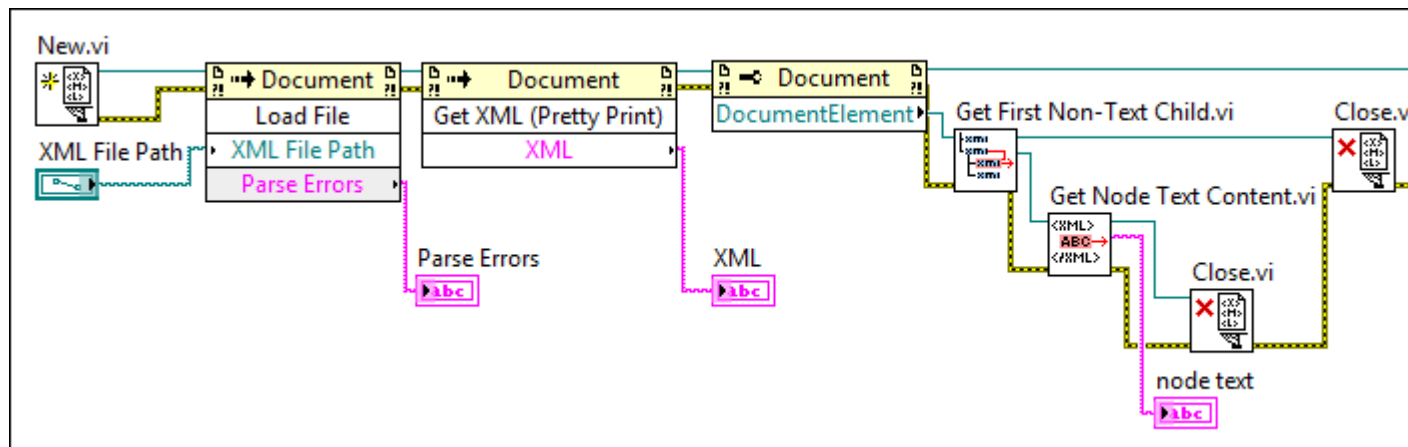
The predefined XML schema that LabVIEW uses is `LVXMLSchema.xsd` located in the `labview\vi.lib\Utility` directory. You can open the file in a text editor to read the schema.

Parsing XML Files in LabVIEW

Extensible Markup Language (XML) is a platform-independent subset of Standard Generalized Markup Language (SGML) that you can use to store and exchange information. When you work with XML documents, you can use a parser to extract and manipulate data without translating the XML format directly. For example, the Document Object Model (DOM) Core specification defines a programming interface for creating, reading, and manipulating XML documents. The DOM Core specification defines the properties and methods an XML parser must support. The XML parser in LabVIEW is the Xerces 2.7 parser, which uses a document object model (DOM).

Use the XML parser and the XML Parser VIs and nodes to read, write, or manipulate XML documents.

The XML Parser VIs allow you to access a cross-platform XML parser in LabVIEW, as shown in the following block diagram.



Refer to the `labview\examples\File IO\XML\Parse XML` directory for examples using XML Parser VIs.

Find related examples

The New VI opens an XML parser session and returns an XML Document reference. You can create an XML document, load an existing document into the XML parser, or configure document settings.

To manipulate an existing file, use the Load File method to load the XML document into memory. LabVIEW can access the XML document unless the XML parser encounters errors within the document that prevent the XML parser from parsing the document. The **Parse Errors** output displays these XML-related errors.

Use the Get XML (Pretty Print) method to pretty print the contents of the XML document to a string. Pretty printing makes XML data easier to read by adding white space and end-of-line (EOL) characters to the XML data. Use the Get XML method if you do not need to pretty print XML data. In the previous block diagram, the **XML** output displays the contents of the XML document.

The Close VI closes the XML Document reference and returns any errors the VI encounters.



NOTE

The XML parser ignores white spaces only when it is performing validation. Otherwise, the parser includes white spaces as child nodes.

Validating XML Documents

You can configure the XML parser to determine if a specific XML document is valid. A document is valid if the document complies with an external vocabulary. In the LabVIEW parser, the external vocabulary can be a Document Type Definition (DTD) or a schema.

Some parsers only parse the XML and do not validate the XML prior to loading the file. The parser in LabVIEW is a validating parser. A validating parser validates an XML document against a DTD or schema and reports invalid items it finds. You must ensure that a specific document is the form and type that you expect. Using a validating parser eliminates the need to create custom validation code for each type of document.

The XML parser reports validation errors in the **Parse Errors** output of the Load File method.



Note The XML parser validates the document or XML string when LabVIEW loads the document or string. If you make changes to the document or string and want to validate the document or string with the changes, use the Load File or Load String methods to reload the document or string. The parser then validates the content again.

Refer to the Load XML with Parser API VI in the `labview\examples\File IO\XML\Parse XML` directory for an example of using the Load File method.

Configuring the XML Parser

You must configure all instances of the XML parser if you want to activate all possible features. You can configure the parser by setting properties on the Document class.

Place a Property Node (XML) on the block diagram and examine the available properties. The Property Node (XML) includes the following properties and methods:

- **Process Namespaces** —Allows you to enable or disable the XML parser namespace processing. The default is TRUE, which causes the XML parser to enforce its namespace specification constraints and rules.
- **Process Schema** —Allows you to enable or disable XML parser schema processing. The default is FALSE, which means the XML parser does not process any schema. If you set the property to TRUE, you also must enable namespace processing.
- **Load External DTD** —Allows you to enable or disable the loading of external DTDs. The default is TRUE, which means the XML parser allows you to load external DTDs. If you set the Validate On Load property to Always or Auto, the parser ignores this property and always loads the DTD.
- **Preserve Whitespace** —Allows you to specify if a validating parser includes ignorable white spaces as text nodes. The default is TRUE, which adds ignorable white spaces to the DOM tree as text nodes. If you set the property to FALSE, the XML parser discards all ignorable white space and does not add text nodes to the DOM tree.

Note The XML parser ignores white spaces only when it is performing validation. Otherwise, the parser includes white spaces as child nodes. For example, the first child of a tag is likely to be white space rather than the next element.

- **Schema Full Checking** —Allows you to set full schema constraint checking. The property takes effect only if you set the Validate on Load property to Always or Auto. The default is FALSE, which runs partial constraint checking. Full schema constraint checking can be time consuming or memory intensive.
- **Validate On Load** —Allows you to set the validation scheme that the XML parser uses. You can select one of the following enumerated values:
 - **Auto** —(Default) Turns on validation if the parser detects any internal or external DTD subset.
 - **Never** —Turns off validation.

Note If you set this property to Never and set the Load External DTD property to TRUE, LabVIEW parses the document but does not return validation errors.

- **Always** —Turns on validation.

Refer to the `labview\examples\File IO\XML\Parse XML` directory for examples using XML Parser VIs.

Find related examples

Querying XML Documents

XML Path Language (XPath) is a language for locating XML nodes, such as elements, attributes, text, and so on. Use the Get First Matched Node VI to locate the first node that matches the XPath expression you specify. Use the Get All Matched Nodes VI to locate all nodes that match the XPath expression you specify. These VIs use the context node you specify to locate nodes. The context node describes the location in an XML document from which LabVIEW performs an XPath query. Therefore, you can specify a relative or absolute XPath expression to query the XML document.

LabVIEW supports XPath version 1.0, which is defined by the World Wide Web Consortium (W3C).

Refer to the Parse XML String for Multiple Nodes VI in the `labview\examples\File IO\XML\Parse XML` directory for an example of using the Get All Matched Nodes VI.

Zip Files

The zip file format enables you to compress and archive data. In LabVIEW, you can programmatically compress one or more files into a zip file while maintaining the directory structure.



NOTE

Zip files are a subset of archive files, which include other file formats, such as RAR, TAR, 7Z, GZ, and so on. LabVIEW supports the zip file format only.

Refer to the `labview\examples\File IO\Zip\Zip Files.lvproj` for an example of using the Zip VIs to create a zip file.

Windows Configuration Settings File Format

Windows configuration settings files are text files divided into named sections. Brackets enclose each section name. Every section name in a file must be unique. The sections contain key/value pairs separated by an equal sign (=). Within each section, every key name must be unique. The key name represents a configuration preference, and the value name represents the setting for that preference. The following example shows the arrangement of the file:

```
[Section 1]
```

```
key1=value
```

```
key2=value
```

```
[Section 2]
```

```
key1=value
```

```
key2=value
```

Use the following data types with Configuration File VIs for the value portion of the **key** parameter:

- String
- Path

- Boolean
- 64-bit double-precision floating-point numeric
- 32-bit signed integer
- 32-bit unsigned integer

**NOTE**

LabVIEW writes Boolean values as TRUE and FALSE by default, but these values are not case sensitive. For example, Fa1se, fa1se, and 0 are all acceptable FALSE Boolean values.

To use the Configuration File VIs, you must format each line of text in an `.ini` file correctly. A line must either be empty or contain a section name, a key-value pair, or a comment.



Note You can use the Configuration File VIs for Windows configuration settings files only in the ANSI format.

Use the following guidelines to format lines with section names:

- Include at least one character
- Do not use a closed bracket as part of the name
- Do not include unprintable characters
- Use an open bracket for the first non-white space character
- Use a closed bracket at the end of the line of text
- Keep all the characters on one line

Use the following guidelines to format lines with key-value pairs:

- Include pairs on the lines after the section line
- Include an equal sign after the key name
- Keep all the characters on one line
- Use valid key names
 - Include at least one character
 - Do not start with a semicolon
 - Do not include an equal sign
 - Do not start with an open bracket
 - Do not start with a white space character
 - Do not end with a white space character
 - Do not include unprintable characters
- Use valid value names
 - Keep the data type consistent

- Do not start with a white space character
- Do not end with a white space character



Note If values begin or end with white space characters, you must put the values in quotation marks. LabVIEW removes the quotation marks when a VI returns the value.

Configuration File VIs ignore lines without section names or value-key pairs during a read operation, and they preserve lines without section names or value-key pairs during a write operation.

When VIs write to a configuration file, they place quotation marks around any string or path data. LabVIEW also supports single quotes around values in `.ini` files.

The accepted comment notation for configuration files is a semicolon, which can indicate that the text following the semicolon is a comment. A semicolon at the beginning of a line indicates a comment. A semicolon in a section name, numeric value, or Boolean value also indicates a comment. However, a semicolon in a string value name does not indicate a comment.

If you include a semicolon as the first character of a key name, LabVIEW does not return a key name or value because the configuration files do not support a semicolon as the first character of a key name. If you include a semicolon at any other place in the key name besides as the first character, LabVIEW returns the key name with the semicolon.

The following table provides examples of how the Configuration File VIs read section names in `.ini` files.

Example text in.ini file	Section name
[sec1]	sec1
[[sec2]]	[sec2
[sec with spaces]	sec with spaces
[seccom] ;comments	seccom
[sectext] sometext	sectext

The following table provides examples of how the Configuration File VIs read key-value pairs in `.ini` files.

Example text in.ini file	Key name	Value
keyname='mystring'	keyname	mystring
;keyname="mystring"	—	—

Example text in.ini file	Key name	Value
keyname="my;string"	keyname	my;string
keyname = 12.3 ;comm	keyname	12.3
key;name="mystring"	key;name	mystring
key\;name="my;string";more	key\;name	"my;string";more
key name6=" mystring6"	key name6	mystring6
keyname8 = mystring8	keyname8	mystring8
keyname9 = " mystring9 "	keyname9	mystring9
keyname12==mystring12	keyname12	=mystring12
#keyname13 = mystring13	#keyname13	mystring13
abc=	abc	—
abc	—	—



Note Because the Read Key VI is a polymorphic VI, it acts differently in certain instances. For example, for the keyname = 12.3 ; comm key-value pair, the String instance returns a string value of 12.3 ; comm and the Double instance returns a numeric of 12.3 for the value output.

The Configuration File VIs can read and write raw or escaped string data. The VIs read and write raw data byte-for-byte, without converting the data to ASCII. In converted, or escaped, strings LabVIEW stores any non-displayable text characters in the configuration settings file with the equivalent hexadecimal escape codes, such as \0D for a carriage return. In addition, LabVIEW stores backslash characters in the configuration settings file as double backslashes, such as \\ for \. Set the **read raw string?** or **write raw string?** inputs of the Configuration File VIs to TRUE for raw data and to FALSE for escaped data.

In configuration settings files, LabVIEW stores path data in the standard Linux format for paths. The VIs interpret the absolute path /c/temp/data.dat stored in a configuration settings file as follows:

- (Windows) c:\temp\data.dat
- (macOS 32-bit) c:temp:data.dat
- (macOS 64-bit and Linux) /c/temp/data.dat

The VIs interpret the relative path temp/data.dat as follows:

- (Windows) temp\data.dat
- (macOS 32-bit) :temp:data.dat

- (macOS 64-bit and Linux) temp/data.dat

Recording Data for Reuse

Use front panel data logging to record data for use in other VIs and in reports. For example, you can log data from a graph and use that data in another graph in a separate VI.

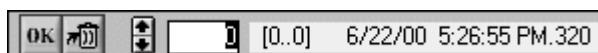


NOTE

LabVIEW does not support front panel data logging through remote panels.

Each time a VI runs, front panel data logging saves the front panel data to a *separate datalog file*, which is in binary format.

A datalog file contains records that include a time stamp and the data from each time you ran the VI. When you access a datalog file, you select which record you want by running the VI in retrieval mode and using the front panel controls to view the data. When you run the VI in retrieval mode, a numeric control appears at the top of the front panel, as follows, so you can navigate among the records.



Automatic and Interactive Front Panel Data Logging

Select **Operate»Log at Completion** to enable automatic logging. The first time you log front panel data for a VI, LabVIEW prompts you to name the datalog file. You can use any file extension, for example .dat or .txt, for datalog files. LabVIEW logs data each time you run the VI and appends a new record to the datalog file each additional time you run the VI. You cannot overwrite a record after LabVIEW writes it to a datalog file.

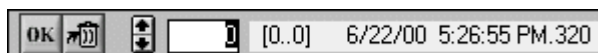
To log your data interactively, select **Operate»Data Logging»Log**. LabVIEW appends the data to the datalog file immediately. Log your data interactively so you can select when to log the data. Logging your data automatically logs the data each time you run the VI.



Note A waveform chart logs only one data point at a time with front panel data logging. If you wire an array to the chart indicator, the datalog file contains a subset of the array the chart displays.

Viewing the Logged Front Panel Data Interactively

After you log data, you can view it interactively by selecting **Operate»Data Logging»Retrieve**. The data retrieval toolbar appears as follows.



The highlighted number indicates the data record you are viewing. The numbers in square brackets indicate the range of records you logged for the current VI. You log a record each time you run the VI. The date and time indicate when you logged the selected record. View the next or previous record by clicking the increment or decrement arrows. You also can use the up and down arrow keys on your keyboard.

In addition to the data retrieval toolbar, the front panel appearance changes according to the record you select on the toolbar. For example, when you click the increment arrow and advance to another record, the controls and indicator display the data for that particular record at the time you logged the data. Click the **OK** button to exit retrieval mode and return to the VI whose datalog file you were viewing.

Deleting a Record

While in retrieval mode, you can delete specific records. Mark for deletion an individual record in retrieval mode by viewing that record and clicking the **Trash** button. If you click the **Trash** button again, the record is no longer marked for deletion.

Select **Operate»Data Logging»Purge Data** while in retrieval mode to delete all the records you marked for deletion.

If you do not delete your marked record before you click the **OK** button, LabVIEW prompts you to delete the marked records.

Clearing the Log-File Binding

Use log-file binding to associate a VI with the datalog file to use when logging or retrieving front panel data. You can have two or more datalog files associated with one VI. This might help you test or compare the VI data. For example, you could compare the data logged the first time you run the VI to the data logged the second time you run the VI. To associate more than one datalog file with a VI, you must clear the log-file binding by selecting **Operate»Data Logging»Clear Log File Binding**. LabVIEW prompts you to specify a datalog file the next time you run the VI either with automatic logging enabled or when you choose to log data interactively.

Changing the Log-File Binding

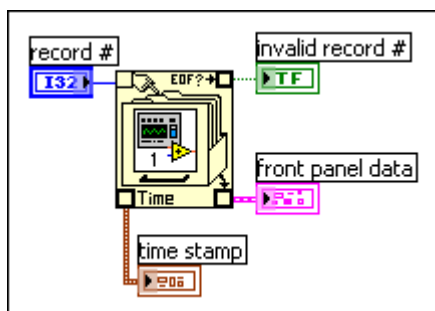
Change the log-file binding to log front panel data to or retrieve front panel data from a different log file by selecting **Operate»Data Logging»Change Log File Binding**. LabVIEW prompts you to select a different log file or to create a new one. You might change log-file binding when you want to retrieve different data into a VI or append data from the VI to another datalog file.

Retrieving Front Panel Data Programmatically

You also can retrieve logged data using a subVI or using the File I/O VIs and functions.

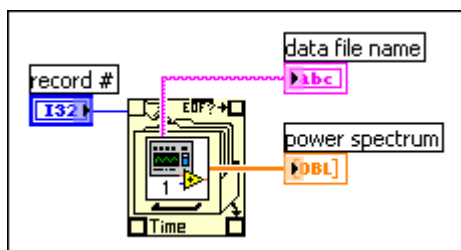
Retrieving Front Panel Data Using a SubVI

When you right-click a subVI and select **Enable Database Access** from the shortcut menu, a yellow box appears around the subVI, as shown in the following block diagram.



The yellow box that looks like a filing cabinet includes terminals for accessing data from the datalog file. When you enable database access for the subVI, the inputs and outputs of the subVI actually act as outputs, returning their logged data. **record #** indicates the record to retrieve, **invalid record #** indicates whether the record number exists, **timestamp** is the time the record was created, and **front panel data** is a cluster of the front panel objects. You can access the data of a front panel object by wiring the **front panel data** cluster to the Unbundle function.

You also can retrieve values for specific inputs and outputs by wiring directly to the corresponding terminal on the subVI, as shown in the following block diagram.



If you run the VI, the subVI does not run. Instead, it returns the logged data from its front panel to the VI front panel as a cluster.



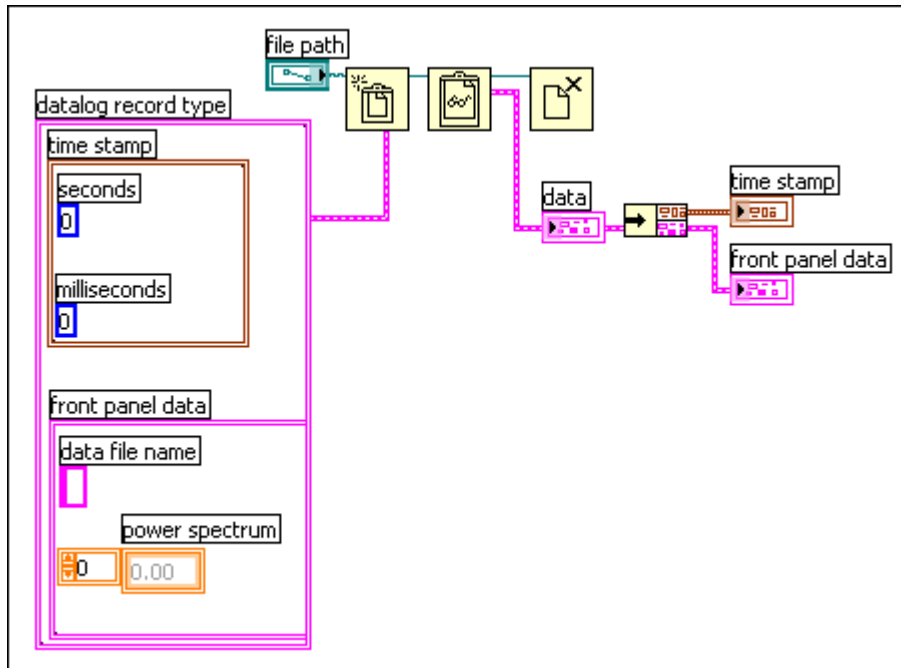
Note If you display a subVI or an Express VI as an expandable node, you cannot enable database access for that node.

Specifying Records

The subVI has n logged records, and you can wire any number from $-n$ to $n - 1$ to the **record #** terminal of the subVI. You can access records relative to the first logged record using non-negative record numbers. 0 represents the first record, 1 represents the second record, and so on, through $n - 1$, which represents the last record.

You can access records relative to the last logged record using negative record numbers. -1 represents the last record, -2 represents the second to the last, and so on, through $-n$,

which represents the first record. If you wire a number outside the range $-n$ to $n-1$ to the **record #** terminal, the **invalid record #** output is TRUE, and the subVI retrieves no data.



LabVIEW Data Directory

Use the default LabVIEW Data directory to store the data files LabVIEW generates, such as .lvproj or .txt files. LabVIEW installs the LabVIEW Data directory in the default file directory for your operating system to help you organize and locate the data files LabVIEW generates. By default, the Write To Measurement File Express VI stores the .lvproj or .tdm files it generates in this directory, and the Read From Measurement File Express VI reads from this directory. The default data directory constant and the Application:Default:Data Directory property also return the LabVIEW Data directory by default.

Select **Tools»Options** and select **Paths** from the **Category** list to specify a different default data directory. The default data directory differs from the default directory, which is the directory you specify for new VIs, custom controls, VI templates, or other LabVIEW documents you create.

Text-Based Measurement File

The text-based measurement file .lvproj includes data the Write To Measurement File Express VI generates. The text-based measurement file is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. In addition to the data an Express VI generates, the .lvproj file includes information about the data, such as the date and time the data was generated. LabVIEW saves data with up to six digits of precision in a .lvproj file.

Use the Read From Measurement File Express VI to *read data from* a .lvproj file.

The `.lvn` file uses commas as delimiters between numbers. To convert data in a `.lvn` file from a string to a number, specify the decimal separator as a period by using the [localization code format specifier](#).

Refer to the following support document at [ni.com](#) for more information about `.lvn` files.

Binary Measurement File

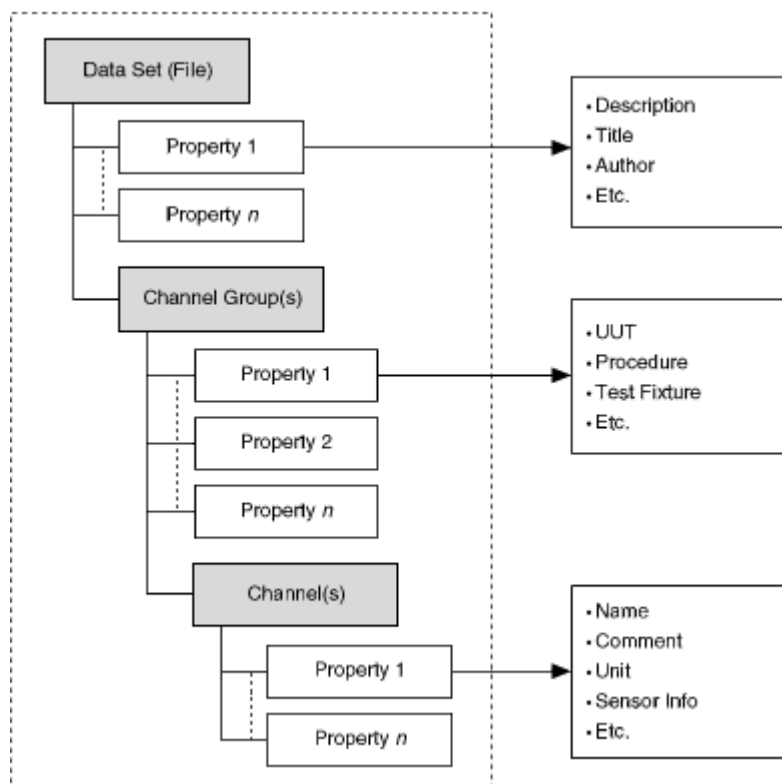
The binary measurement file `.tdm` or `.tdms` includes data the Write To Measurement File Express VI, the Storage/DataPlugin VIs, or the TDM Streaming VIs and functions generate. The `.tdm` file uses an XML-based format to store waveform properties and links to a binary file that contains waveform data. The `.tdms` file stores waveform properties and is a binary file that contains waveform data. Binary measurement files provide higher accuracy, take up less space on disk, and perform faster than [LabVIEW measurement data files \(.lvn\)](#).

You can use a binary measurement file to exchange data between NI software, such as LabVIEW and DIAdem.

(Windows) You can use the Read From Measurement File Express VI and the Write To Measurement File Express VI to [read data from](#) and [write data to](#) `.tdm` files. You can use the Storage/DataPlugin VIs to read data from and write data to `.tdm` files. You can use the TDM Streaming VIs and functions to read data from and write data to `.tdms` files.

TDM Data Model

TDM data model arranges the data in three hierarchical levels: file, channel groups, and channels. The following figure illustrates the TDM data model.



In the previous figure, the file level can contain an unlimited number of channel groups, and each group can contain an unlimited number of channels. Each channel contains a 1D array of data values.

Benefits of the TDM Data Model

This data model enables you to organize your data flexibly and make the data easy to understand. For example, you can have one group for raw data and another group for the analyzed data in one single file. You can also have multiple groups that correspond to sensor types or locations.

At each level of the hierarchy, you can store an unlimited number of custom scalar properties. Each level accepts an unlimited number of custom properties to achieve well-documented and search-ready data files. The descriptive information located in the file, a key benefit of this data model, provides an easy way to document the data without having to design your own header structure. As your documentation requirements increase, you do not have to redesign your application; you only need to extend the current model to meet the specific requirements. The more custom properties you use to document the measurement data, the more easily you can locate the data at a later time.

Related Information

[TDM versus TDMS](#)

[Interacting with TDM Files](#)

[Interacting with TDMS Files](#)

TDM versus TDMS

TDM data model supports the following two file formats:

- (Windows) **TDM** —The TDM file format saves the raw data that the measurement hardware acquires and the meta data in separate files. Meta data can be the name and unit of a measurement channel, the name of the test engineer, and other information. The meta data is in the XML format with the .tdm extension. The raw data is in a compact binary format with the .tdx extension.
- **TDM Streaming (TDMS)** —The TDMS file format saves both the raw data and the meta data in the binary format in one file with the .tdms extension.



NOTE

When creating or opening a .tdms file, LabVIEW might automatically create a .tdms_index file. LabVIEW uses the .tdms_index file to speed up random access to the .tdms file. The .tdms_index file is optional in TDMS applications. If you have disk space limitations, configure the TDMS Open function to stop generating this file.

You can use either of them to store measurement data. The following table helps you decide which file format is more suitable to your application.

Application	TDM	TDMS	Notes
Cross-Platform Support	✗	✓	The TDM file format supports Windows only. The TDMS file format supports Windows, macOS, Linux, and real-time operating systems such as Phar Lap ETS, VxWorks, and NI Linux Real-Time.
DAQmx Support	✗	✓	Requires DAQmx 9.0 or later.
DIAdem Support	✓	✓	Requires DIAdem 10.0 or later.
LabWindows™/CVI™ Support	✓	✓	Requires LabWindows/CVI 8.1 or later.
Measurement Studio Support	✗	✓	Requires Measurement Studio 8.6 or later.
LabVIEW Toolkit Support	✓	✓	The TDM and TDMS file formats work with the DataFinder Connectivity VIs. The TDMS file format also works with the following LabVIEW toolkits and modules: • LTE Measurement Suite 1.0 or later • Simulation Interface Toolkit 4.0 or later • Statechart Module 2.0 or later • Sound and Vibration Measurement Suite 7.0 or later

Application	TDM	TDMS	Notes
Complex Data Support	✗	✓	In addition to supporting all data types that the TDM file format supports, the TDMS file format also supports other data types, such as single-precision and double-precision complex floating-point numbers.
High-Speed Data Streaming Support	✗	✓	LabVIEW includes the TDM Streaming VIs and functions that you can use to stream TDMS data at a high speed. However, LabVIEW does not provide such VIs or functions for the TDM file format.
Advanced File Operations Support	✗	✓	In addition to the standard TDM Streaming VIs and functions, LabVIEW also provides advanced VIs and functions, which enable you to stream TDMS data synchronously or asynchronously. However, LabVIEW does not provide such VIs or functions for the TDM file format.
Data Subset Removal Support	✓	✓	For TDM files, you can use the Delete Data VI in LabVIEW to remove a channel group or channel. For TDMS files, you can use the TDMS Delete Data function in LabVIEW to delete data from a channel or multiple channels in a group.

Related Information

[TDM Data Model](#)

[Interacting with TDM Files](#)

[Interacting with TDMS Files](#)

Interacting with TDM Files (Windows)

In LabVIEW, you can use the following options to read or write TDM files:

- **Express VIs** —The Read From Measurement File and Write To Measurement File Express VIs enable you to read data from or write data to TDM files.
- **Storage/DataPlugin VIs** —The Storage/DataPlugin VIs enable you to perform more operations on TDM files than the Express VIs. In addition to reading or writing data, you can also get or set the properties of a TDM file, channel group, or channel. You also can view the data in a TDM file by displaying the data in a table or a graph.

Refer to the following LabVIEW project files for examples of reading and writing data from and to a channel or a channel group, respectively:

- `labview\examples\File IO\DataPlugins\TDM\TDM Event Data\TDM Event Data.lvproj`
- `labview\examples\File IO\DataPlugins\TDM\TDM Channel Groups\TDM Channel Groups.lvproj`

Related Information

[TDM Data Model](#)

[TDM versus TDMS](#)

[Interacting with TDMS Files](#)

Interacting with TDMS Files

In LabVIEW, you can use the following options to read or write TDMS files:

- **Express VIs** —The Read From Measurement File and Write To Measurement File Express VIs enable you to read data from or write data to TDMS files. The Express VIs offer the ease of interactive configuration; however, they sacrifice performance and are not suitable for high-speed streaming or real-time applications.
- (Windows) **Storage/DataPlugin VIs** —Use the Open Data Storage Express VI to open a TDMS file before you read data from or write data to the file. You also can get or set the properties of a TDMS file, channel group, or channel. However, these VIs also sacrifice performance and are not suitable for high-speed streaming or real-time applications.
- **TDM Streaming VIs and functions** —For more flexibility and to achieve the best performance, use the TDM Streaming VIs and functions. With these VIs and functions, you can read and write TDMS files and their properties in the most efficient way. You can also use these VIs and functions in high-speed streaming or real-time applications.

Refer to the following LabVIEW project files for examples of reading and writing TDMS data:

- `labview\examples\File IO\TDMS\Standard Read and Write\TDMS Standard Read and Write.lvproj`
- `labview\examples\File IO\TDMS\Advanced Read and Write\Asynchronous Read and Write\TDMS Advanced Asynchronous Read and Write.lvproj`

Related Information

[TDM Data Model](#)

[TDM versus TDMS](#)

Interacting with TDM Files

Standard versus Advanced TDMS Functions

The TDM Streaming palette includes both standard and advanced TDMS VIs and functions. The Advanced TDMS VIs and functions enable you to perform advanced file I/O operations, such as asynchronous reads and writes, on .tdms files.



NOTE

Use the standard TDMS VIs and functions when the features of the Advanced TDMS VIs and functions are not necessary. Incorrect use of the Advanced TDMS VIs and functions might result in a corrupt .tdms file. If you disable **buffering** when using the Advanced TDMS VIs and functions, the data size you read or write must be a multiple of the sector size of the hard disk.

Compared to the standard TDMS VIs and functions, the Advanced TDMS VIs and functions have the following advantages:

- Writing meta data and raw data separately—The standard TDMS VIs and functions write meta data and raw data at the same time. The Advanced TDMS VIs and functions enable you to write meta data and raw data separately. You can use the TDMS Set Channel Information function to write the meta data into a .tdms file, and then use the TDMS Advanced Asynchronous Write function to write the raw data into the .tdms file.

Note When you write data to a .tdms file, the data includes meta data and raw data. Refer to the following support document at ni.com for more information about meta data and raw data in a .tdms file.

- (Windows) Reading and writing data asynchronously—The standard TDMS VIs and functions only enable you to read or write data synchronously, which might result in lower performance when the buffer size is small and you have a large size of data to read or write. The Advanced TDMS VIs and functions enable you to read or write data asynchronously. Use the TDMS Configure Asynchronous Reads or TDMS Configure Asynchronous Writes function to allocate buffers and initiate multiple asynchronous reads or writes in the background.
- (Windows) Accessing external data directly to achieve optimal data-streaming performance—Unlike the standard TDMS VIs and functions, the Advanced TDMS VIs and functions enable you to interact with data that is owned by a component external to LabVIEW, such as the direct memory access (DMA) buffer of a device driver that controls a data-streaming device. You can **asynchronously write data from the DMA buffer directly into .tdms files**, or read data from .tdms files directly into the DMA buffer. Accessing the external data in the DMA buffer directly can reduce copies of large data sets and improve the data-streaming performance.

Note This feature requires an **external data value reference** returned from the driver API of data-streaming devices. Wire the external data value reference to the **data reference** input of the TDMS Advanced Asynchronous Read (Data Ref) or TDMS Advanced Asynchronous Write (Data Ref) function to interact with the external data.

- **Overwriting existing data**—Unlike the standard TDMS VIs and functions, the Advanced TDMS VIs and functions enable you to overwrite a subset of data in a `.tdms` file. Use the TDMS Set Next Write Position function to specify the offset at which to overwrite the existing data.
- **Reserving file size before writing data**—Unlike the standard TDMS VIs and functions, the Advanced TDMS VIs and functions enable you to pre-allocate disk space for the data you want to write, which can prevent fragmentation on a file-system level. Use the TDMS Reserve File Size function to pre-allocate disk space before writing data to a `.tdms` file.

File Format Versions

The TDM Streaming (TDMS) file format has two versions: 1.0 and 2.0. File format version 1.0 provides fundamental file operation features, such as reading data from and writing data to a TDMS file, getting and setting properties, and so on. File format version 2.0 includes all the features of version 1.0, as well as the following additional features:

- You can write interleaved data to a TDMS file.
- You can write TDMS data in different byte orders, or endian formats.
- You can write TDMS data without using the operating system buffer. Doing so results in large performance improvements, especially in Redundant Array of Independent Disks (RAID) systems.
- You can use NI-DAQmx to write raw data to TDMS files along with scaling information.
- File format version 2.0 optimizes writing performance for contiguous data acquisition by using a single header for multiple chunks of contiguous data. This improves performance for single value acquisitions.
- File format version 2.0 supports asynchronous writes to TDMS files, which enables the application to process data in memory while writing data to a file instead of having to wait for the writing function to finish.

Synchronous versus Asynchronous File Operations

In LabVIEW, you can read and write TDMS data either synchronously or asynchronously. The terms synchronous and asynchronous here do not define whether a hardware connection shares a common clock or timebase. Instead, these terms refer to how the TDM Streaming VIs and functions behave according to the calling threads.

- **Synchronous I/O** —In this mode, a TDMS application starts a file I/O operation and immediately enters a wait state until the I/O request completes. Before the I/O request completes, this application cannot start a new I/O operation or execute the remaining code. When your application makes many I/O operations, the processor might spend almost all its time waiting idly for the I/O operations to complete.
- (Windows) **Asynchronous I/O** —In this mode, a TDMS application can start and execute multiple file I/O operations simultaneously in the background. This application does not have to wait for the previous I/O operation to complete before

starting a new I/O operation. You can use this mode to improve the throughput, latency, or responsiveness of your TDMS application.

Related Information

[TDM Data Model](#)

[TDM versus TDMS](#)

File Buffering with TDMS Files

During TDMS file I/O operations, data that LabVIEW writes to a TDMS file resides in a buffer until the buffer fills up or until you close the file. Buffering can occur at the channel layer, in the operating system (OS) file system, or in the TDMS disk cache.

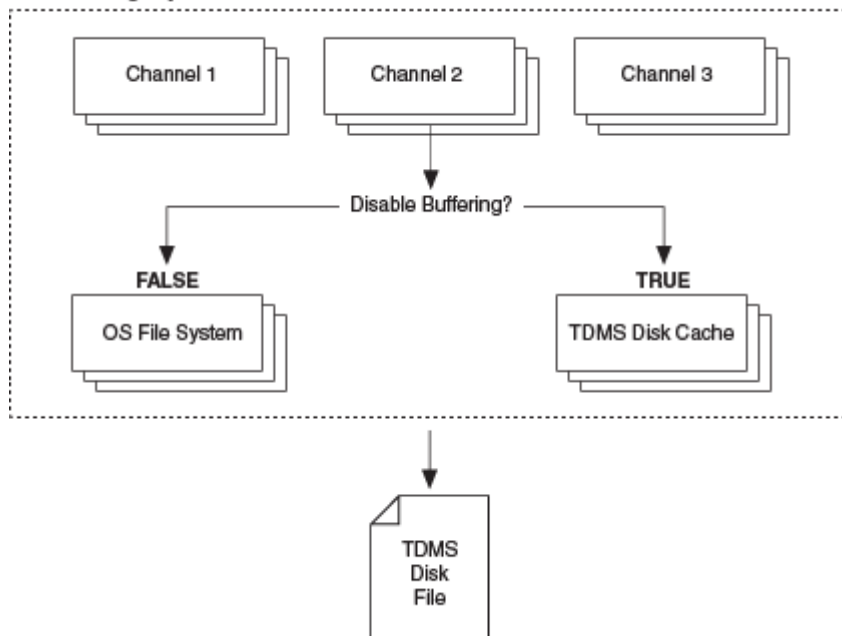
The following figure illustrates buffering with TDMS files for standard TDMS VIs.



NOTE

For the Advanced TDMS VIs and functions, buffering occurs only in the OS file system. If the **disable buffering?** input is TRUE, buffering does not occur at any layer.

File Buffering Layers



The layer at which buffering occurs can depend on the property you set using the TDMS Set Properties function. For example, the `NI_MinimumBufferSize` property determines the number of data samples to buffer at the channel layer. The **disable buffering?** input of TDMS functions determines whether LabVIEW disables system buffering, which occurs in the OS. When the **disable buffering?** input is TRUE, LabVIEW disables system buffering and buffering occurs in the TDMS disk cache. The `NI_DiskCacheSize` property determines

the amount of cache size in the TDMS disk cache to allocate for data. This property is valid only if the **disable buffering?** input is TRUE.

To write all buffers directly to the TDMS file, use the TDMS Flush function.



Note The `NI_MinimumBufferSize` property and the `NI_DiskCacheSize` property do not apply to the Advanced TDMS VIs and functions.

When to Disable Buffering

You may want to disable buffering to speed up data transfers in the following situations:

- When you want to read or write a large sample size.
- When you want to read or write a large amount of data from different samples.
- When you use a Redundant Array of Independent Disks (RAID) to increase processing speed. A RAID is a set of hard disks acting as a single disk that the OS can simultaneously access, taking less time to read or write data. If you access a RAID with buffering enabled, LabVIEW can take more time to copy data to the OS than the OS takes to actually write the data to the disk.

Related Information

[Buffering Data](#)

[File Buffering](#)

[Standard versus Advanced TDMS Functions](#)

Reading TDMS Data Asynchronously

The following guideline describes how to read data from a `.tdms` file asynchronously.

1. Use the TDMS Advanced Open function to open the `.tdms` file. Verify that the value of the **enable asynchronous?** is TRUE.
2. Use the TDMS Configure Asynchronous Reads function to configure settings and allocate buffers for asynchronous reads.
3. (Optional) Use the TDMS Set Next Read Position function to set the file position from which you want to start reading data.
4. Use the TDMS Start Asynchronous Reads function to start reading data to the buffers.
5. (Optional) Use the TDMS Get Asynchronous Read Status function to detect the number of buffers that contain available data.
6. Use the TDMS Advanced Asynchronous Read function to read data from the buffers.

7. (Optional) Use the TDMS Stop Asynchronous Reads function to stop reading data.
8. Use the TDMS Advanced Close function to close the .tdms file when the asynchronous reading process finishes.

Writing TDMS Data Asynchronously

The following guideline describes how to write data to a .tdms file asynchronously.

1. Use the TDMS Advanced Open function to open the .tdms file. Verify that the value of the **enable asynchronous?** is TRUE.
2. Use the TDMS Set Channel Information function to set channel information for the raw data.
3. (Recommended) Use the TDMS Reserve File Size function to pre-allocate disk space for writing.
4. Use the TDMS Configure Asynchronous Writes function to configure settings and allocate buffers for asynchronous writes.
5. Use the TDMS Advanced Asynchronous Write function to stream data that you acquired to the .tdms file.
6. Use the TDMS Advanced Close function to close the .tdms file when the asynchronous writing process finishes.

Waveforms

Waveforms include timing information and corresponding values. After you acquire a waveform signal, you can write the signal to spreadsheet, text, or datalog files.



RELATED INFORMATION

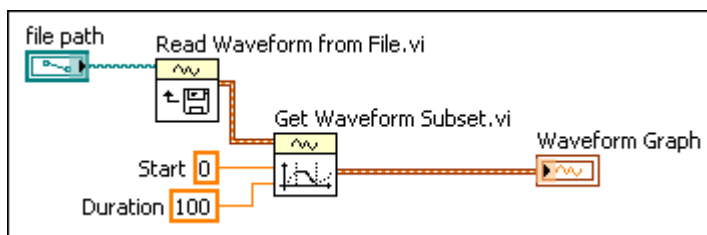
[Reading Waveforms from Files](#) on page 560

[Writing Waveforms to Files](#) on page 561

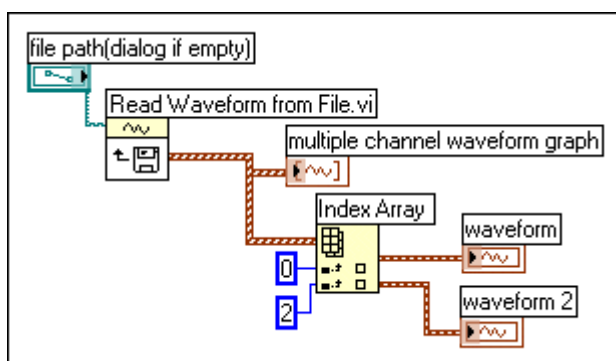
Reading Waveforms from Files

Use the Read Waveforms from File VI to read waveforms from a file. After you read a single waveform, you can retrieve a subset of the waveform at a specified time or index with the Get Waveform Subset VI, or you can extract waveform attributes with the Get Waveform Attribute function.

The following VI reads a waveform from a file, returns the first 100 elements of the waveform, and plots the subset of the waveform to a graph.



The Read Waveforms from File VI also reads multiple waveforms from a file. The VI returns an array of waveform data, which you can display in a multi-plot graph. If you want to access a single waveform from a file, you must index the array of waveform data, as shown in the following block diagram. The VI accesses a file that includes multiple waveforms. The Index Array function reads the first and third waveforms in the file and plots them on two separate waveform graphs.



You also can use the Storage/DataPlugin VIs or the Read From Measurement File Express VI to read waveforms from a file.



RELATED INFORMATION

[Types of Graphs and Charts](#) on page 477

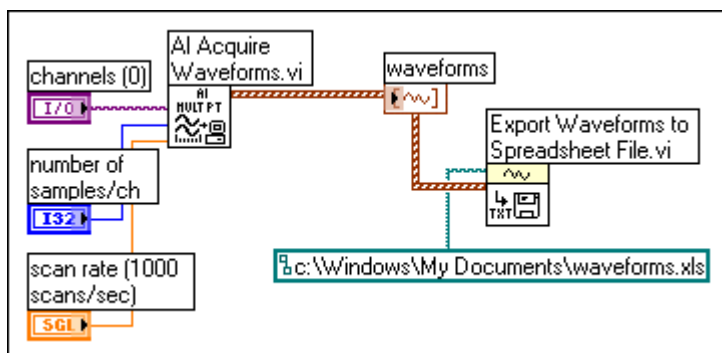
[Grouping Data with Arrays and Clusters](#) on page 460

Writing Waveforms to Files

Use the Write Waveforms to File and Export Waveforms to Spreadsheet File VIs to send waveforms to files. You can write waveforms to spreadsheet, text, or datalog files.

If you expect to use the waveform you create only in a VI, save the waveform as a [datalog file](#).log.

The following VI acquires multiple waveforms, displays them on a graph, and writes them to a spreadsheet file.



You also can use the Storage/DataPlugin VIs or the Write To Measurement File Express VI to write waveforms to files.

Classes and Objects in LabVIEW

LabVIEW Object-Oriented Programming

LabVIEW object-oriented programming uses concepts from other object-oriented programming languages such as C++ and Java, including class structure, encapsulation, and inheritance. You can use these concepts to create code that is easier to maintain and modify without affecting other sections of code within the application. You can use object-oriented programming in LabVIEW to create user-defined data types.

The conceptual information about object-oriented programming in the LabVIEW Help assumes you have knowledge of object-oriented programming principles. The following references might be helpful to you as you use object-oriented programming in LabVIEW if you are not already familiar with object-oriented programming.

Budd, Timothy. *An Introduction to Object-Oriented Programming*. Redwood City, Calif.: Addison-Wesley, 2001.

Coad, Peter, and Jill Nicola. *Object-Oriented Programming*. Upper Saddle River, N.J.: Pearson Education, 1993.

Conway, Jon, and Steve Watts. *A Software Engineering Approach to LabVIEW*. Upper Saddle River, N.J.: Prentice Hall, 2003.

Weisfeld, Matt. *The Object-Oriented Thought Process*. Indianapolis, Ind.: Sam's Publishing, 2004.



NOTE

National Instruments recommends that you begin with *The Object-Oriented Thought Process* as an introduction to object-oriented programming.

Refer to the following support document at ni.com for a LabVIEW object-oriented programming FAQ that contains additional resources.

Refer to the `labview\examples\lloop` directory for examples of object-oriented programming.

Find related examples

Related Concepts

[*Classes and Objects*](#)

[*Creating LabVIEW Classes*](#)

[*Developing LabVIEW Classes*](#)

[*Comparing LabVIEW Class Objects*](#)

[*LabVIEW Interfaces*](#)

[*Using LabVIEW Classes and Interfaces in an Application*](#)

Related How-To

[*Creating a LabVIEW Class or Interface*](#)

[*Creating a Member VI in a LabVIEW Class or interface*](#)

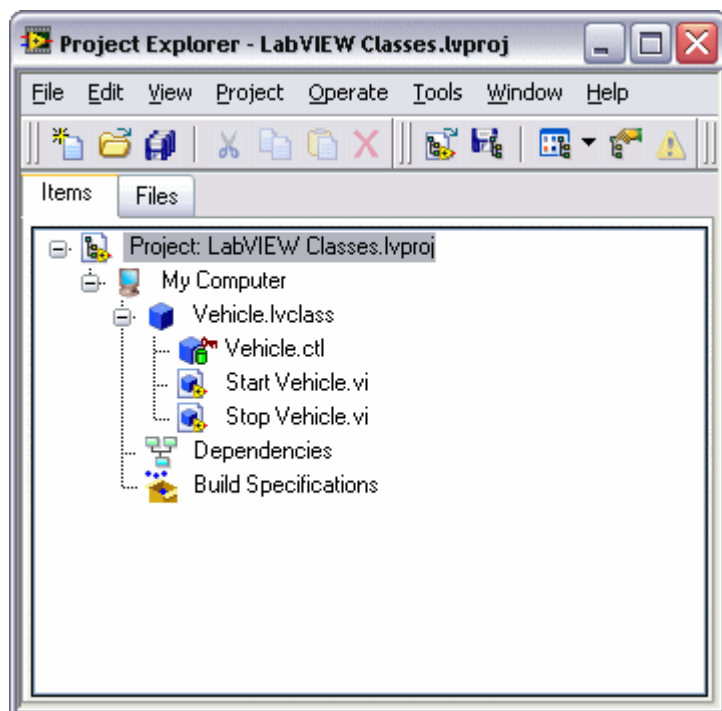
[*Adding Items to a LabVIEW Class or Interface*](#)

[*Changing the Wire Appearance of a LabVIEW Class or Interface*](#)

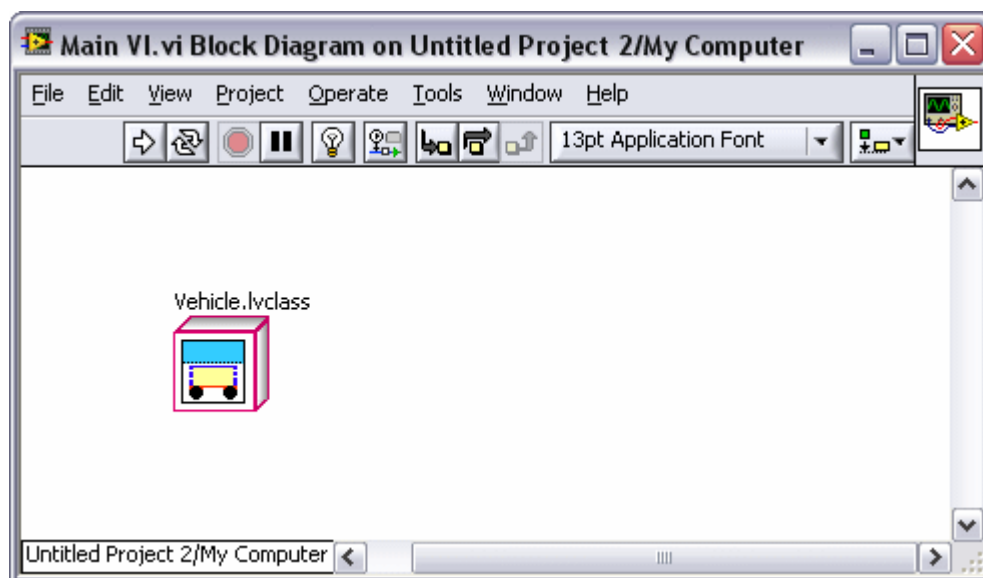
Classes and Objects

In object-oriented programming, classes represent general traits that items share. You can visualize a class, generically, as a vehicle. Vehicles share common traits and the class defines those traits. An object is a specific occurrence of a class. An object of a vehicle class can be the specific car you drive. The class definition determines how your car, the object, behaves.

A class defines the data and methods associated with an object of the class. Continue to use a vehicle as an example. Many vehicles exist in the world. You can generically classify all types of cars, trucks, and buses as vehicles. Vehicles have doors on the body and gears in the transmission. Information about the number of doors and number of gears in the transmission is data about the vehicle. Vehicles also can accelerate and brake. Accelerating and braking are behaviors, or methods, of the vehicle. The data and methods associated with the vehicle define the class Vehicle. You create classes to define the data and methods of the object. The following image displays the LabVIEW class representation of class Vehicle. LabVIEW stores the data of class Vehicle in `Vehicle.ctl` and the methods of Vehicle in VIs named `Start Vehicle VI` and `Stop Vehicle VI`.



An object is a specific instance of a class. The vehicle that you drive is a specific implementation of class Vehicle, or an object of class Vehicle. The following block diagram of Main VI displays an object of class Vehicle. Objects have data and methods defined by the class. LabVIEW stores class data in a control and you create member VIs that are the methods of the LabVIEW class.



For the purpose of conceptualizing the uses for LabVIEW object-oriented programming, you can differentiate users of LabVIEW classes in the following manner:

- **LabVIEW class developer** —Develops LabVIEW classes for other developers and programmers to use. The LabVIEW class developer likely has prior experience in object-oriented programming. The LabVIEW class developer needs to understand the LabVIEW class and its mechanics.

- **LabVIEW class user** —Uses the LabVIEW classes the LabVIEW class developer creates. The LabVIEW class user can take advantage of the benefits of object-oriented programming but does not need to know the mechanics of how the class works to use it in his or her application and may have no previous experience with object-oriented programming. The LabVIEW class user might not have access to the internal operations of the class after the LabVIEW class developer distributes the class. The changes a LabVIEW class developer makes to a LabVIEW class should have minimal impact to the applications developed by LabVIEW class users.

The LabVIEW class user does not need to know how to create a LabVIEW class, but rather how to use the data type a class defines in an application, what information is available to them in debugging code that uses LabVIEW classes, and how a new version of the LabVIEW class might affect the application the programmer has built. If you are using LabVIEW classes only and are not developing them, refer to the [Using LabVIEW Classes and Interfaces in an Application](#) topic for specific information about using LabVIEW classes another developer provides you.

Creating LabVIEW Classes

You create user-defined data types in LabVIEW by [creating LabVIEW classes](#). LabVIEW classes define data associated with an object, as well as the methods that define the actions you can perform on the data. The benefits of [encapsulation](#) and [inheritance](#) allow you to create modular code that is easy to change without affecting code throughout the application.

In LabVIEW, the data of a class is private, which means only VIs that are members of the class can access the data. You define the data of the class in the private data control. When you create and save a LabVIEW class, LabVIEW creates a class library file `.lvclass` that defines a new data type. The class library file records the private data control and information about any member VIs you create, such as a list of the VIs and various properties of the VIs. The class library is similar to the [project library](#) `.lvlib`. However, the class library defines a new data type.

The private data control is a unique class library file that defines a cluster of data for the new data type and is the data on the class wire. LabVIEW does not save the private data control on disk. Instead, LabVIEW saves it inside the class library file. Saving the private data control inside the class library file allows LabVIEW to ensure that you never use the wrong private data with the class definition.



TIP

You can create a directory on disk named the same as the LabVIEW class to save the class library file, [class member VIs](#), and [class custom default probes](#). You can include the files that the class library owns in the directory. If you include files for more than one class library in the same directory, conflicts might occur if you try to include VIs of the same name in different class libraries. Naming conflicts can occur in the development process when you override [dynamic member VIs](#).

Encapsulation

Each LabVIEW class consists of a cluster of data and methods to read and write to that cluster. The data of a LabVIEW class is always private or hidden from VIs that are not

members of the class. To access the private data, you create methods, in the form of member VIs of the class, to perform functions on the private data of the class. Encapsulation is the consolidation of data and methods into a class where the data is accessible only through class member VIs. Encapsulation allows you to create modular blocks of code that you can easily update or change without affecting other sections of code within the application.

While data of the class is always private, you can expose member VIs to the user in varying degrees. You can *set the access scope* of a method to the following options:

- **Public** —Any VI can call the member VI as a subVI.
- **Community** —Only VIs within the same class, friends of the class, or VIs within a friend library of the class can call the member VI. Community member VIs display a dark blue key glyph in the Project Explorer window.
- **Protected** —Only VIs within the same class or a descendant class can call the member VI. Protected member VIs display a dark yellow key glyph in the Project Explorer window.
- **Private** —Only VIs within the same class can call the member VI. Private member VIs display a red key glyph in the Project Explorer window.
- **Not specified** —This option appears only when you select a folder. The folder does not have access items specified. Access is public. By default, folders in a class do not have access specified, which means the folders are publicly accessible.



NOTE

If you specify access options for a folder, the access setting applies to all items in the folder and overrides access options for individual items in the folder.



Note *Dynamic dispatch VIs* break if you set them to community scope. To grant a friend access to a dynamic dispatch VI in protected scope, create a static-dispatch wrapper VI for the friend to call and set that wrapper VI to community scope.

Designating Friends of Libraries

Designate a VI as a friend of a library to give the VI the ability to call any member of the library that is in community scope. You also can designate another library as a friend of a library.

Friendship is not transitive. For example, if a first library designates a second library as a friend, and the second library names a third library as a friend, the third library is not considered a friend of the first library. The third library cannot access VIs of the first library unless the first library designates the third library as a friend. If a library with community scope access designates a class as a friend, the member VIs of that class can access the VIs of the library, but this friendship does not extend to descendants of the class.

You can create a VI outside of the LabVIEW class and use public member VIs as subVIs on the block diagram. Public member VIs allow you to manipulate the private data of the class. You can use private and protected member VIs on block diagrams of member VIs to manipulate the private data of the class that is not exposed to LabVIEW class users. A VI outside the class might be designated a friend of the class, which gives the friend VI the

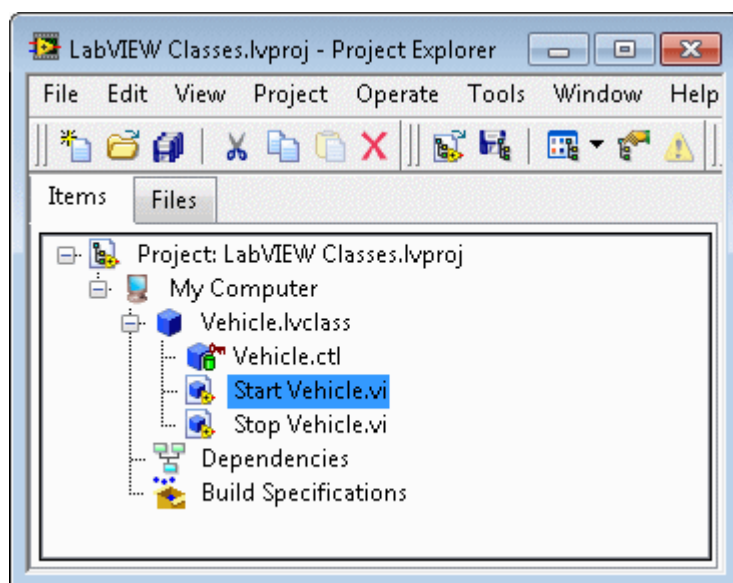
right to call community members of the class. Limiting the entry points of a class can make it easier to debug code because you decrease the opportunity for introducing errors to the data.

Refer to the following projects for examples of encapsulation and access scope in LabVIEW classes.

- labview\examples\Object-Oriented Programming\Data Encapsulation\Data Encapsulation with LabVIEW Classes.lvproj
- labview\examples\Object-Oriented Programming\Access Scope\Access Scope with LabVIEW Classes.lvproj

Defining Private Data Controls

LabVIEW creates a private data control of the class automatically when you [create a LabVIEW class](#). In the following [Project Explorer](#) window, notice that the LabVIEW class icon is a colored cube. The cube represents a LabVIEW class. The icon of the private data control is a colored cube with a green cylinder. The cylinder represents data storage. Also, the icon of the private data control displays a red key glyph indicating the control is private.

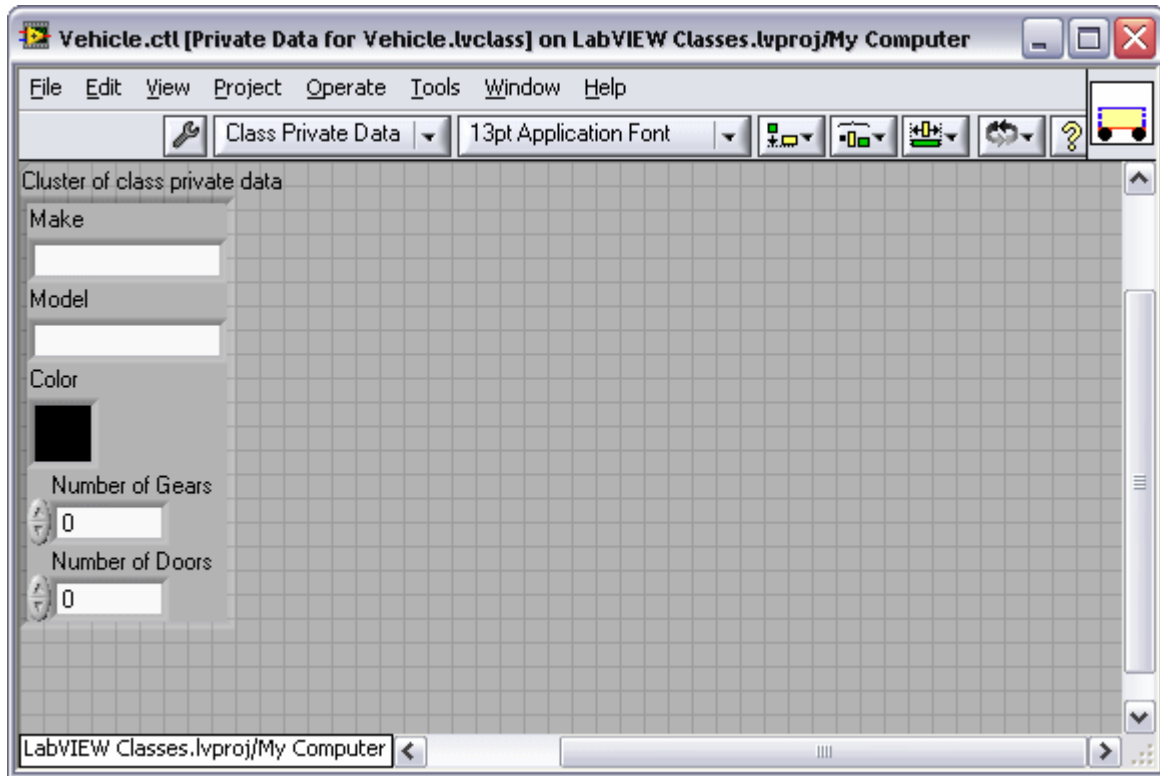


You use the [Control Editor](#) window to customize the private data control of a class. LabVIEW displays the Control Editor window when you double-click the private data control of the class in the [Project Explorer](#) window. You can place controls and indicators in the **Cluster of class private data** to [define the private data type](#) of a LabVIEW class. The default values you set for the controls in the **Cluster of class private data** are the default values for the class.



Note A private data control cannot contain an [XControl](#).

In the following example, the data type of the Vehicle class contains three numerics, **Number of Gears**, **Number of Doors** and **Color**, and two strings, **Make** and **Model**.



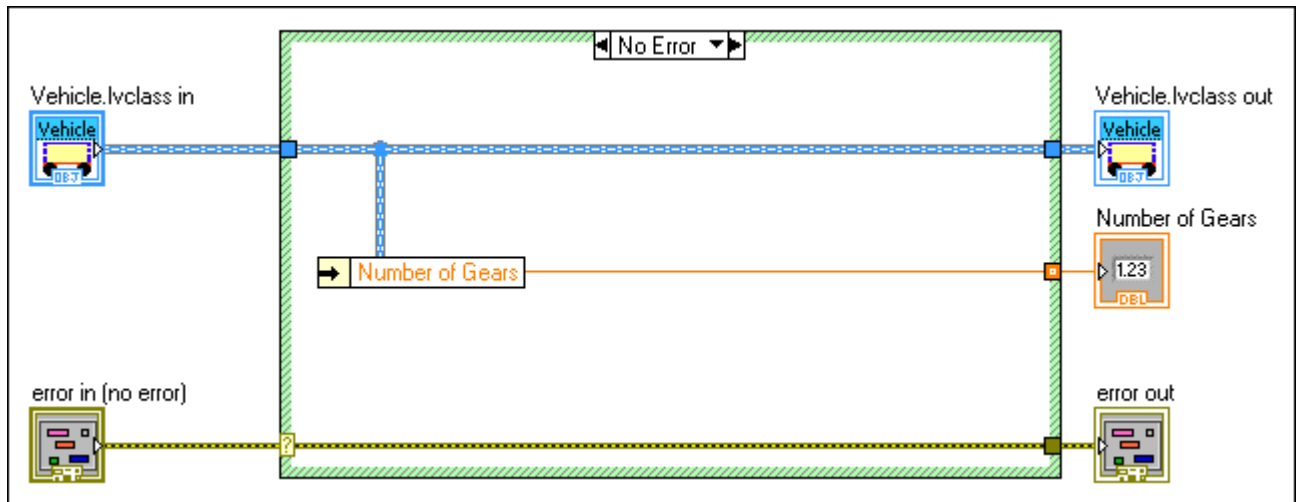
Note You can leave the **Cluster of class private data** empty if the class does not need any private data.

You can create *icons* that visually represent objects of the class on the block diagram or front panel. Click the **Edit** button on the General Settings page of the Class Properties dialog box to launch the Icon Editor dialog box. *Create a class icon*, which LabVIEW *applies to all objects in the class*. Then *modify the individual icon* for each object in the class.

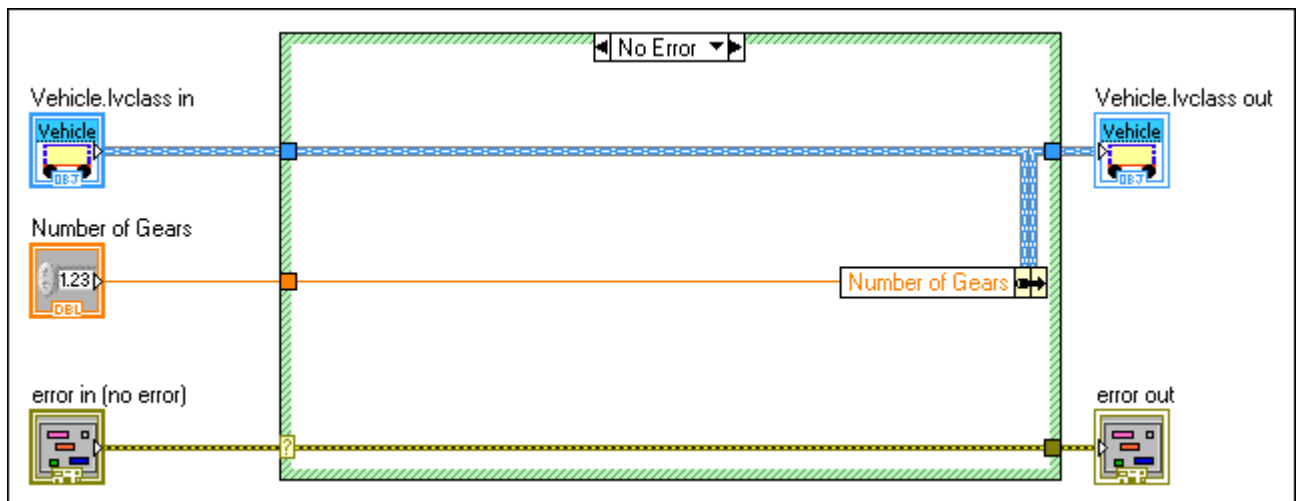
Creating Member VIs

Create member VIs to perform operations on the private data of the class. Member VIs are the methods of the LabVIEW class. Member VIs are members of the LabVIEW class in which you create them and appear in the Project Explorer window under the private data control of the class.

LabVIEW defines data of the class as a cluster. Any member VI can read or write to the cluster of class data. LabVIEW provides a shortcut to *create a VI to access* individual elements of the cluster. These accessor VIs are members of the LabVIEW class and can read from or write to class data. If you create an accessor VI to read class data, LabVIEW unbundles the class data for you, as seen in the following image.



If you create an accessor VI to write to class data, LabVIEW bundles the new value into the class data for you, as seen in the following image.



You also can use the Unbundle and Unbundle By Name functions to unbundle private data of the class on the block diagram of a member VI. Use the Bundle and Bundle By Name functions to re-bundle the cluster of private data after you access and manipulate it. Because the data of a class is private, the Bundle and Unbundle nodes break if you try to use them with class data on the block diagram of a non-member VI.



Note Consider using the Bundle By Name and Unbundle By Name functions instead of Bundle or Unbundle whenever possible to prevent VIs from breaking if you insert a new element into the cluster of private data.

If you are writing a member VI that will unbundle a value, modify it, and then bundle it back into the object, you may gain better performance by using the In Place Element structure with a pair of Unbundle and Bundle nodes on its borders. Using the structure guarantees that LabVIEW applies certain memory optimizations. These same memory optimizations may be applied when you use the regular unbundle and bundle nodes, but under certain conditions with complex VIs, the LabVIEW compiler may not recognize that the optimizations are safe to apply and will avoid applying them, resulting in lower

performance. The structure node guarantees the safety of those optimizations, and thus guarantees that your VI will be optimized.

You can create a member VI in several ways. Right-click the class and select among the following shortcut menu items:

- **New»VI** —Opens a blank member VI.
- **New»Property Definition Folder** —Creates a property definition folder, in which you can create new or add existing member VIs. If a LabVIEW class contains a property definition folder, you can wire the LabVIEW class to a Property Node to access the private data.
- **New»VI from Dynamic Dispatch Template** —LabVIEW populates the new member VI with **error in** and **error out** clusters, a Case structure for error handling, the input LabVIEW class, and the output LabVIEW class. LabVIEW sets both the input and output terminals as *dynamic on the connector pane of the VI*.
- **New»VI from Static Dispatch Template** —LabVIEW populates the new member VI with **error in** and **error out** clusters, a Case structure for error handling, the input LabVIEW class, and the output LabVIEW class. Contrary to creating a dynamic dispatch VI, LabVIEW does not set the input and output terminals as dynamic on the connector pane of the static dispatch VI.
- **New»VI for Data Member Access** —Opens the Create Accessor dialog box. Use this dialog box to quickly create member VIs that can access the LabVIEW class data.

Note You must save a new LabVIEW class before using this option. LabVIEW dims the **VI for Data Member Access** option if you have not saved the new class.

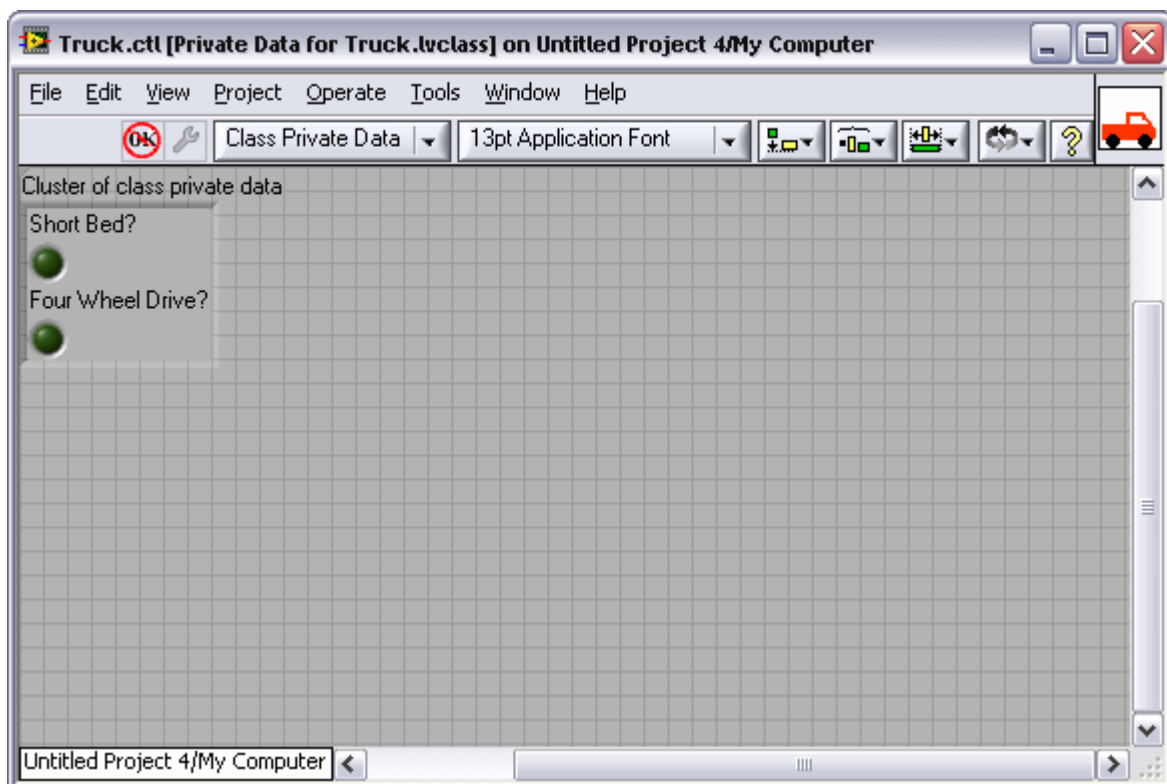
- **New»VI for Override** —Creates a member VI that overrides an ancestor member VI. LabVIEW uses the ancestor VI icon and overlays the icon mask of the child class to create the icon of the new VI.

Note LabVIEW dims the **New»VI for Override** option if there is not a valid member VI to override. Refer to the *Inheritance* section for more information about dynamic VIs and overriding.

Right-click a class constant, control, or indicator on the front panel or block diagram and select **Show Class Library** from the shortcut menu to highlight the class in the Project Explorer window. If the class you are working with is not in a LabVIEW project, LabVIEW opens a class window to display the class.

Inheritance

Inheritance allows you to use an existing class as the starting point for a new class. If you create a new LabVIEW class and *set it to inherit data and member VIs* from another class, the new class can use the public and protected member VIs of the class from which it inherits. It also can add its own data and member VIs to increase its functionality. For example, the private data of the Vehicle class contains **Number of Gears**, **Number of Doors**, **Color**, **Make**, and **Model**. If you create a new class and name it Truck, you can set Truck to inherit the data from Vehicle as well as add the Boolean data **Short Bed?** and **Four Wheel Drive?**. However, as shown in the following image, when you bundle or unbundle a LabVIEW class, the nodes show terminals only for the private data of the current class, not for any of the data the class inherits from the ancestor classes.



Ancestor data is private and you modify it by using functions the ancestor class provides through member VIs. Member VIs of descendant classes may call any of the public member VIs, just like any VI in LabVIEW. But the descendant class member VIs also can call the protected member VIs of an ancestor class. When you designate an ancestor member VI as protected, member VIs of any child class may call the method but no VI outside of the inheritance hierarchy may do so. If you want to access **Number of Gears** of class Vehicle in class Truck, you can create a public or protected member VI in class Vehicle called Get Gears.vi. You can unbundle class Vehicle on the block diagram of Get Gears.vi, exposing **Number of Gears**. You then can assign **Number of Gears** to an output terminal on the connector pane, which allows you to access specific private data of class Vehicle in a descendant class, such as class Truck.

You also can access **Number of Gears** by [creating a member VI for accessing data members](#) in the class Vehicle. When you create the member VI, in the Create Accessor dialog box, place a checkmark in the **Make available through Property Nodes** checkbox. You then can wire the class Truck to a Property Node, right-click the property terminal, and select **Select Property»Number of Gears**.



Note LabVIEW classes cannot call the private member VIs of another LabVIEW class even from a parent class. You can use private member VIs only on block diagrams of other member VIs within the same class.

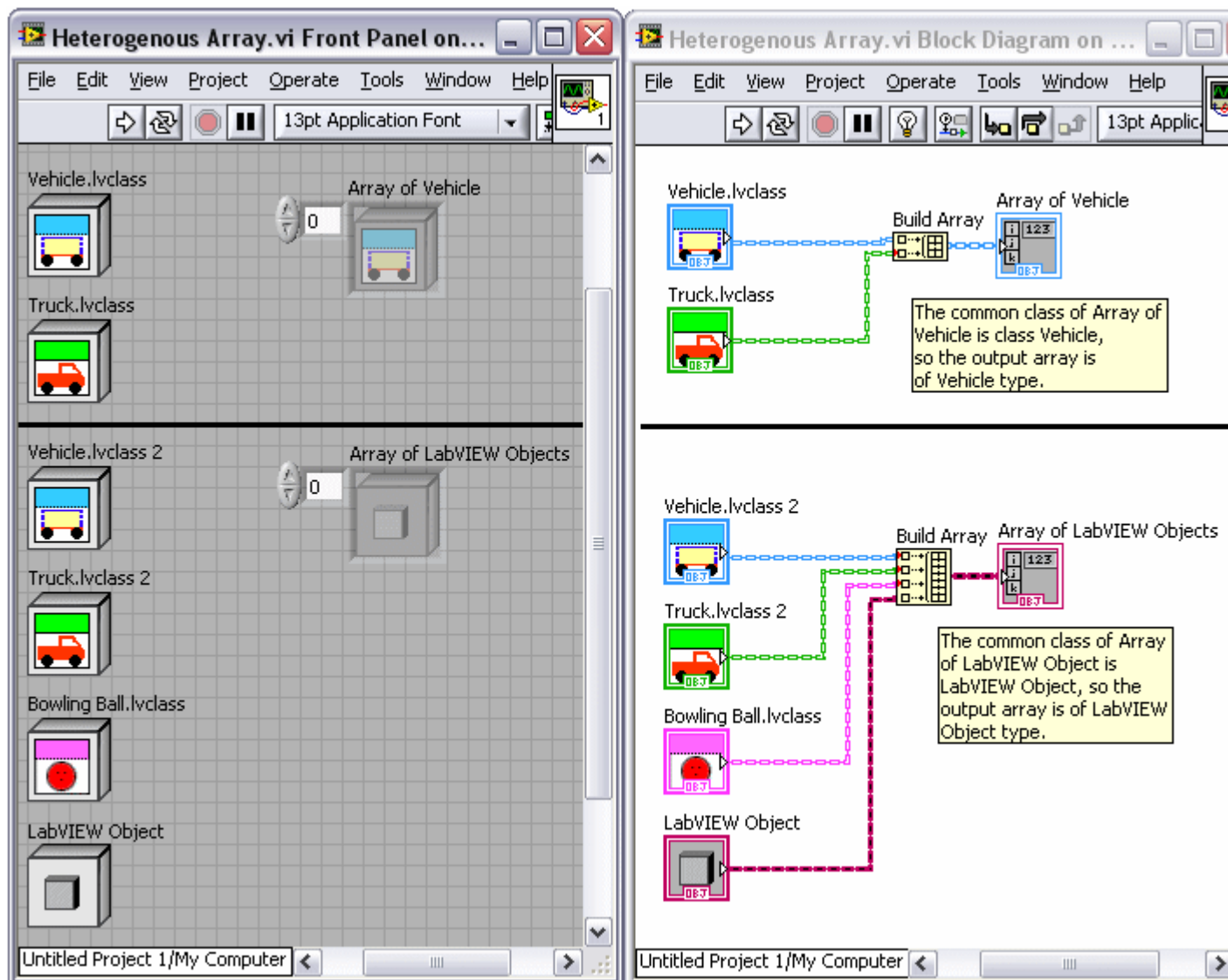


Note (FPGA Module) You can use inheritance with certain restrictions when creating FPGA VIs.

LabVIEW Object

The phrase LabVIEW Object is the name of a specific class. *LabVIEW Object* is the ultimate ancestor of the *inheritance tree* in object-oriented programming in LabVIEW. By default, all LabVIEW classes inherit from LabVIEW Object. You can use LabVIEW Object to create VIs that can conduct generic operations on multiple LabVIEW classes. For example, if you create an array of LabVIEW classes, the data of the array is heterogeneous because it can contain elements of the class type of the array or any descendant class. If an array is of type LabVIEW Object, it can contain Vehicle, Truck, and Bowling Ball. Class Bowling Ball does not inherit from class Vehicle or class Truck, so LabVIEW creates an array of the most common ancestor base class, in this case LabVIEW Object.

The following image displays **Array of Vehicle**, an array that contains class Vehicle and class Truck. Because Truck inherits from Vehicle, the common ancestor base class of this array is type Vehicle. The image also displays **Array of LabVIEW Objects** that contains classes LabVIEW Object, Vehicle, Truck, and Bowling Ball. Bowling Ball does not inherit from Vehicle or Truck, but all three ultimately inherit from LabVIEW Object; therefore, **Array of LabVIEW Objects** is of type LabVIEW Object.



Setting Inheritance

All LabVIEW classes inherit from LabVIEW Object by default. If you want to change which class another class inherits from, you must [change the inheritance](#) after you create the involved classes. You can configure the inheritance and other options of the class in the Class Properties dialog box. You can [view the hierarchy of LabVIEW classes](#) in the LabVIEW Class Hierarchy window. The class inheritance hierarchy can include the following types of classes.

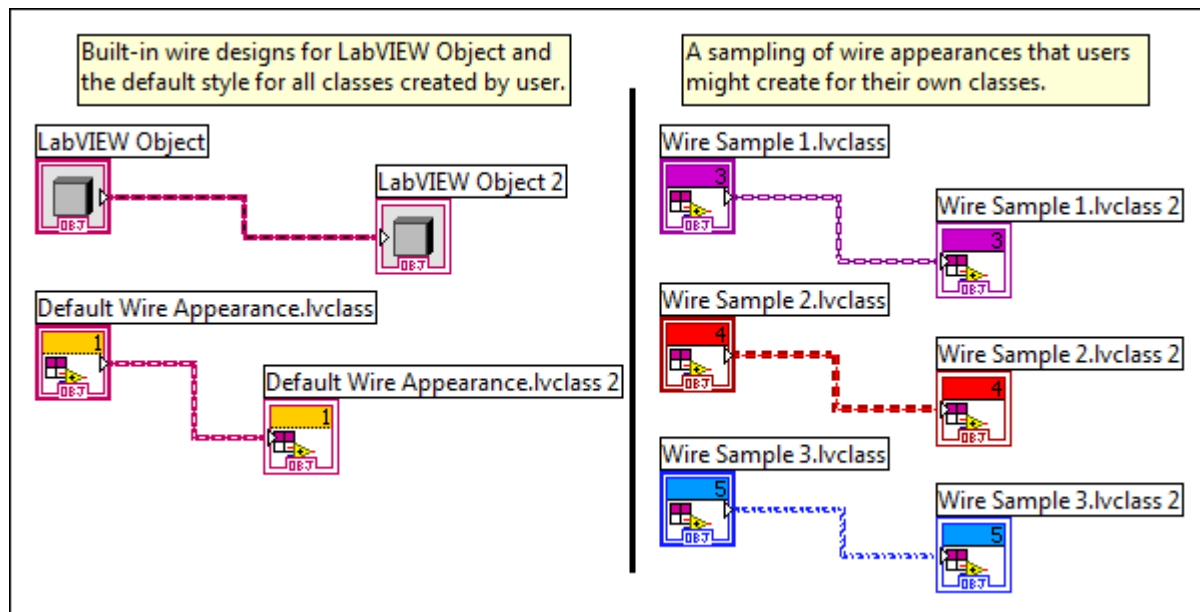
- **Parent class** —LabVIEW class from which other LabVIEW classes inherit data and public and protected member VIs.
- **Child class** —LabVIEW class that inherits the public and protected member VIs of the parent class. The child class does not inherit private data unless the parent class provides an accessor VI.
- **Sibling class** —LabVIEW class that shares the same parent class as another LabVIEW class.
- **Ancestor class** —LabVIEW class that is the parent, grandparent, great-grandparent, and so on of a LabVIEW class. LabVIEW Object is the ultimate ancestor of all LabVIEW classes.
- **Descendant class** —LabVIEW class that is the child, grandchild, great-grandchild, and so on of a LabVIEW class.



Note If you create a VI that overrides a property accessor VI in a parent LabVIEW class, you must provide the same name and folder path in the property definition folders for both the child and parent classes.

Wire Appearance

Classes define new data types. Wires of those class types appear on the block diagram as the default LabVIEW class wire or inherit the wire appearance of the parent class. You can [change the wire appearance](#) of a LabVIEW class in the Class Properties dialog box. To create a block diagram that is easy to read, you can change the wire appearance of different LabVIEW classes when appropriate. Having excessive wire colors and patterns can make the block diagram more difficult to read. The following image demonstrates the LabVIEW built-in wire appearances on the left and some possibilities for custom wire appearances on the right.



Refer to the [LabVIEW Style Checklist](#) for more tips on how to avoid creating excessive wire patterns and colors in LabVIEW.

Dynamic and Static Dispatch Member VIs

A method is an operation that may be performed on or by an object. In LabVIEW object-oriented programming, methods are member VIs you create. The member VIs perform an operation on the data of the LabVIEW class. You can define some methods with a single VI. These are called static dispatch methods because LabVIEW calls the same VI every time. You also can define methods by multiple VIs with the same name throughout the class hierarchy. These are called dynamic dispatch methods because exactly which one of the set of VIs LabVIEW calls is not known until run time. Dynamic dispatch methods are similar to polymorphic VIs. Where polymorphic VIs determine which VI to call depending on the data type you wire to it, dynamic dispatch methods wait until run time to determine which member VI in the class hierarchy to call depending on the data that arrives at the input terminal.

You designate a member VI as either static or dynamic on the connector pane of the member VI. If the connector pane includes a dynamic dispatch input terminal, the member VI is part of a dynamic dispatch method. If there is no dynamic dispatch input terminal, the member VI defines a static dispatch method.

When one LabVIEW class inherits from another LabVIEW class, the child class inherits all the public and protected methods defined on the parent class. By naming a member VI in a child class exactly the same name as a member VI in the parent class, you define the child implementation of the method.

Because LabVIEW defines static dispatch methods by a single VI, do not name a child class member VI with the same name as a static dispatch member VI in the ancestor class. For example, if parent class Vehicle includes a static dispatch member VI, Open Door VI, then child class Truck cannot have a member VI named Open Door VI. The method Open Door VI is already defined on Truck because Truck inherits the member VI from Vehicle. If you place static dispatch member VIs on the block diagram as a subVI they behave in all ways as a normal subVI call.

You can define multiple dynamic dispatch VIs for a method, one at each level of an inheritance hierarchy. If a dynamic dispatch VI is defined in the parent class, and you also define it in the child class, the child class implementation overrides, or extends, the parent implementation. In the following example, class `Vehicle` and class `Truck` both define an implementation of the dynamic dispatch method `Set Make VI`. If you place a dynamic dispatch VI on the block diagram as a subVI, the node on the block diagram behaves just like a regular subVI call when LabVIEW is in edit mode. If you run the VI however, the data that comes into the dynamic dispatch input terminal determines which implementation of the dynamic member VI in the class hierarchy LabVIEW calls. Because a LabVIEW class wire can carry data of its own type or data of any child type, the node executes whichever implementation of the dynamic dispatch VI you define for the class data. Examine the following example again. Only the class `Vehicle` implementation of `Set Make VI` is on the block diagram of `Main VI`. In the first iteration of the loop, LabVIEW executes the class `Vehicle` implementation of `Set Make VI` because the data for class `Vehicle` is on the class wire. In the second iteration of the loop, LabVIEW executes the class `Truck` implementation of the `Set Make VI` because the data for class `Truck` is on the class wire.

If the parent class defines a dynamic dispatch VI but does not provide an implementation of that VI, each child class must override the parent VI. In many cases, it is impossible for you to provide any meaningful behavior in the parent class VI. For example, suppose you have class `Shape` which defines `Area VI`. `Area VI` returns the area of the shape in the `Shape` object. There is no single formula for calculating area that works for all, or even a majority of, shapes. Therefore, each descendant class must override `Area VI` with the corresponding formula for their shape. Thus, if you create a child class `Circle`, class `Circle` must provide an `Area VI` that calculates $\pi * \text{radius} * \text{radius}$.

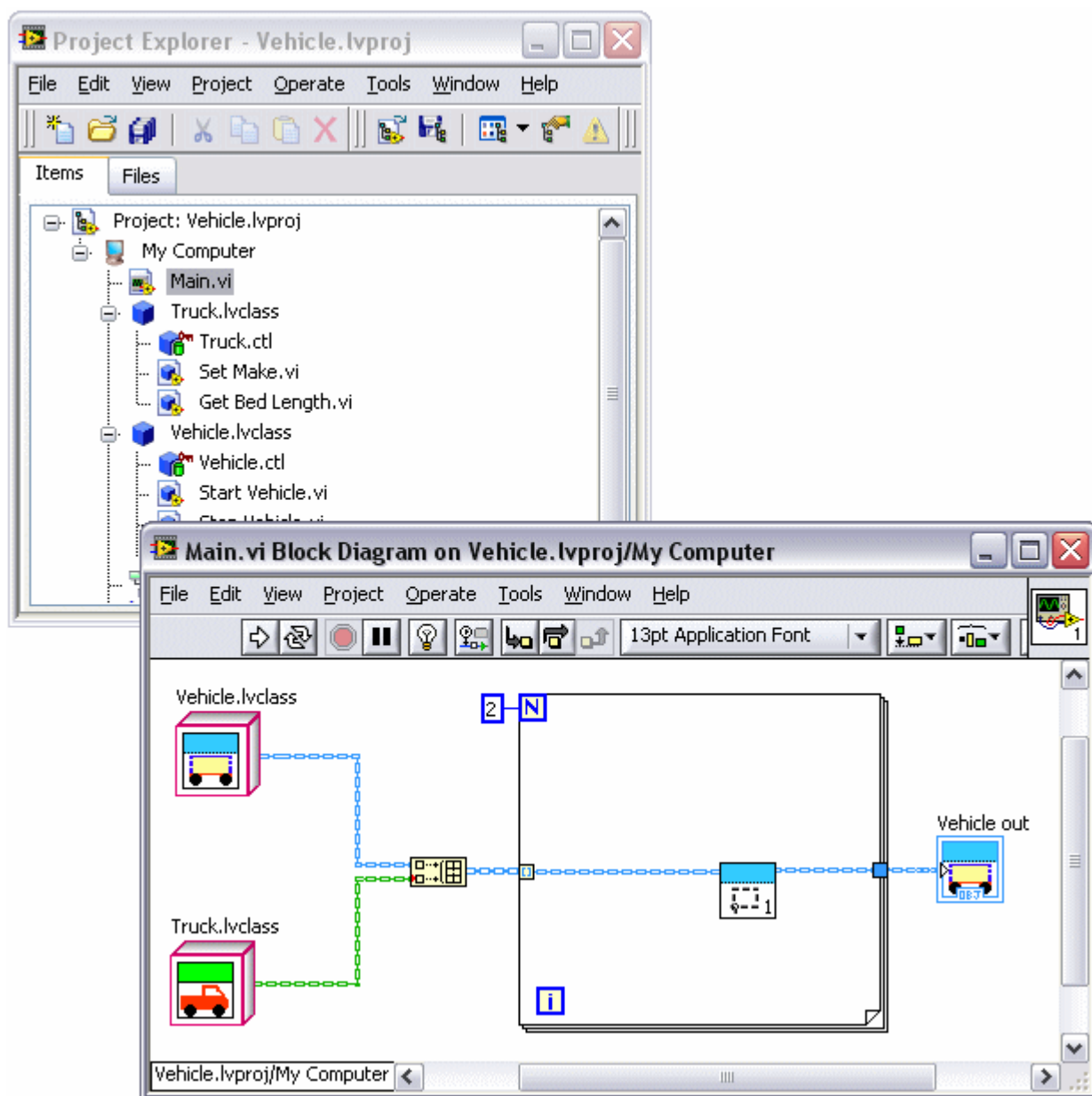
If you do not define an implementation of the parent class VI, then the parent VI is merely a definition of the connector pane and VI properties that all override VIs must match. Each child class must provide one such override VI. To ensure LabVIEW enforces this requirement, you can [mark the VI](#) in the parent class as one that each child class must override and let LabVIEW enforce the requirement.

Some child classes might not be able to override the functionality of the member VI. For example, if class `Quadrilateral` is a child of class `Shape`, you cannot provide the correct formula for area unless you know the specific kind of quadrilateral. To avoid the creation of an empty implementation of `Area VI` within class `Quadrilateral`, you can instruct class `Quadrilateral` to transfer the override requirement to its own child classes. To do so, right-click the child class and select **Properties** from the shortcut menu to display the Class Properties dialog box. On the Inheritance page, place a checkmark in the **Transfer all Must Override requirements to descendant classes** checkbox. LabVIEW then requires child classes of class `Quadrilateral`, such as `Trapezoid` and `Rectangle`, to override `Area VI`.

Requiring descendant classes to override parent class member VI has no effect on how the member VI runs.



Note LLBs cannot contain files of the same name. Therefore, if you have dynamic member VIs that share names in a class hierarchy, you cannot put these classes in the same LLB.



Note When a child class' VI overrides a parent class' VI, the child's VI must match the parent's VI for reentrant settings, preferred execution settings, priority settings, connector pane terminal, connector pane pattern, and [access scope](#).

Double-click a dynamic dispatch subVI that is on the block diagram to display the Choose Implementation dialog box. You can use this dialog box to view all implementations of a dynamic dispatch subVI that are currently in memory and then [open one or more implementations](#) of the subVI.

If you create a VI to override an ancestor implementation of a dynamic dispatch member VI by selecting **New»VI for Override**, you create another dynamic dispatch member VI, because VIs that override an ancestor member VI have the same name as the ancestor and also include dynamic dispatch terminals. LabVIEW places a Call Parent Class Method node on the block diagram with the appropriate [dynamic dispatch input and output class](#)

[terminals](#) and other terminals the VI needs to match the ancestor VI. LabVIEW disables the **VI for Override** option if no ancestor member VI exists to override.



Note (FPGA Module) You can use dynamic dispatching with certain restrictions when creating FPGA VIs.

Dynamic Dispatch Outputs

You can mark a LabVIEW class output terminal as dynamic by right-clicking the output terminal on the connector pane and selecting **Dynamic Dispatch Output (Recommended)**. When you call a VI with a dynamic dispatch output terminal as a subVI the dynamic dispatch output changes to the same data type as the wire connected to the dynamic dispatch input terminal. For example, if you connect a wire of class Vehicle to a dynamic dispatch input terminal, the output of the member VI is the same as the input; in this case the output is class Vehicle. You can modify the data between the dynamic dispatch input terminal and the dynamic dispatch output terminal. However, to ensure the run-time safety of LabVIEW classes, the data from the dynamic dispatch input terminal must flow to all dynamic dispatch output terminals. Also, to ensure that LabVIEW reads from a dynamic dispatch input terminal exactly once and writes to a dynamic dispatch output terminal exactly once, you cannot place dynamic dispatch block diagram terminals inside a structure.



Note If you are debugging a dynamic dispatch member VI with dynamic dispatch inputs and dynamic dispatch outputs, you can examine the wire that flows from the dynamic dispatch input to the dynamic dispatch output for errors. The background color of the wire is gray instead of the usual white for any wire that originates at a dynamic input and does not pass through any functions that can change the run-time data type. The background color of the wire turns red if the wire passes through a function that can change the data type. In order for the dynamic outputs to work correctly, you cannot change the data type of a LabVIEW class.

If you want to complete an operation on the block diagram of a member VI that you know results in a LabVIEW class output data type that is different than the input, make sure the LabVIEW class dynamic output terminal on the connector pane is set to **Recommended** instead of **Dynamic Dispatch Output (Recommended)**. For example, if the LabVIEW class input is Vehicle and you know that you want to output the LabVIEW class Truck, you should change the LabVIEW class default terminal on the connector pane. Alternatively, you can [create the member VI](#) using a blank VI, which allows you to manually set the connector pane terminals.



Note If you use a Case or Event structure in a member VI with dynamic dispatch inputs and dynamic dispatch outputs, you cannot select **Use Default if Unwired** on the output tunnels. If you use **Use Default if Unwired** on the output tunnel, LabVIEW breaks the VI. You must wire all the cases in the structure. Consider [configuring the tunnels](#) to wire the input and output tunnels automatically.

Optimization of Memory Copies for Dynamic Dispatching

As mentioned above, when you create *dynamic dispatch methods*, each child class inherits all the public and protected methods defined on the parent class. The child class can override, or extend, these member VIs with versions of their own. When a VI calls a dynamic dispatch method, LabVIEW does not know which version of the method it will invoke until run time. So LabVIEW optimizes the memory allocations of the caller VI to assume that any member VI that a child class contains will be set up the same as the member VI in the parent class. If inputs are constant in the parent VI, the caller VI assumes they are constant in all child VIs. If inputs return as outputs on the parent class member VI, the caller VI assumes they do the same on all child class member VIs.

Optimization is lost if any of these assumptions are wrong. For example, you can create a member VI that contains inputs that are not modified, which makes the caller VI assume that none of the inputs will ever be modified, even though every single overriding implementation may modify some or all of the inputs. Or, if you do not wire any of the inputs through to their outputs on the parent class member VI, the caller VI assumes that none of the outputs share memory with the inputs, even though subsequent member VIs might wire the inputs through to the outputs. LabVIEW must create code to handle the unexpected behavior in the child classes and this leads to inefficient optimization.

For better optimization, you can write the parent class VI so that it is as consistent as possible with what you expect to occur in the child class VIs. One way you can create a dynamic dispatch method so that you clearly indicate the expected behavior of each terminal is to use the In Place Element structure. Place the In Place Element structure inside of the dynamic dispatch VI within the parent class and add In Place Element node pairs to the In Place Element structure. With these node pairs, you are able to tell LabVIEW which inputs should connect to which outputs, which inputs ought to be treated as constant, and which should be treated as modified. LabVIEW is then able to optimize the caller VI.

Refer to the following projects for examples using dynamic dispatch terminals.

- labview\examples\Object-Oriented Programming\Dynamic Dispatching\Dynamic Dispatching.lvproj
- labview\examples\Object-Oriented Programming\Dynamic Terminals\Dynamic Terminals.lvproj

Developing LabVIEW Classes

While LabVIEW object-oriented programming is similar to other object-oriented programming languages conceptually, because LabVIEW is a dataflow graphical programming environment some differences exist in how LabVIEW manipulates and interacts with LabVIEW class data and how you can debug the class code.

Objects in LabVIEW are passed by value, not by reference. LabVIEW creates copies of the object following the same rules traditionally used for *clusters and arrays*.

Constructors and Destructors

Constructors and destructors in LabVIEW object-oriented programming are implicit. You do not have to call a constructor to initialize LabVIEW class data. LabVIEW calls a default constructor whenever LabVIEW needs to initialize a class. Initialization of a class typically occurs at a front panel control or block diagram constant. LabVIEW initializes the values of the class as the default values you set in the private data control. When LabVIEW no longer needs the LabVIEW class information, LabVIEW handles memory deallocation as it does for [clusters and arrays](#). If you want to set another value for the class data, you must create a member VI that sets the new value. For example, if you set the default value of **Number of Gears** in class Vehicle to 3 when you create class Vehicle, and you want to assign a different value to **Number of Gears** that class Truck inherits, you must create a member VI that changes the value in class Vehicle. One option for setting a new value is to create a member VI that has no inputs but outputs a class with the value you want to assign to **Number of Gears**, thereby creating a block diagram constant of the class data type.

Flattening and Unflattening Data

LabVIEW stores data as [flattened data](#). The Flatten To String and Unflatten From String functions can handle all class data types. LabVIEW flattens and unflattens the data automatically. While all [type descriptors](#) retain the type of data for flattened data types in LabVIEW, the LabVIEW class flattened data itself retains this information as well as class version information. Because the LabVIEW class retains information pertinent to unflattening the LabVIEW class, if the LabVIEW class cannot be found because you move or delete the class, LabVIEW cannot unflatten the data and you receive an error message. Similar to the behavior LabVIEW exhibits when it encounters a missing subVI, you can open the member VI and notice LabVIEW dims the control that is missing the LabVIEW class data. When you load the LabVIEW class corresponding to the broken control, the data is unflattened and the control is no longer broken.



NOTE

You also can use the Flatten To XML and Unflatten From XML functions to handle class data.

Mutating Data

A LabVIEW class control reports a warning if the data in the control is written with a more recent version of the LabVIEW class than the version of the class LabVIEW has in memory. LabVIEW increments the LabVIEW class version number anytime you make changes to the data type of the private data control or the inheritance of the class. You can view the LabVIEW class version number in the Class Properties dialog box. LabVIEW increments the version number of the class automatically if:

- the LabVIEW class inheritance changes.
- the private data control of the LabVIEW class changes. These changes include adding, replacing, reordering, or deleting controls; changing the representation of a control; or any updates from modified type definitions.
- the LabVIEW class loads in a new version of LabVIEW.
- you increment the version number in the Class Properties dialog box manually. You might do this to indicate a new revision of the class.

If you rename a class, LabVIEW considers it a new class, deletes the mutation history of the class, and resets the version number to 1.0.0.0. LabVIEW renames the class if you do one of the following actions:

- Rename the class by selecting **File»Save As** from the pull-down menu.
- Move the class into an owning library.
- Move the class out of an owning library.
- Rename the `.lvclass` file on disk.



Note If you decide to change the name back to the original class name without making any other changes, the class will not be identical to the original class because the mutation history no longer exists. For example, if you rename class Vehicle to Vehicle #1 and then change the name back to Vehicle, class Vehicle is no longer identical to the original class.

Tracking the version number of the LabVIEW class is beneficial to LabVIEW class users who develop applications using LabVIEW classes. For example, you have a LabVIEW class in an application that uses an unsigned 32-bit integer in the private data control of the class. A LabVIEW class developer can send you a version of the LabVIEW class that changes the numeric control in the private data control to a double-precision floating-point number. Because LabVIEW tracks version changes and knows how to flatten and unflatten data on all LabVIEW classes, you can replace the old version of the LabVIEW class with the new version and run the program without making changes to it.



Note If you try to unflatten data from a future version of a LabVIEW class, LabVIEW returns an error. For example, this error might occur if you have data with a version number of 1.0.0.3 but the LabVIEW class in memory is version 1.0.0.2.

When LabVIEW encounters an older version of the LabVIEW class, LabVIEW handles the data mutation in one of the following ways:

- If you remove classes from the LabVIEW class hierarchy, LabVIEW deletes the flattened data of the class you remove. For example, class C inherits from class B, which inherits from class A. If you change the inheritance of class C to inherit directly from class A, LabVIEW deletes the flattened data of class B in that instance.
- If you add layers in the LabVIEW class hierarchy, LabVIEW inserts the class default data of the class you insert. For example, class C inherits from class A. If you change the inheritance of class C to inherit from class B, which inherits from class A, LabVIEW inserts the class default data of class B.
- If you add an element to the private data cluster of the class at the end of the **tabbing order**, LabVIEW unflattens the old data and appends the class default data.
- If you delete an element from the private data cluster of the class and do not add any new elements, LabVIEW unflattens the old data and deletes the information about the element.
- If you change the tabbing order of the private data cluster of the class, LabVIEW reorders the unflattened data accordingly.

- If you add, delete, replace, or reorder the private data cluster of the class in the same edit, LabVIEW creates and executes a procedure for appropriately mutating the old data through combinations of the steps above.

LabVIEW always applies mutations across multiple versions serially, for example from 1.0.0.2 to 1.0.0.6.



Note You cannot revert a LabVIEW class if the class version is incremented. LabVIEW does not have sufficient information to undo a change to the data in LabVIEW class controls and indicators. To revert current changes, do not save any changes. Unload the class and all VIs that reference the class and then reload from disk.

Refer to the following support document at ni.com for more information about how LabVIEW mutates data when you edit a LabVIEW class.

Casting LabVIEW Classes

You can upcast classes using the To More Generic Class function and downcast using the To More Specific Class function. You also can use these functions for refnum data types that have an inheritance hierarchy, such as [VI Server Control Refnums](#), and the functionality is identical. Using the To More Generic Class function does not change the data but it does change the type of the wire that carries the data. However, the function has no effect when the VI is running. The primary use of the To More Generic Class function is as a way of eliminating coercion dots for those programmers who have a strict code convention against such data conversions.

Using the To More Specific Class function leaves the data unchanged unless an error occurs. If the data on the wire at run time is not of the more specific class, the function returns an error and the output data is the default value of the wire type. The primary use of the To More Specific Class function is to do type testing on a value of a parent class. Frequently programmers connect the same parent class wire to several To More Specific Class function calls, each going to a different more specific class, and then execute different code based on which To More Specific Class function does not return an error. This is highly inefficient. If you are considering this method of testing and have permission to modify the parent class, you might [write a dynamic member VI](#) on the parent class and have each of the child classes override it with the functionality appropriate for each child class.

If, at run time, LabVIEW detects that you wired a child class object to a subVI that accepts a parent class object, LabVIEW can downcast the subVI output to a child class object automatically. Automatic downcasting does not require you to use the To More Specific Class function. However, automatic downcasting can occur only if LabVIEW can guarantee that the class object you wire to the subVI is compatible with the input the subVI accepts. For example, if you store a class object in a variant and wire the variant data to a subVI, LabVIEW cannot guarantee that the subVI contains data that belongs to the same class as the data the variant originally stored. To help LabVIEW check that the class object you wire to the subVI is compatible with the class object the subVI accepts, use the Preserve Run-Time Class function. If the two objects are incompatible, the function returns an error and sets the class of the output data to the parent class the subVI accepts. You also can use this function with data value references. The Data Value Reference Read / Write Element border node must preserve run-time type. You can use the Preserve Run-Time Class function to

check that the class object you wire to the Data Value Reference Write node is compatible with the class object you wired to the Data Value Reference Read border node.



Note LabVIEW cannot automatically downcast recursive subVIs. Automatic downcasting breaks caller VIs if they call recursive subVIs on the block diagram of dynamic dispatch VIs or inside In Place Element structures for data value references. Use the Preserve Run-Time Class function on the block diagram of the recursive subVI to check that the class object you wire to the subVI is compatible with the class object the subVI accepts. If the class objects are compatible, the caller VIs outside of the recursive cycle do not break.

Locking and Unlocking LabVIEW Classes

You can [lock LabVIEW classes](#) to prohibit LabVIEW class users from viewing the member VIs that are part of the internal implementation of the application. Locking the LabVIEW class prevents access to the private member VIs which in turn can help prevent errors being introduced to the application. As with [project libraries](#), adding password protection to LabVIEW classes does not add password protection to the member VIs. You must password protect the member VIs individually.

If you lock a LabVIEW class, only the data type appears in the Context Help window when you move the cursor over the class wire. If you move the cursor over the wire of an unlocked LabVIEW class, the **Context Help** window shows the private data of the class and the private data of the class for all unlocked parent classes. Similar rules apply to the information LabVIEW displays in generic probes. Refer to [Using Probes in LabVIEW Classes](#) for information about using generic probes with LabVIEW classes.



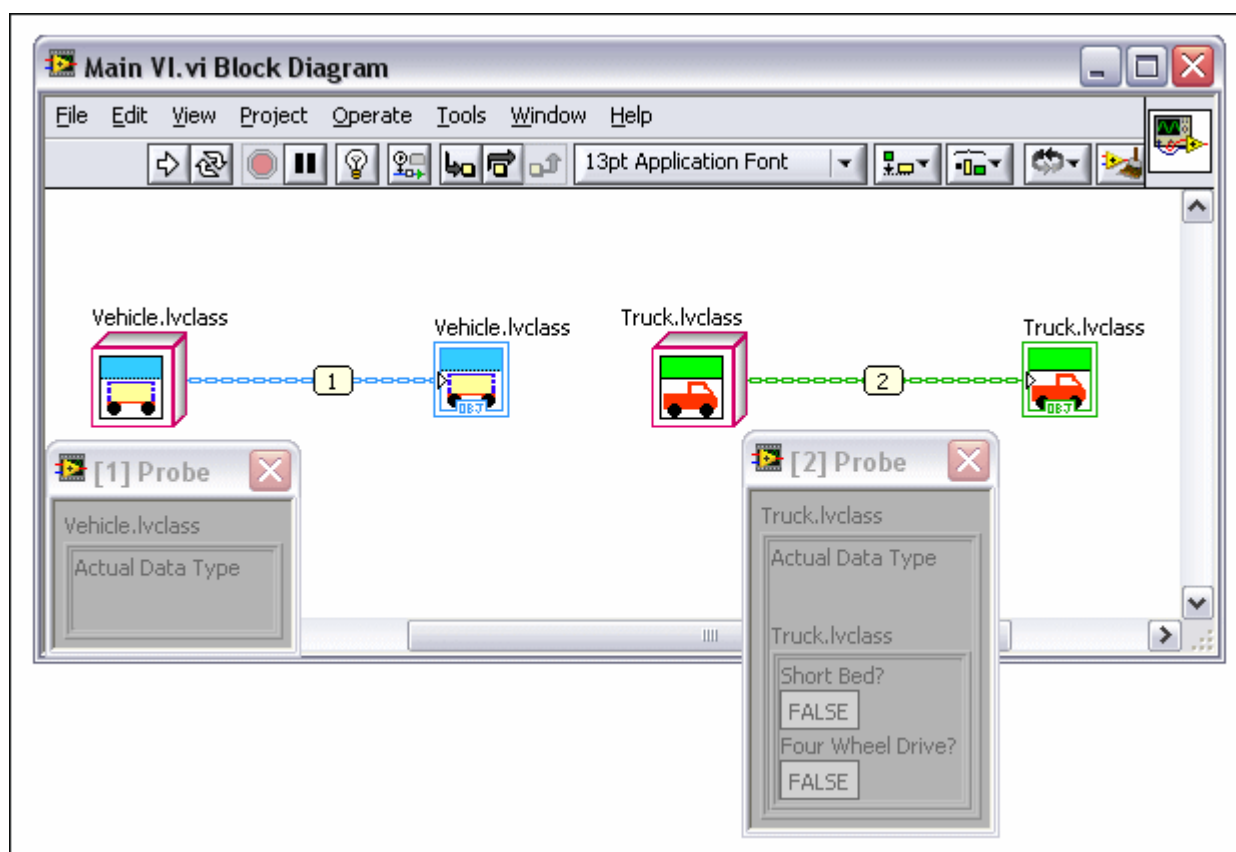
Note Notice the Context Help window for the LabVIEW class wire and the LabVIEW class control display different information. The LabVIEW class control displays class description information for end users, while the LabVIEW class wire displays additional information about the data type for the LabVIEW class user.

Because LabVIEW class users only can view the class information you make available, be sure to adequately document the LabVIEW class you develop. You can [add class descriptions](#) in the Class Properties dialog box and you can [add member VI descriptions](#) in the VI Properties dialog box.

Using Probes in LabVIEW Classes

You can use [generic probes and custom probes](#) to debug LabVIEW classes.

You can use generic probes on LabVIEW class wires. The generic probe on a LabVIEW class wire shows the name of the class at run time, the private class data, and the private class data for all of the parent classes. If you use the generic probe on a locked LabVIEW class, only the name of the class at run time and any unlocked private data within the class hierarchy is visible in the probe. In the following image, class Vehicle, on the left, is locked and LabVIEW does not display the private data of the class: **Number of Gears, Number of Doors, Color, Make, and Model**. Class Truck is not locked and LabVIEW does display its private data. Notice that LabVIEW does not display the inherited private data of class Vehicle in the generic probe for class Truck because class Vehicle is locked.



Creating custom default probes for LabVIEW classes is useful. You only can use a custom probe on a LabVIEW class wire of the exact class type as the wire type or of an ancestor of the wire type. A LabVIEW class developer can create custom probes that are members of the class and capable of displaying the private data of the class directly. A LabVIEW class user can create a custom default probe that may display any information available using public methods of the class.

A class developer can **set a custom probe as the default probe** for a LabVIEW class user. The generic probe of a locked LabVIEW class does not allow the LabVIEW class user to access any information about the value of the data because the data is private. Creating custom probes and setting them as the default probe for a LabVIEW class allows the class developer to provide the LabVIEW class user with a default probe that displays the appropriate data after the class developer locks the class. The custom probe must be a member of the LabVIEW class for the LabVIEW developer to set it as the default probe.

Distributing a LabVIEW Class to other Developers and Users

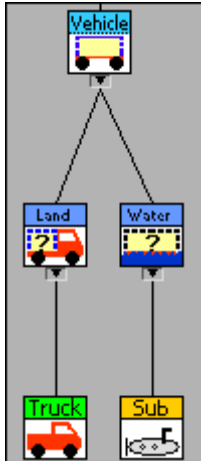
You can distribute the LabVIEW class you develop to other LabVIEW class developers and LabVIEW class users. You can distribute the class in several ways so choose the manner that most suits your needs. You can use the **Application Builder** to **create a zip file** to distribute the class or classes. You also can **lock the LabVIEW class** before you distribute it to limit the access the LabVIEW class user has to the private data and member VIs. Locking the class can help prevent users from introducing errors in the application.



Tip You also can access LabVIEW classes through a .NET Framework assembly. LabVIEW can generate a .NET class for LabVIEW classes that you specify, and you can access these .NET classes through a .NET assembly.

Comparing LabVIEW Class Objects

Use the Comparison functions to compare LabVIEW class objects.



If you compare two objects of the same class, for example two objects of the Truck class, LabVIEW compares the data at all levels of the class hierarchy, similar to the manner in which LabVIEW compares a cluster of clusters.

If you compare two objects of different classes, for example an object of the Truck class and an object of the Submarine class, LabVIEW compares the data for as many ancestor classes as the objects have in common. In this case, LabVIEW compares the data of the Vehicle class, the only common ancestor class.

When comparing two objects of different classes, if all data in the common ancestor class or classes is the same, LabVIEW orders the objects based on the following criteria.

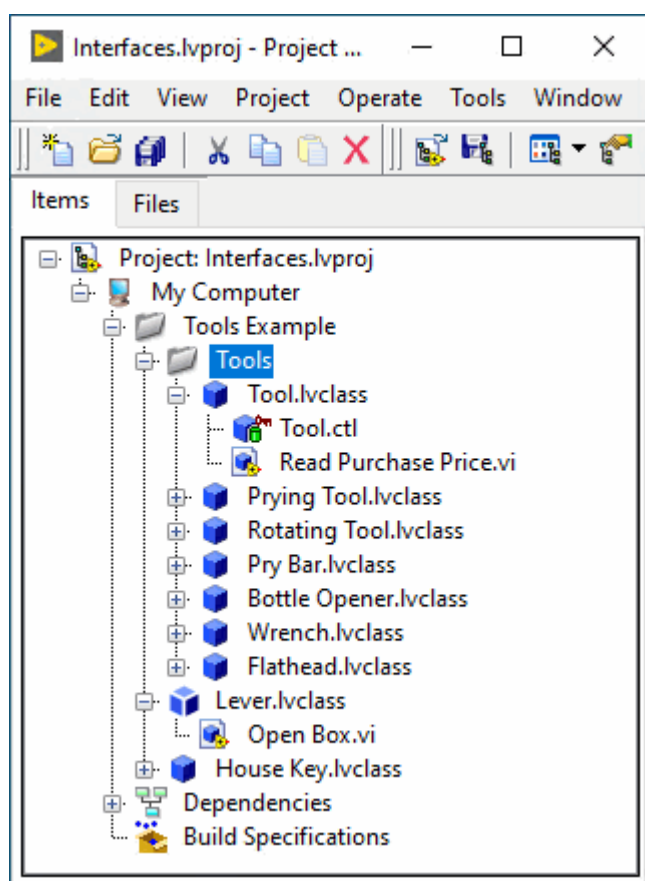
1. LabVIEW orders parent classes before child classes. For example, if you compare an object of the Vehicle class with an object of the Land Vehicle class, LabVIEW orders the object of the Vehicle class first because it is an object of the parent class.
2. LabVIEW orders first differing classes alphabetically by the name of the class. For example, if you compare an object of the Truck class and an object of the Submarine class, LabVIEW compares the data of the Vehicle class object first, as the common ancestor class. If the data of the Vehicle class object is the same, LabVIEW sorts the data of the Truck object first because of the first differing classes (Land Vehicle and Water Vehicle), Land Vehicle takes priority alphabetically.

LabVIEW Interfaces

In LabVIEW, an interface can be thought of as a class without a private data control, but that small difference enables an interface to serve entirely different purposes in software architectures than classes. Specifically, interfaces enable a form of multiple inheritance.

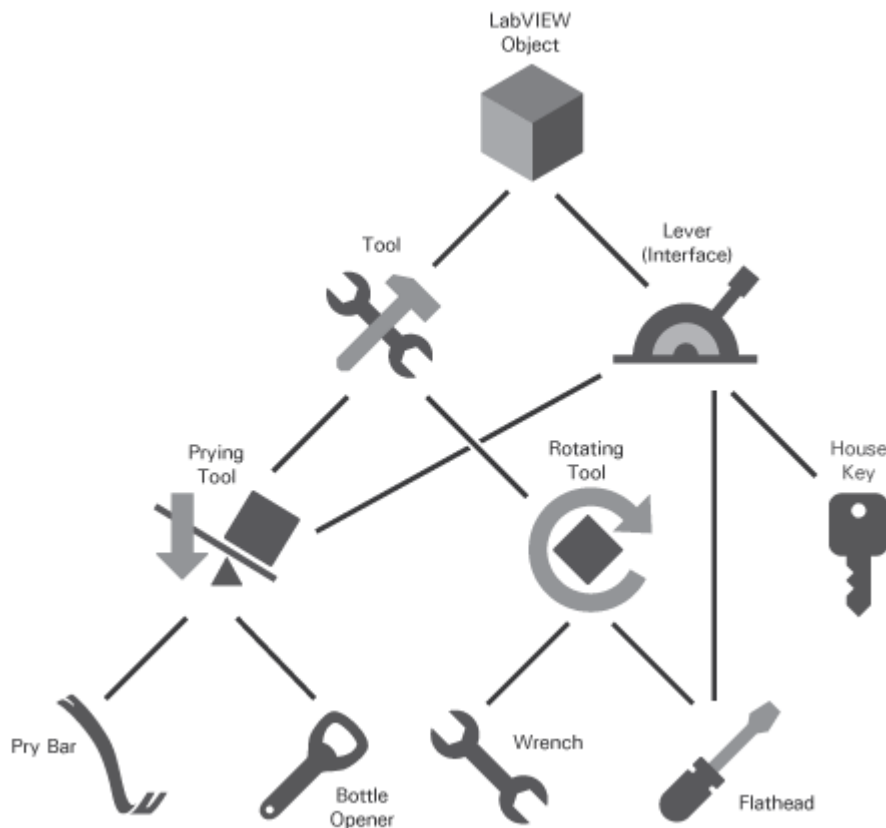
An interface declares a role that an object can play without defining how to perform that role. By inheriting from an interface, a class declares that its objects fulfill that role, and the class becomes responsible for specifying how the behaviors are performed. When a class inherits from multiple interfaces, its objects can be passed into multiple software modules requiring different roles.

The following project includes the Tool class with several child classes of different tools. The project also includes the Lever interface. You can see that the Tools class has a control that defines the data in the class, whereas the Lever interface does not have a control because interfaces do not have private data. A class is represented by a solid cube . An interface is represented by the faces of a cube . Interfaces and classes use the same file extension .lvclass.



The following image illustrates the inheritance relationship among the Tool classes, the House Key class, and the Lever interface. The Tool class and the Lever interface both inherit from LabVIEW Object. In addition to having their own methods and methods of their respective parent classes, the Prying Tool class and the Flathead class also inherit from the Lever interface because they both can be used as a lever. They inherit the methods of the Lever interface. Since Lever is an interface rather than a class, this multiple inheritance is

legal. The three unrelated classes (Prying Tool, Flathead, House Key) have a common ancestor (Lever) other than LabVIEW Object.



Best Practices for Naming Interfaces

Use the following guidelines to provide the interface you create with a name that identifies the purpose of the interface:

- Use an adjective or adverb that describes the capability that an object has if the object inherits the interface. For example, name an interface `Can Measure Voltage.lvclass` if the interface represents hardware that can measure voltage. Any class or interface that inherits that interface is capable of measuring voltage.
- Use a noun that describes the category of classes that inherit from the interface. For example, name an interface `Lever.lvclass` if the interface describes a category of classes that represent levering tools.
- Avoid using a leading capital letter "I". Although most text programming languages commonly name interfaces with a leading capital letter "I" to differentiate interfaces from classes, LabVIEW distinguishes interfaces and classes using glyphs. Additionally, most parts of the LabVIEW development environment intentionally treat classes and interfaces identically. Callers of a method generally do not care whether the underlying type is an interface or a class. Therefore, avoiding the "I" enables you to convert a class to an interface or vice versa without refactoring caller code.

Refer to the following projects for examples of using interfaces:

- labview\examples\Object-Oriented Programming\Basic Interfaces\Basic Interfaces.lvproj
- labview\examples\Object-Oriented Programming\Actors and Interfaces\Actors and Interfaces.lvproj

Using LabVIEW Classes and Interfaces in an Application

LabVIEW classes and interfaces are user-defined data types. LabVIEW class and interface developers create and distribute these data types. The LabVIEW class or interface user does not need to know how to create a LabVIEW class or interface, but rather how to use the data type a class or interface defines in an application, what information is available to them in debugging code that uses LabVIEW classes or interfaces, and how a new version of the LabVIEW class or interface might affect the application the programmer has built.

Creating an Instance of a Class or Interface

The LabVIEW class or interface does not need to be in the project. You can [add the LabVIEW class or interface to the palette](#) so you can easily find and drag the class or interface to the block diagram or front panel window. To create an instance of the LabVIEW class or interface you receive or [create](#), place the class library file `.lvclass` as a control or constant on the front panel or block diagram on which you want to use it. You can use a LabVIEW user-defined data type similarly to other [LabVIEW data types](#).

Using the Class Browser window, you can [place member VIs](#) of the LabVIEW class or interface on the block diagram.

Coercing LabVIEW Classes and Interfaces

LabVIEW coerces classes and interfaces to the nearest common parent class or interface with the following rules:

- If the input classes and interfaces are unrelated, LabVIEW coerces the inputs to LabVIEW Object.
- If the input classes and interfaces have only one common parent, whether it is a class or interface, LabVIEW coerces the inputs to the common parent.
- If the input classes and interfaces have multiple common parents, among which one is a class, LabVIEW coerces the inputs to the common parent class. To force LabVIEW to coerce the inputs to a common parent interface, cast one of the inputs to the desired parent type using the To More Generic Class function.
- If the input classes and interfaces have multiple common parents that are all interfaces, LabVIEW breaks the VI. To force LabVIEW to coerce the inputs to a common parent interface, cast one of the inputs to the desired parent type using the To More Generic Class function.

Using Context Help and Probes with Locked LabVIEW Classes or Interfaces

The LabVIEW class or interface developer may *lock the LabVIEW class or interface* when he or she distributes the class or interface. If the LabVIEW class or interface developer locks the LabVIEW class or interface, the Context Help window only displays the data type of the class or interface and the description of the class if the developer provides one. LabVIEW also only displays the data type of the class or interface in the *generic probe on a locked class or interface*. You can create a *non-member custom probe for a LabVIEW class or interface*, however the non-member custom probe only can display the public interface of the class or interface. The LabVIEW class or interface developer also can set a *custom default probe* for the class or interface to provide specialized class or interface information for debugging. Work with the LabVIEW class or interface developer to provide the appropriate amount of documentation or probe information necessary for working with the LabVIEW class or interface.

Dynamic Dispatch VIs

Some member VIs of a LabVIEW class or interface can be dynamic dispatching. Dynamic dispatch methods are similar to polymorphic VIs. Where polymorphic VIs decide which VI to call depending upon the data type you wire to it, dynamic dispatch methods wait until run time to decide which member VI in the class hierarchy to call depending on the data that arrives at the input. Because a LabVIEW class or interface wire can carry data of its own type or data of any child type, LabVIEW determines which member VI in the class hierarchy to execute at run time. You can double-click dynamic dispatch member VI on the block diagram to view the particular VI that is on the block diagram. To view the other dynamic dispatch member VIs in the class hierarchy, you must view each individual VI in the class hierarchy.

Mutating Data

Tracking the version number of the LabVIEW class or interface is beneficial to LabVIEW class users who develop applications using LabVIEW classes or interfaces. For example, you have a LabVIEW class in an application that uses an unsigned 32-bit integer in the private data control of the class. A LabVIEW class developer can send you a version of the LabVIEW class that changes the numeric control in the private data control to a double-precision floating-point number. Because LabVIEW tracks version changes and knows how to flatten and unflatten data on all LabVIEW classes, you can replace the old version of the LabVIEW class with the new version and run the program without making changes to it.



NOTE

If you try to unflatten data from a future version of a LabVIEW class, LabVIEW returns an error. For example, this error might occur if you have data with a version number of 1.0.0.3 but the LabVIEW class in memory is version 1.0.0.2.

LabVIEW Front Panel

Front Panel Controls and Indicators

Use the following subpalettes to choose controls and indicators to build the front panel for your LabVIEW application.



NOTE

The availability of front panel controls and indicators may vary depending on the subpalette style you choose. Refer to the Modern, Silver, System, or Classic subpalette for front panel control and indicator availability.

Subpalette	Description
Data Containers	Use array and cluster controls and indicators to group data elements in unsorted order. Use set or map controls and indicators to group unique data elements in sorted order. Use matrix controls and indicators to enter and display matrix data.
Boolean	Use Boolean controls and indicators to enter and display TRUE/FALSE value with buttons, switches, and lights.
Containers	Use container controls and indicators to group other controls and indicators or to display the front panel of another VI on the front panel of the current VI.
Decorations	Use decorations to graphically group or separate front panel objects without affecting the functionality of the VI. Decorations include objects such as boxes, lines, or arrows.
Graph	Use graph and chart indicators to plot and display numeric data.
I/O	Use I/O name controls and indicators to pass names associated with I/O hardware configurations to I/O Vis to communicate with an instrument or a device.
List, Table & Tree	Use list and tree controls and indicators to give users lists of items from which to select. Use tables to display strings of data in columns and rows.
Numeric	Use numeric controls and indicators to enter and display numeric data.
Refnum	Use refnum controls and indicators to reference files, directories, devices, and network connections.
Ring & Enum	Use ring and enumerated controls and indicators to create a list of items for the end user to select from. The ring controls and indicators support only the numeric data type. The enumerated controls and indicators support the numeric data type with their corresponding strings included.

Subpalette	Description
String & Path	Use string controls and indicators to enter or display text. Use path controls and indicators to enter or return file or directory paths.
Variant & Class	Use variant and class controls and indicators to interact with variant and object-oriented data.



RELATED INFORMATION

- [Container Controls and Indicators](#) on page 124
- [Boolean Controls and Indicators](#) on page 122
- Use Boolean controls and indicators located on the Boolean subpalettes to enter and display Boolean (TRUE/FALSE) values with objects such as buttons, switches, and LED lights.
- [Container Controls and Indicators](#) on page 124
- [Graph Subpalette](#) on page 151
- [I/O Name Controls and Indicators](#) on page 153
- [List, Table, and Tree Controls and Indicators](#) on page 157
- [Numeric Controls and Indicators](#) on page 159
- [Refnum Controls and Indicators](#) on page 160
- [Ring and Enumerated Type Controls and Indicators](#) on page 161
- [String and Path Controls and Indicators](#) on page 162
- [Variant and Class Controls and Indicators](#) on page 164
- [Showing and Hiding Palette Categories](#) on page 845
- [Creating Custom Controls, Indicators, and Type Definitions](#) on page 165

Building the Front Panel

The front panel is the user interface of a VI. Generally, you design the front panel first and then design the block diagram to perform tasks on the inputs and outputs you create on the front panel.

You build the front panel using controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input mechanisms. Indicators are graphs, LEDs, and other output displays. Controls simulate instrument input mechanisms and supply data to the block diagram of the VI. Indicators simulate instrument output mechanisms and display data the block diagram acquires or generates.

Select **View»Controls Palette** to display the Controls palette and then select controls and indicators from the Controls palette and place them on the front panel.



RELATED INFORMATION

[Building the Block Diagram](#) on page 126

[Front Panel Controls and Indicators](#) on page 589

[Front Panel Controls and Indicators](#) on page 589

[Configuring Front Panel Objects](#) on page 591

[Configuring the Front Panel](#) on page 593

[Controls Subpalette](#) on page 152

[Designing User Interfaces](#) on page 598

[Creating Custom Controls, Indicators, and Type Definitions](#) on page 165

[Labeling](#) on page 1363

[Creating XControls](#) on page 1078

[Dragging and Dropping in LabVIEW](#) on page 125

Configuring Front Panel Objects

Use **Properties** dialog boxes or shortcut menus to configure how constants, controls, indicators, and Numeric functions appear or behave. Use **Properties** dialog boxes when you want to configure a constant, control, indicator, or Numeric function using a dialog box that includes context help and in which you can set several properties at the same time for an object. Use [shortcut menus](#) to quickly configure common constant, control, indicator, and function properties. The options available in **Properties** dialog boxes and shortcut menus differ for different objects. Most options you set using a shortcut menu are reflected in the **Properties** dialog box, and most options you set using the **Properties** dialog box are reflected in the shortcut menu.

Right-click a constant, control, indicator, or Numeric function and select **Properties** from the shortcut menu to display the **Properties** dialog box for that object. You cannot display **Properties** dialog boxes while a VI runs.



TIP

To perform a common operation on multiple objects simultaneously, select the objects and use the shortcut menu or the Properties dialog box.

You also can [create a custom control or indicator](#) to extend the available set of front panel objects. Right-click the control and select **Advanced»Customize** from the shortcut menu to customize a control or indicator. You can save a custom control or indicator you created in a directory or LLB and use the custom control or indicator on other front panels.

Showing and Hiding Optional Elements

Front panel controls and indicators have optional elements you can show or hide, such as labels, captions, and digital displays. Set the visible elements for the control or indicator on the **Appearance** page of the *Properties* dialog box for the front panel object. You also can set the visible elements by right-clicking an object, selecting **Visible Items** from the shortcut menu, and selecting among the available options.

Changing Controls to Indicators and Indicators to Controls

LabVIEW initially configures objects in the Controls palette as controls or indicators based on their typical use. For example, if you place a toggle switch on the front panel, it appears as a control because a toggle switch is usually an input mechanism. If you place an LED on the front panel, it appears as an indicator because an LED is usually an output device.

Some palettes contain a control and an indicator for the same type or class of object. For example, the **Numeric** palette contains a numeric control and a numeric indicator because you can have a numeric input or a numeric output.

You can change a control to an indicator by right-clicking the object and selecting **Change to Indicator** from the shortcut menu, and you can change an indicator to a control by right-clicking the object and selecting **Change to Control** from the shortcut menu.

Replacing Front Panel Objects

You can replace a front panel object with a different control or indicator. When you right-click an object and select **Replace** from the shortcut menu, a temporary Controls palette appears. Select a control or indicator from the temporary Controls palette to replace the current object on the front panel.

Selecting **Replace** from the shortcut menu preserves as much information as possible about the original object, such as its name, description, default data, dataflow direction (control or indicator), color, size, and so on. If you replace a numeric terminal with another numeric terminal, LabVIEW tries to preserve the original representation. However, if the control does not support the new data type, the new object retains its own data type. Wires from the terminal of the object or local variables remain on the block diagram, but they might be broken. For example, if you replace a numeric terminal with a string terminal, the original wire remains on the block diagram, but is broken.

The more the new object resembles the object you are replacing, the more original characteristics you can preserve. For example, if you replace a slide with a different style slide, the new slide has the same height, scale, value, name, description, and so on. If you replace the slide with a string control instead, LabVIEW preserves only the name, description, and dataflow direction because a slide does not have much in common with a string control.

You also can **Edit»Copy** and **Paste** to copy objects to the clipboard and paste them from the clipboard to replace existing front panel controls and indicators. This method does not preserve any characteristics of the old object, but the wires remain connected to the object.

Configuring the Front Panel

You can customize the front panel by setting the tabbing order of front panel objects, by using imported graphics, by setting front panel objects to automatically resize when the window size changes, and so on.

Setting Keyboard Shortcuts for Controls

You can assign keyboard shortcuts to controls so *users can navigate* the front panel without a mouse. Right-click the control and select **Advanced»Key Navigation** from the shortcut menu to display the **Key Navigation** page of the *Properties* dialog box.



NOTE

LabVIEW does not respond to keyboard shortcuts for hidden controls.

When a user presses a keyboard shortcut while a VI runs, the associated control receives the focus. If the control is a text or numeric control, LabVIEW highlights the text so you can edit it. If the control is Boolean, press the spacebar, the <Enter>, or the <Enter> key on the numeric keypad to change its value.

The **Advanced»Key Navigation** shortcut menu item is dimmed for indicators because you cannot enter data in an indicator.



Note You also can use the Key Down event to generate an event when the user presses a specific key on the keyboard.

Controlling Button Behavior with Key Navigation

You can associate function keys with various buttons that control the behavior of a front panel.



Note (macOS) Keyboard shortcuts for the function keys work only if the function keys are not assigned as shortcut keys for Exposé. LabVIEW does not support <VolumeUp>, <VolumeDown>, and function keys <F15> to <F24> as shortcut keys on macOS.

You can configure a button in a VI to behave like a dialog box button so that pressing the <Enter> key on the alphanumeric keyboard or the <Enter> key on the numeric keypad is the same as clicking the button. If you associate either <Enter> key with a dialog box button, LabVIEW automatically draws that button with a thick border around it.

If you navigate to a Boolean control and press the <Enter> key, the Boolean control changes, even if another control uses the <Enter> key as its keyboard shortcut. The assigned <Enter> keyboard shortcut applies only when a Boolean control is not selected.

Setting the Tabbing Order of Front Panel Objects

Controls and indicators on a front panel have an order, called tabbing order, that is unrelated to their position on the front panel. LabVIEW determines tabbing order based on the order in which you place controls on the front panel. The first control or indicator you create on the front panel is element 0, the second is 1, and so on. If you delete a control or indicator, the tabbing order adjusts automatically.

The tabbing order determines the order in which LabVIEW selects controls and indicators when the user presses the <Tab> key while a VI runs. The tabbing order also determines the order in which the controls and indicators appear in the records of datalog files you create when you [log the front panel data](#).

You can [set the tabbing order](#) of front panel objects by selecting **Edit»Set Tabbing Order**.

To prevent users from accessing a control by pressing the <Tab> key while a VI runs, place a checkmark in the **Skip this control when tabbing** checkbox in the **Key Navigation** page of the [Properties](#) dialog box.



Note If you add a splitter bar to the front panel and [create panes](#) the tabbing order of objects on the front panel does not change. A control in any pane can be the next control in the tabbing order.

Coloring Objects

You can change the color of many objects in LabVIEW. You can change the color of most front panel objects, panes on the front panel, and block diagram workspaces. You cannot change the color of system controls and indicators because these objects appear in the colors you have set up for the system.

Use the Coloring tool to right-click an object or workspace to change the color of front panel objects, panes on the front panel, and block diagram workspaces. You also can change the default colors for some objects by selecting **Tools»Options** and selecting Environment from the **Category** list.

Refer to the following guidelines when using color on a front panel:

- Color can distract the user from important information. For example, a yellow, green, or bright orange background makes it difficult to see a red danger light.
- Some platforms provide limited colors. Black, white, and gray are often the best options for maintaining consistency on other platforms.
- Avoid using color as the sole indicator of device state. Use line styles for plots to add another differentiating factor because users with some degree of color blindness may have problems detecting the change in state.
- Use light gray, white, or pastel colors for backgrounds. Use the first row of colors in the color picker for front panel backgrounds and normal controls. Use the second row of colors in the color picker for highlighting important controls, such as an error notification.

Using Imported Graphics

You can import graphics from other applications to use as pane backgrounds, items in ring controls, and parts of other controls and indicators.

LabVIEW supports most standard graphic formats, including BMP, JPEG, animated GIF, MNG, animated MNG, and PNG. LabVIEW also supports transparency.

However, imported graphics might slow down screen updates. Make sure you do not place indicators and controls on top of a graphic object so that LabVIEW does not have to redraw the object each time the indicator updates.



Tip If you must use a large background picture with controls on top of it, divide the picture into several smaller objects and import them separately because large graphics usually take longer to draw than small ones.

Use one of the following ways to import a graphic.

Windows

macOS

Linux



Tip Check how the imported graphics look when you load the VI on [another platform](#). For example, a macOS PICT file that has an irregular shape might convert to a rectangular bitmap with a white background on Windows or Linux.

After you import a graphic, you might need to resize or reorder the image. You cannot resize imported GIFs.

Importing Graphics to the Front Panel

You can import graphics for front panel backgrounds using either of the following ways.

- Right-click the scroll bar of a pane and select **Properties** from the shortcut menu to display the **Pane Properties** dialog box. On the Background tab, use the **Background**, **Browse**, and **Position** options to [set a pane background image](#).
- Copy the graphic to the clipboard and paste it on the front panel. You also can select **Edit»Import Picture to Clipboard** and paste the graphic on the front panel.



Note (Windows and macOS) If you import an image by copying and pasting it, the image loses any transparency.

Aligning and Distributing Objects

Select **Edit»Enable Panel Grid Alignment** to enable the grid alignment on the front panel and *align objects* as you place them. Select **Edit»Disable Panel Grid Alignment** to disable the grid alignment and use the visible grid to align objects manually. You also can press the <Ctrl-#> keys to enable or disable the grid alignment. On French keyboards, press the <Ctrl-”> keys.

(macOS) Press the <Command-*> keys. (Linux) Press the <Alt-#> keys.

You also can use the alignment grid on the block diagram. When objects are aligned and distributed evenly, you can use straight wires to wire the objects together. This makes the block diagram easier to read.

Select **Tools»Options** and select Front Panel from the **Category** list to hide or customize the grid.

To align objects after you place them, select the objects and select the **Align Objects** pull-down menu on the toolbar or select **Edit»Align Items**. To space objects evenly, select the objects and select the **Distribute Objects** pull-down menu on the toolbar or select **Edit»Distribute Items**.

Grouping and Locking Objects

Use the Positioning tool to select the front panel objects you want to group and lock together. Click the **Reorder** button on the toolbar and select **Group** or **Lock** from the pull-down menu. Grouped objects maintain their relative arrangement and size when you use the Positioning tool to move and resize them. Locked objects maintain their position on the front panel. You cannot delete grouped or locked objects from the front panel or the block diagram until you ungroup or unlock them. You can set objects to be grouped and locked at the same time. If you set objects to be grouped and locked at the same time, you must ungroup and unlock them before you can delete them from the front panel or the block diagram. Tools other than the Positioning tool work normally with grouped or locked objects.

Resizing Objects

You can change the size of most front panel objects. When you move the Positioning tool over a resizable object, resizing handles appear at the points where you can resize the object. When you resize an object, the font size remains the same. Resizing a group of objects resizes all the objects within the group.

Some objects change size only horizontally or vertically when you resize them, such as digital numeric controls and indicators. Others keep the same proportions when you resize them, such as knobs. The Positioning cursor appears the same, but the dashed border that surrounds the object moves in only one direction.

You can manually restrict the growth direction when you resize an object. To restrict the growth vertically or horizontally or to maintain the current proportions of the object, press the <Shift> key while you click and drag the resizing handles. To resize an object around its center point, press the <Ctrl> key while you click and drag the resizing handles. To restrict

the growth vertically or horizontally or to maintain the current proportions of the object and to resize an object around its center point, press the <Ctrl-Shift> keys while you click and drag the resizing handles.

(macOS) Press the <Option> key in place of the <Ctrl> key. (Linux) Press the <Alt> key in place of the <Ctrl> key.

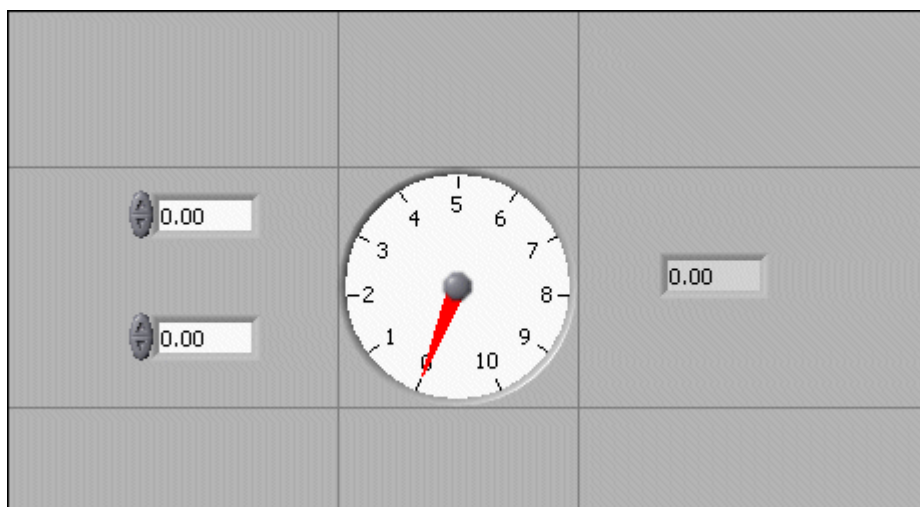
To resize multiple objects to the same size, select the objects and select the **Resize Objects** pull-down menu on the toolbar. You can resize all the selected objects to the width or height of the largest or smallest object, and you can resize all the selected objects to a specific size in pixels. If you add splitter bars to the front panel and create panes, you can select different objects on multiple panes by pressing the <Shift> key and selecting the objects you want to resize.

Scaling Front Panel Objects

You can set front panel objects to scale, or automatically resize in relation to the pane size, when you resize the pane. You can set one object on the front panel to scale, or you can set all objects on an owning pane to scale. However, you cannot set multiple objects to scale on the front panel unless you set all of them to scale, you group the objects first, or the objects are on separate panes. You also can set a single object to fill a pane.

To set a single object to scale automatically when the pane resizes, right-click the object and select **Scale Object with Pane** from the shortcut menu. If you set a single front panel object to scale, the object resizes itself automatically in relation to any change in the pane size. The other objects on the front panel reposition themselves to remain consistent with their previous placement on the front panel but do not scale to fit the new size of the pane.

Immediately after you designate a single object on the front panel to scale automatically, gray lines outline several regions on the front panel, as shown in the following front panel. The regions define the positions of the other front panel objects in relation to the object you want to scale. When you resize the pane, the object you set to scale automatically resizes and repositions itself relative to its original location. The gray lines disappear when you run the VI.



To set objects in an owning pane to scale automatically while the pane resizes, right-click a splitter bar and select **Pane Sizing»Scale Objects While Resizing** from the shortcut menu.

An item that scales on the pane will scale while you are resizing the pane instead of when you are finished resizing the pane. To set all objects in an owning pane to scale automatically when the pane resizes, right-click a splitter bar and select **Pane Sizing»Scale all Objects with Pane** from the shortcut menu of the appropriate pane. If you set all objects on a pane to scale, all objects resize and reposition automatically in relation to any change in the pane size. If you set all objects on a pane to scale and a single object on the pane is already set to scale, a dialog box appears that prompts you to select between setting automatic scaling for the one object or for all objects on the pane. You also can use the Set Scaling Mode method to set all the pane's objects to scale programmatically.

When LabVIEW scales objects automatically, it follows the same conventions as when you resize an object manually. For example, some objects can resize only horizontally or vertically, and the font size remains the same when you resize an object.

After LabVIEW automatically scales an object, the object might not scale back to its exact original size when you size the pane back to its original position. Before you save the VI, select **Edit»Undo** to restore the original front panel window and object sizes.

You can set an array to scale or set the objects within an array to scale. If you set the array to scale, you adjust the number of rows and columns you can see within the array. If you set the objects within the array to scale, you always see the same number of rows and columns, though different sizes, within the array.

You also can set a cluster to scale or set objects within the cluster to scale. If you set a cluster to scale, the objects within the cluster do not scale. If you set objects within the cluster to scale, the cluster does not scale unless you right-click the cluster and select **AutoSizing»Size to Fit** from the shortcut menu.

To set a single object to fill the pane, right-click the object and select **Fit Control to Pane** from the shortcut menu. The object should be the only object in the pane, otherwise other objects are overlapped by the object you select to fit to the pane. The object resizes itself to fill the pane, and automatically resizes in relation to any change in the pane size.

Adding or Reducing Space on the Front Panel without Resizing the Window

You can [add or reduce space](#) on the front panel without resizing the window.

Designing User Interfaces

If a VI serves as a user interface or dialog box, front panel appearance and layout are important. [Design front panels](#) to resemble instruments or other devices so users easily can identify which actions to perform. Use front panel controls and indicators, splitter bars and panes, window settings, and so on to improve the usability of the front panel. You also can [use events](#) to enhance the functionality of user interfaces.

Designing Front Panels

The main components of the front panel are controls and indicators. Use the following guidelines to design the front panel to serve as a user interface:

- Consider how users interact logically with the VI. Group controls and indicators accordingly.
- Place related controls in clusters or add decorative borders around them.
- Use the decorations located on the [Decorations](#) palette to group or separate objects with boxes, lines, or arrows.
- Leave blank space between objects to make the front panel easier to read. Blank space also prevents users from accidentally clicking the wrong control or button.
- Avoid placing objects on top of other objects. Partially covering a control or indicator with a label or other object slows down screen updates and can make the control or indicator flicker.
- Assign specific names to buttons such as Start, Stop, and Save. Use common terminology.
- Use the **Align Objects** and **Distribute Objects** pull-down menus to create a uniform layout.

Configuring Fonts and Colors

You also can improve the usability of user interfaces by configuring fonts and colors. LabVIEW defines its [built-in fonts](#) as the default system fonts. When you move VIs between platforms, LabVIEW automatically updates its built-in fonts so that they match the default system fonts of the current platform. Additionally, if you attempt to open a VI that uses an unavailable font, LabVIEW substitutes the closest match. LabVIEW also substitutes colors similarly to how it substitutes fonts. If one of the colors of the VI is unavailable, LabVIEW replaces it with the closest match. The system colors vary across computers, so any VI that uses system colors reflects the specific system colors of the current computer. The system colors also define the appearance of Panel & Object, Window, and Highlight, and the color of the text that appears on top of these system colors. You can [configure user-defined colors and view system colors](#) in the Options dialog box.

Using Splitter Bars and Panes

You can use splitter bars, such as [toolbars](#) or [status bars](#), to create professional user interfaces in the front panel window. You can [create a splitter bar](#) to separate the front panel into multiple regions, called panes. Each pane acts similarly to a unique front panel, with its own sets of pane coordinates and controls and indicators on it. You can scroll each pane individually. The splitter bar separates the controls on one pane from those on another pane, yet the terminals for all controls are on the same block diagram.

When you create a new, blank VI, the front panel has a single pane that fills the window. The front panel owns this pane and is the pane's parent. When you place a splitter bar on a pane, the splitter bar replaces the pane in the front panel object hierarchy and creates two new child panes. The front panel owns the splitter bar, and the splitter bar owns the two child panes. If you place a new splitter bar on one of the child panes, the new splitter bar replaces that pane and becomes the parent of the two new child panes. This hierarchy forms a binary tree, and the front panel owns the top pane.

Refer to the Resizable Panel with Splitter Bars VI in the `labview\examples\Controls and Indicators\Containers` for an example of using splitter bars and panes.

Configuring Splitter Bars

Right-click a splitter bar and select one of the following shortcut menu options to configure splitter bars:

- **Visible Items** —Adds the label for the splitter bar to the front panel. LabVIEW hides the splitter bar label by default.
- **Create** —Allows you to create a Property Node, VI Server reference, or Invoke Node to configure the splitter bar programmatically.
- **Remove Entire Splitter Hierarchy** —Removes the splitter bar you select and any splitter bars that are children of the splitter bar in the hierarchy. The objects remain in the same screen position on the front panel. If you select a splitter bar that is not another splitter bar's parent, LabVIEW disables this option.
- **Remove with Adjoining Splitters** —Removes the splitter bar you select and any splitter bars with ends that touch the selected splitter bar.
- **Splitter Sizing** —Allows you to control how resizing the window affects panes. By selecting an option in the **Splitter Sizing** shortcut menu, you can set a pane to remain the same size, or stick, while the other pane grows or shrinks as you resize the windows. You can also choose **Splitter Sizes Proportionally** to allow both panes to grow or shrink together when you resize the window.
- **Locked** —Locks the splitter bar in the current position.
- **Splitter Style** —Allows you to customize the style of the splitter bar.
- **Upper Pane/Lower Pane/Left Pane/Right Pane** —Allows you to configure the selected pane. When you select a pane, a new shortcut menu opens with options for configuring the selected pane. Refer to the [Configuring Panes](#) section of this topic for more information about configuring panes.

You also can use the Splitter properties to configure splitter bars programmatically.

Configuring Panes

Right-click a scroll bar in the pane, or right-click a splitter bar and select the pane you want to configure and select one of the following shortcut menu options to configure panes:

- **Visible Items** —Adds the label for the pane to the front panel. LabVIEW hides the pane label by default.
- **Create** —Allows you to create a Property Node, VI Server reference, or Invoke Node to configure the pane programmatically.
- **Horizontal Scrollbar** —Allows you to select if the horizontal scroll bar is always visible, never visible, or not visible while the VI is running.

- **Vertical Scrollbar** —Allows you to select if the vertical scroll bar is always visible, never visible, or not visible while the VI is running.



NOTE

If you turn off the horizontal or vertical scroll bar by right-clicking the scroll bar in the front panel window, you can display the scroll bar again by enabling the **Show horizontal scroll bar** option or the **Show vertical scroll bar** option in the Customize Window Appearance dialog box.

- **Pane Sizing** —Allows you to control how resizing the window affects panes. You can choose to have the objects in the resized pane remain at the bottom, top, left, or right of the pane. You also can choose to scale the objects in the pane when you resize the window.

If there are no splitter bars on the front panel, you can configure some properties of the single pane in the VI Properties dialog box. However, if you add a splitter bar to the front panel, LabVIEW limits the menu options in the **VI Properties** dialog box because the configurations cannot apply to all panes. You must configure each pane individually. For example, adding a splitter bar to the front panel dims the **Scale all objects on front panel as the window resizes** checkbox on the **Window Size** page of the **VI Properties** dialog box.

You also can use the Pane properties to configure panes programmatically.

Designing Dialog Boxes

Select **File»VI Properties**, select Window Appearance from the **Category** pull-down menu, and select the **Dialog** option to hide the menu bar and scroll bars and to create VIs that look and behave like standard dialog boxes for each platform.

If a VI contains consecutive *dialog boxes* that appear in the same screen location, organize them so that the buttons in the first dialog box do not directly line up with the buttons in the next dialog box. Users might double-click a button in the first dialog box and unknowingly click a button in the subsequent dialog box.

Users also might attempt to use the keyboard shortcut <Ctrl-Z> to perform an undo operation while a VI is running. This can cause unexpected behavior in dialog boxes, such as a closed dialog box reopening. To avoid this behavior, you can *create a custom menu* that does not include the **Edit»Undo** option, or you can add an Event structure to the VI and *configure* the Menu Selection (App) event to handle the command.

Use the system controls located on the **System** palette in dialog boxes you create.

Selecting the Screen Size

When you design a VI, consider *whether the front panel can display* on computers with different screen resolutions.

Consider making the window as small as possible without crowding controls or sacrificing a clear layout. Some displays have a limited resolution, especially LCD displays and touchscreens.

Also, configure front panels to open in the upper-left corner of the screen for the convenience of users with small screens. Place sets of VIs that are often opened together so the user can see at least a small part of each. Place front panels that open automatically in the center of the screen. Centering the front panels makes the VI easier to read for users on monitors of various sizes.

In **VI Properties**, select **Window Size** in the **Category** pull-down menu, and place a checkmark in the **Maintain proportions of window for different monitor resolutions** checkbox to maintain front panel window proportions relative to the screen resolution.

Controlling Front Panel Objects

Use the control refnum controls located on the **Refnum** and **Classic Refnum** palettes to pass front panel object references to other VIs. You also can right-click a front panel object and select **Create»Reference** from the shortcut menu to create a VI Server reference. Additionally, you can locate a front panel object from the VI Server reference constant on the block diagram. Right-click the reference on the block diagram and select **Find Control** from the shortcut menu. After you pass a VI Server reference to a subVI, use Property Nodes and Invoke Nodes to read and write properties and invoke methods of the referenced front panel object.



NOTE

Control references do not automatically keep a VI in memory and do not control whether or not a VI remains in memory. If a VI leaves memory, the control reference becomes invalid. To maintain a VI in memory when using a control reference, you can use the Open VI Reference function to retrieve the VI and use the Close Reference function to properly close the VI. You also can maintain a VI in memory by keeping the front panel of the VI open.

You also can [use events](#) to control block diagram behavior programmatically through front panel objects.

Although you can use control references to pass data, you should read and write to the control itself or use [local or global variables](#) instead to improve performance.

Control references correspond to user interface object references in text-based programming languages. Control references do not correspond to pointers in text-based programming languages.

Refer to the Control References VI in the `labview\examples\Application Control\VI Server\Control References` directory for an example of using control references. Find related examples

Strictly Typed and Weakly Typed Control Refnums

Strictly typed control refnums accept only control refnums of exactly the same data type. For example, if the type of a strictly typed control refnum is 32-bit integer slide, you can wire only a 32-bit integer slide to the control refnum terminal. You cannot wire an 8-bit integer slide, a double-precision scalar slide, or a cluster of 32-bit integer slides to the control refnum terminal.

Control references you create from a control are strictly typed by default. A red star in the lower left corner of the control reference on the front panel indicates the control reference is strictly typed. On the block diagram, (strict) appears on the Property Node or Invoke Node wired to the control reference terminal to indicate that the control reference is strictly typed.



Note Because the latch mechanical actions are incompatible with strictly typed control references, Boolean controls with latch mechanical action produce weakly typed control references.

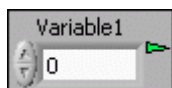
Weakly typed control refnums are more flexible in the type of data they accept. For example, if the type of a weakly typed control reference is slide, you can wire a 32-bit integer slide, single-precision slide, or a cluster of 32-bit integer slides to the control reference terminal. If the type of a weakly typed control reference is control, you can wire a control reference of any type of control to the control reference terminal.



Note When you wire a Property Node to a weakly typed control reference terminal, the Value property produces *variant data*, which might require conversion before you can use the data. The History Data property for charts is available only if the chart reference is strictly typed.

Troubleshooting Front Panel Data Binding

When you bind a front panel control or indicator to a shared variable at edit-time, a small transparent indicator appears next to the front panel object. When you run the VI, if the connection to the Shared Variable Engine is successful, the indicator turns green, as shown in the following figure.



If the connection is not successful, the indicator turns red. A status message also appears if you move the cursor over the indicator.

**NOTE**

You can hide this indicator using the following methods:

- Right-click the front panel object and select **Visible Items»PSP LED** from the shortcut menu.
- Use the Data Binding:LED Visible property to hide the indicator programmatically.

If you receive errors when attempting to read or write data from the front panel, investigate the following possible causes:

- Ensure that you have *deployed* the shared variable.
- Ensure that the device for the target on which the shared variable resides is functional and supports front panel data binding. Refer to the documentation for the LabVIEW modules and toolkits you have installed to determine whether they support front panel data binding.

Variables in LabVIEW

Local Variables, Global Variables, and the Feedback Node

In LabVIEW, you read data from or write data to a front panel object using its block diagram terminal. However, a front panel object has only one block diagram terminal, and your application might need to access the data in that terminal from more than one location.

Local and global variables pass information between locations in the application that you cannot connect with a wire. Use local variables to access front panel objects from more than one location in a single VI. Use global variables to access and pass data among several VIs.

Use a Feedback Node to store data from a previous VI or loop execution.

Refer to the Feedback Node - Building an Array in the Structures\Feedback Node directory for examples of the Feedback Node.



RELATED INFORMATION

[Local Variables](#) on page 605

[Feedback Nodes: Storing Data between Loop Iterations](#) on page 417

[Local Variables](#) on page 605

[Global Variables](#) on page 606

[Feedback Nodes: Storing Data between Loop Iterations](#) on page 417

[Read and Write Variables](#) on page 607

[Using Local and Global Variables Carefully](#) on page 607

[Creating Local Variables](#) on page 1136

[Changing the Data Direction of a Local Variable](#) on page 1135

[Creating Global Variables](#) on page 1137

[Changing the Data Direction of a Global Variable](#) on page 1137

Local Variables

Create local variables for front panel objects in a single VI when you do not have access to a front panel object or you need to pass data between block diagram nodes. When you create a local variable, it appears on the block diagram, but it does not appear on the front panel.

With a local variable, you can write to or read from a control or indicator on the front panel. Writing to a local variable is similar to passing data to any other terminal. However, with a local variable you can write to it even if it is a control or read from it even if it is an indicator. In effect, with a local variable, you can access a front panel object as both an input and an output.

For example, if the user interface requires users to log in, you can clear the Login and Password prompts each time a new user logs in. Use a local variable to read from the **Login** and **Password** string controls when a user logs in and to write empty strings to these controls when the user logs out.

Creating Local Variables

Right-click an existing front panel object or block diagram terminal and select **Create»Local Variable** from the shortcut menu to [create a local variable](#). A local variable icon for the object appears on the block diagram.

You also can select a local variable from the Functions palette and place it on the block diagram. The local variable node, shown as follows, is not yet associated with a control or indicator.



To associate a local variable with a control or indicator, right-click the local variable node and select **Select Item** from the shortcut menu. The expanded shortcut menu lists all the front panel objects that have owned labels.

LabVIEW uses owned *labels* to associate local variables with front panel objects, so label the front panel controls and indicators with descriptive owned labels.

Global Variables

Use global variables to access and pass data among several VIs that run simultaneously. Global variables are built-in LabVIEW objects. When you create a global variable, LabVIEW automatically creates a special global VI, which has a front panel but no block diagram. Add controls and indicators to the front panel of the global VI to define the data types of the global variables it contains. In effect, this front panel is a container from which several VIs can access data.

For example, suppose you have two VIs running simultaneously. Each VI contains a While Loop and writes data points to a waveform chart. The first VI contains a Boolean control to terminate both VIs. You must use a global variable to terminate both loops with a single Boolean control. If both loops were on a single block diagram within the same VI, you could use a local variable to terminate the loops.

Creating Global Variables

Select a global variable, shown as follows, from the Functions palette and place it on the block diagram.



Double-click the global variable node to display the front panel of the global VI. Place controls and indicators on this front panel the same way you do on a standard front panel.

LabVIEW uses owned *labels* to identify global variables, so label the front panel controls and indicators with descriptive owned labels.

You can create several single global VIs, each with one front panel object, or if you want to group similar variables together, you can create one global VI with multiple front panel objects.

After you finish placing objects on the global VI front panel, save it and return to the block diagram of the original VI. You must then select the object in the global VI that you want to access. Click the global variable node and select a front panel object from the shortcut menu. The shortcut menu lists all the front panel objects in the global VI that have owned labels. You also can right-click the global variable node and select a front panel object from the **Select Item** shortcut menu.

If you create a copy of a global variable node, LabVIEW associates the new global variable node with the global VI of the variable node you copy it from.

Read and Write Variables

After you create a local or global variable, you can read data from a variable or write data to it. By default, a new variable receives data. This kind of variable works as an indicator and is a write local or global. When you write new data to the local or global variable, the associated front panel control or indicator updates to the new data.

You also can configure a variable to behave as a data source, or a read local or global. Right-click the variable and select **Change To Read** from the shortcut menu to configure the variable to behave as a control. When this node executes, the VI reads the data in the associated front panel control or indicator.

To change the variable to receive data from the block diagram rather than provide data, right-click the variable and select **Change To Write** from the shortcut menu.

On the block diagram, you can distinguish read locals or globals from write locals or globals the same way you distinguish controls from indicators. A read local or global has a thick border similar to a control. A write local or global has a thin border similar to an indicator.

Using Local and Global Variables Carefully

Local and global variables are advanced LabVIEW concepts. They are inherently not part of the LabVIEW dataflow execution model. Block diagrams can become difficult to read when you use local and global variables, so you should use them carefully. Misusing local and global variables, such as using them instead of a connector pane or using them to access values in each frame of a sequence structure, can lead to unexpected behavior in VIs. Overusing local and global variables, such as using them to avoid long wires across the block diagram or using them instead of *data flow*, slows performance.

Initializing Local and Global Variables

To initialize a local or global variable, write known values to the variable before you run the VI. Otherwise, the variables might contain data that causes the VI to behave incorrectly. If the variable relies on a computation result for the initial value, make sure LabVIEW writes the value to the variable before it attempts to access the variable for any other action. Wiring the write action in parallel with the rest of the VI can cause a *race condition*.

To ensure that the variables initialize before the rest of the VI executes, you can isolate the code that writes the initial value for the variables to the first frame of a *sequence structure* or to a subVI and wire the subVI to execute first in the data flow of the block diagram.

If you do not initialize the variable before the VI reads the variable for the first time, the variable contains the default value of the associated front panel object.

When a global or local variable value is wired to anything else on the diagram, the value on the wire does not change when the variable changes. To update the wire value, you must read the value of the variable after the change.

Race Conditions

A race condition can occur when two or more pieces of code execute in parallel and have access to a shared piece of memory. If each piece of code is independent, there is no way to distinguish the order LabVIEW uses to access the shared resource.

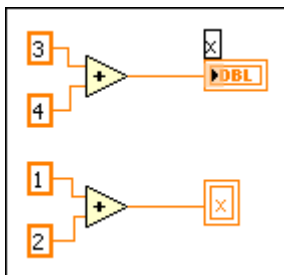
Race conditions can cause unpredictable outcomes. For example, if two independent pieces of code access a queue and you do not control the order LabVIEW uses to access the queue, you can cause a race condition.

Race conditions can be dangerous because they can appear and disappear when the timing of a program changes. They can occur when you change operating systems, LabVIEW versions, or other software that runs on the system.

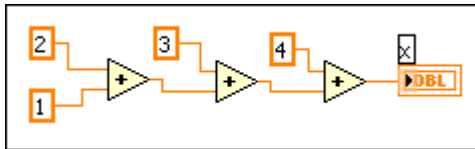
If you cause timing to change in a VI, such as when you update the operating system or LabVIEW version, look for parallel code that accesses the same data, and use a timing condition to control which actions execute first.

Race Conditions when Using Local and Global Variables

Although race conditions can exist any time more than one action updates the value of the same stored data, race conditions often occur with the use of local and global variables or an external file. The following block diagram shows an example of a race condition with a *local variable*.



The output of this VI, the value of local variable x, depends on which operation runs first. Because each operation writes a different value to x, there is no way to determine whether the outcome will be 7 or 3. In some programming languages, a top-down dataflow paradigm ensures execution order. In LabVIEW, you can use wiring to perform multiple operations on a variable while avoiding race conditions. The following block diagram performs addition operations using wiring instead of a local variable.

**TIP**

If you must perform more than one action on a local or global variable, make sure you determine the order of execution.

Race conditions also occur when two operations are trying to update a global variable in parallel. In order to update the global variable, an operation reads the value, modifies it, and writes it back to the location. When the first operation performs the read-modify-write action and the second operation follows after, the outcome is correct and predictable. When the first operation reads and then the second operation reads, both operations modify and write a value. This action causes the read-modify-write race condition and produces invalid or missing values.

To help avoid the race conditions associated with global variables, you can use *functional global variables* to protect critical sections of code that access the action on a variable. Using one functional global variable instead of multiple local or global variables ensures that only one operation executes at a time, so you never perform conflicting operations or assign conflicting values to stored data.

Variable Behavior in Loops

When a variable controls multiple parallel loops, the loops must execute after the variable is written and you must read the variable in each iteration of each loop.

When the system restarts, the variable must be set to a proper starting condition before any loops that read the variable execute or the loops will stop prematurely.

Memory Considerations when Using Local Variables

Local variables make copies of data buffers. When you read from a local variable, you create a new buffer for the data from its associated control.

If you use local variables to transfer large amounts of data from one place on the block diagram to another, you generally use more memory and, consequently, have slower execution speed than if you transfer data using a wire. If you need to store data during execution, consider using a shift register.

Memory Considerations when Using Global Variables

When you read from a global variable, LabVIEW creates a copy of the data stored in that global variable.

When you manipulate large arrays and strings, the time and memory required to manipulate global variables can be considerable. Manipulating global variables is

especially inefficient when dealing with arrays because if you modify only a single array element, LabVIEW stores and modifies the entire array. If you read from the global variable in several places in an application, you create several memory buffers, which is inefficient and slows performance.

Refer to the [Managing Performance and Memory](#) book for more information about performance and memory management considerations in LabVIEW.

Understanding Shared Variable Technology

Network-published shared variables publish data over a network through a software component called the Shared Variable Engine (SVE). The SVE is installed as a service on your computer when you install LabVIEW, and it manages shared variable updates using a proprietary technology called the NI Publish-Subscribe Protocol (NI-PSP). The term publish-subscribe describes a model of communication where writers, or publishers, do not send data to specific readers, or subscribers. Rather, publishers send updates to a server, in this case the SVE, and subscribers receive those updates from the server.



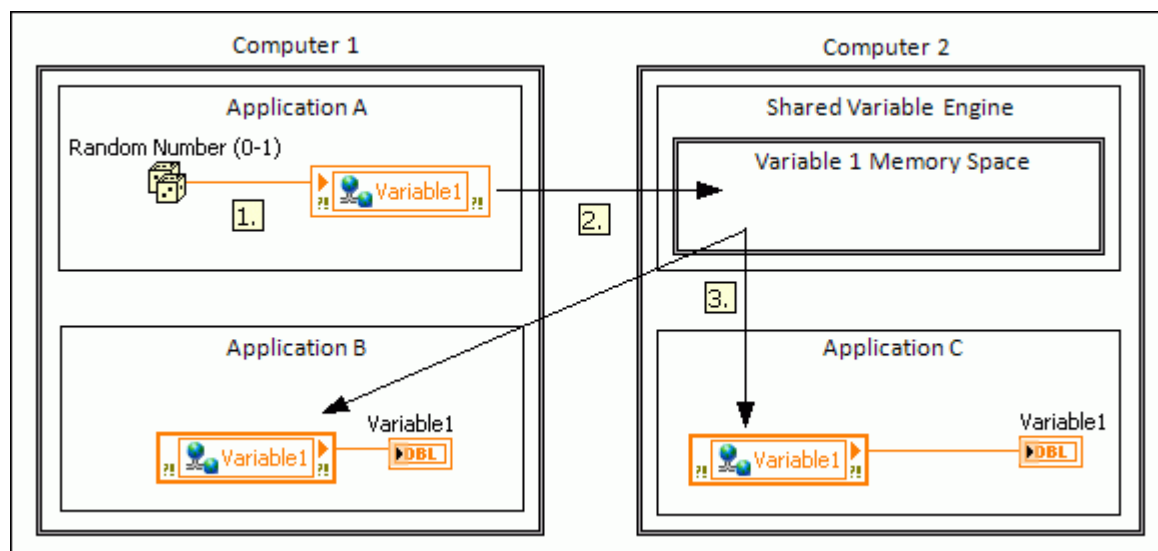
NOTE

To transmit network-published shared variables through firewalls or routers, you must [configure firewalls and network address translating \(NAT\) routers](#).

The following figure uses Shared Variable nodes to demonstrate how the SVE manages shared variable updates with NI-PSP.



Note The [Shared Variable node](#) is one of many programming tools you can use to [read and write shared variables](#).



The following events occur in the previous figure.

1. In Application A, the Random Number (0-1) function writes a random number to the Shared Variable node that corresponds to Variable 1.

Note To read and write a shared variable over a network, you first must *make that shared variable available on the network*.

2. The Shared Variable node in Application A sends a request to the SVE to update the value of Variable 1.
3. The SVE approves and sends the new value to the Shared Variable nodes that correspond to Variable 1 in Applications B and C.

In the previous figure, although Computer 1 hosts a writer of Variable 1 in Application A and a reader of Variable 1 in Application B, Application A cannot write a new value directly to Application B. Instead, Application A must send a request to the SVE on Computer 2 to update every application that subscribes to Variable 1. Therefore, the latency involved in these updates makes shared variables ideal for publishing latest values only. To stream data continuously, use *network streams*.

Making Shared Variables Available on a Network

You must *deploy shared variables to the SVE* to read and write those shared variables over a network. When you deploy a shared variable, LabVIEW deploys the *project library* that contains the shared variable, including all of the other shared variables in the project library, to the SVE. The SVE then creates a memory space for that project library that includes memory spaces for each shared variable.

NI-PSP Data Items

Any data item that LabVIEW publishes over a network using NI-PSP is an NI-PSP data item. Common NI-PSP data items include the following:

- Network-published shared variables
- I/O variables with network-publishing enabled
- Data items on OPC servers
- Data items on FieldPoint modules

LabVIEW identifies NI-PSP data items with a network path that consists of the name of the computer the data item resides on, the name of the process the data item resides in, and the name of the data item. For example, the network path `\\computer1\\mylib\\myvariable` identifies a shared variable named `myvariable` in the `mylib` project library on a computer named `computer1`. If the shared variable exists in a sub-project library named `mysublib`, the shared variable path includes the sub-project library: `\\computer1\\mylib\\mysublib\\myvariable`.

Specify the network path of an NI-PSP data item, or browse to an NI-PSP data item, when reading and writing shared variables with *existing data sources* or *controls and indicators*.

Shared Variable Engine Errors

The SVE reports server status updates and critical errors, such as invalid process configurations. You can view these errors in the Windows Event Viewer. In the Windows **Control Panel**, navigate to the Event Viewer in the Administrative Tools directory. The exact location of the Administrative Tools directory varies depending on which version of Windows you use.



[Data Communication Methods Home](#)

Publishing Latest Values with Shared Variables

A shared variable is a memory space that you can read data from and write data to. You can read and write shared variables on a single computer with single-process shared variables or on multiple computers with network-published shared variables, which publish data over a network using the NI Publish-Subscribe Protocol (NI-PSP). Use shared variables to publish only the latest values in a data set to one or more computers.



NOTE

To transfer every data point from one computer to another, National Instruments recommends sending commands or streaming data continuously with network streams. Refer to the Transferring Data over a Network topic for more information about choosing a networking feature that best fits your application needs.

In general, you must complete the following tasks to publish latest values with a shared variable:

1. Create a shared variable.
2. Configure the shared variable.
3. Program your applications to read and write the shared variable.
4. Make the shared variable available on the network.

Extending the Functionality of Shared Variables with the Real-Time and DSC Modules

The LabVIEW Real-Time and DSC Modules offer additional configuration and programming options for shared variables. For example, you can enable the real-time FIFOs of shared variables with the Real-Time Module and configure scaling and alarming properties for shared variables with the DSC Module. Refer to the Real-Time Module and DSC Module Help for more information about extending the functionality of shared variables.

Using Other Types of Variables

This book mainly describes the process of publishing latest values over a network using network-published shared variables. However, you also can perform the following tasks with other types of variables:

- Use I/O variables for single-point access to I/O data through the NI Scan Engine.
- Use global variables to read and write data on a single computer.
- Use local variables to read and write data in a single VI.



RELATED INFORMATION

[Binding Shared Variables and Other NI-PSP Data Items to Controls and Indicators](#) on page 633

[Designing Applications Using Network Streams](#) on page 663

[Choosing a Method of Reading and Writing Shared Variables](#) on page 619

[Creating Shared Variables](#) on page 618

[Configuring Shared Variables](#) on page 613

[Reading and Writing Shared Variables with Shared Variable Nodes](#) on page 629

[Making Shared Variables Available on a Network](#) on page 649

[Using I/O Variables \(Real-Time, Windows\)](#) on page 1696

[Point-by-Point Analysis and Array-Based Analysis in](#) on page 1688

[Accessing Scanned I/O Data \(Real-Time, Windows\)](#) on page 1706

Use the NI Scan Engine for single-point access to scanned I/O data.

[Global Variables](#) on page 606

[Local Variables](#) on page 605

[Data Communication Methods in LabVIEW](#) on page 651

Configuring Shared Variables

Use the Shared Variable Properties dialog box to configure shared variables from the [Project Explorer](#) window. To display this dialog box, [create a new shared variable](#) or right-click an existing shared variable in the Project Explorer window and select **Properties** from the shortcut menu.

The following list describes the functionality and use cases of some common shared variable configuration options:

- Enter a name for the shared variable in the **Name** text box.
- Select the [data type](#) for the shared variable from the **Data Type** pull-down menu. If you do not see the data type you want to use, select **From Custom Control** to use the data type of a [custom control](#).
- Select the type of shared variable you want to create based on the network location of the computers you will read and write the variable with. You can choose from the following options with the **Variable Type** pull-down menu:
 - Select **Network-Published** to create shared variables that you want to read and write on remote computers and targets on the same network.

- Select **Single-Process** to create shared variables that you want to read and write on a single computer.
- If you select **Network-Published** from the **Variable Type** list, you can place a checkmark in the **Use Buffering** checkbox on the Network page to enable client-side [buffering](#) for the shared variable data. Depending on the data type you select from the **Data Type** pull-down menu, you might need to enter additional values in the **Use Buffering** section.

**NOTE**

When you select this option, the shared variable stores data in a FIFO buffer. However, this option does not guarantee lossless data communication because new values still can overwrite old values if the buffer reaches capacity. If your application requires lossless communication, use network streams to [stream data between applications continuously](#). Refer to the National Instruments Web site for a detailed explanation of shared variable buffering behaviors.

- If you select **Network-Published** from the **Variable Type** list, you can place a checkmark in the **Single Writer** checkbox on the **Network** page to accept writes from only one application at a time. When you select this option, the first application that connects to the shared variable can write values to that shared variable, whereas any subsequent applications cannot. When the first application disconnects, the next application in the queue can write values to the shared variable.



[Data Communication Methods Home](#)

Configuring Scaling for Shared Variables

Scaling is useful for converting a range of values from measured units into a calculated range. You can configure scaling for each data type of [shared variables](#). You can configure scaling on the Scaling page of the Shared Variable Properties dialog box.

(DSC Module) You also can use the Scaling SharedVariableIO properties to configure scaling.

Scaling Numeric Shared Variables

You can use [linear and square root scaling](#) for numeric shared variables.

Often an application needs you to manipulate the raw data used in the device server to put the data in a form, called engineering units, suitable for the users of the application. You can define the raw range and engineering range for a shared variable to perform simple conversions between the two ranges on the Scaling page of the Shared Variable Properties dialog box. The raw range refers to the values used by the device server. The engineering range refers to the values used by the [Shared Variable Engine](#) and the Human Machine Interface (HMI) application.

Linear and Square Root Scaling

Linear scaling is defined by the following equation:

$$m * raw + b$$

where b is the **Engineering Zero Scale**, and m is the **Engineering Full Scale – Engineering Zero Scale/Raw Full Scale – Raw Zero Scale**.

Square root scaling is defined by the following equation:

$$b + m * \sqrt{raw - o}$$

where b is the **Engineering Zero Scale**, m is the **Engineering Full Scale – Engineering Zero Scale/sqrtRaw Full Scale – Raw Zero Scale**, and o is the **Raw Zero Scale**.

The following table shows shared variable values that range from 0 to 100 scaled to engineering units ranging from 0 to 10.

Raw Units	Linear Scale	Square Root Scale
0	0	0
4	.4	2
9	.9	3
10	1	3.16
16	1.6	4
20	2	4.47
25	2.5	5
30	3	5.48
36	3.6	6
40	4	6.32
49	4.9	7
50	5	7.07
60	6	7.75
64	6.4	8
70	7	8.37
80	8	8.94
81	8.1	9

Raw Units	Linear Scale	Square Root Scale
90	9	9.49
100	10	10

The following table shows shared variable values that range from 0 to 100 scaled to engineering units ranging from 15 to 30. Offset scaling delivers more complicated results.

Raw Units	Linear Scale	Square Root Scale
0	15	15
4	15.60	18
10	16.50	19.74
16	17.40	21
20	18	21.71
30	19.50	23.22
36	20.40	24
40	21	24.49
50	22.50	25.61
60	24	26.62
64	24.60	27
70	25.50	27.55
80	27	28.42
90	28.5	29.23
100	30	30

Linear Scaling Example

In this example, a device server returns a voltage from 0 to 5 V. The voltage is related to a position sensor, and the real-world position is measured in centimeters, with 0 V mapped to 50 cm and 5 V mapped to 100 cm. The following table shows the values of the shared variables after scaling.

Raw Units (V)	Linear Scale (cm)
0	50

Raw Units (V)	Linear Scale (cm)
1	60
2	70
3	80
4	90
5	100

Square Root Scaling Example

In this example, a flow meter measures the flow rate of a liquid using a differential pressure reading. The device server provides 4 to 20 mA readings. The actual flow is measured in gallons per minute (GPM), where 4 mA corresponds to 0 GPM and 20 mA corresponds to 100 GPM. The following table shows the values of the shared variables after scaling.

Raw Units (mA)	Square Root Scale (GPM)
4	0
5	25
6	35.36
7	43.3
8	50
10	61.24
13	75
15	82.92
20	100

(DSC Module) Use the `Scaling.UnitsSharedVariableIO` property to assign units to a shared variable. In the previous example, you could specify gallons per minute as a constant to the `Scaling.Units` property.

Scaling Boolean Shared Variables

You can use *inverted scaling* for Boolean shared variables. When you enable inverted scaling, LabVIEW inverts the Boolean value when communicating with the server.

Scaling U32 Bit Field Shared Variables

You can use *invert mask and select mask scaling* for U32 bit field shared variables.

Mask scaling applies only to U32 bit fields. LabVIEW stores a bit field as a number, translating the number back into a 32-bit field when needed. The bit field holds a collection of 32 values, each value limited to 1 or 0. With mask scaling, you can decide which of these array members to pay attention to. Any bit filtered out by the mask appears as a zero. Bits not filtered out appear as zero or one, reflecting the actual value. A mask of

00000000000000000011111111111111 0x0000FFFF

where 0 marks a position to filter out and 1 marks a place to report, reads the bit array

10110011100011001100111000110010

as the following:

0000000000000000001100111000110010



NOTE

The invert mask does not invert bits that are masked out.

The following table shows examples of shared variables configured for U32 bit field scaling.

Shared Variable Name	Raw Value	Scaling Invert Mask	Scaling Select Mask	Scaled Value
Shared Variable 1	0x0000000F	0x00000000	0x000000FF	0x0000000F
Shared Variable 2	0x0000000F	0x00000033	0x000000FF	0x0000003C
Shared Variable 3	0x0000000F	0x00000033	0x0000000F	0x0000000C
Shared Variable 4	0x0000000F	0x00000000	0x00000033	0x00000003
Shared Variable 5	0x0000000F	0x00000033	0x00000033	0x00000030
Shared Variable 6	0x00000FF0	0x0000000F	0x000000FF	0x000000FF

Creating Shared Variables

Complete the following steps to create a *shared variable*.



NOTE

You must have a project open to create a shared variable.

1. Right-click a target, project library, or folder within a project library in the *Project Explorer* window.

Note If you right-click a target or a folder that is not inside a project library and select **New»Variable** from the shortcut menu, LabVIEW places the shared variable inside a new project library.

2. Select **New»Variable** from the shortcut menu. The Shared Variable Properties dialog box opens.
3. Use this dialog box to *configure the shared variable*.
4. Click the **OK** button to create the shared variable.

After you create a shared variable, *choose the method of reading and writing shared variables that best fits your application needs*.



Data Communication Methods Home

Choosing a Method of Reading and Writing Shared Variables

When choosing a method of reading and writing *shared variables*, you must consider several factors, including the following:

- Update speed
- Programming complexity
- Scalability

The following table describes the use cases, advantages, and disadvantages of the methods you can use to read and write shared variables over a network.



NOTE

After you program your applications to read and write shared variables with one or more of the following features, you must *deploy the shared variables* to make them available on the network.

Method of Reading and Writing Shared Variables	Recommended Uses	Advantages	Disadvantages
<i>Programmatically</i> with the Shared Variable functions	Reading and writing large numbers of shared variables. For example, if you need to read 50 shared variables of the double data type in a specific <i>project library</i> , reading those shared variables programmatically is the most efficient method.	• A single Shared Variable function can access multiple shared variables. Other methods can read and write only one shared variable at a time. • Scalable. If you add 50 shared variables to a project, you do not have to update your block diagram to read and write each of those variables. • Offers access to <i>I/O variables</i>.	Requires more programming than other methods because you must program your application to read and write shared variables that meet specific criteria.

Method of Reading and Writing Shared Variables	Recommended Uses	Advantages	Disadvantages
With <i>Shared Variable nodes</i>	• Reading and writing small numbers of shared variables. For example, if you need to update the values of 10 shared variables, you can use Shared Variable nodes. • Reading and writing data at a fast rate in a human-machine interface (HMI) application. For example, if you need to display the latest value from a temperature sensor once per second, use a Shared Variable node. • Offers access to I/O variables.	• Ease of programming. To add a Shared Variable node to your application, drag a shared variable from the <i>Project Explorer</i> window to the block diagram of your application. • LabVIEW labels each Shared Variable node with the name of the corresponding shared variable.	• Each Shared Variable node can access only a single shared variable. • Not scalable. You must add a Shared Variable node to the block diagram every time you want to read or write an additional shared variable.

Method of Reading and Writing Shared Variables	Recommended Uses	Advantages	Disadvantages
With <i>controls and indicators</i>	Reading and writing data at a slow rate in an HMI application. For example, if you need to display the latest value from a temperature sensor once every few seconds, read shared variables with an indicator.	Ease of programming. To read or write a shared variable with a control or indicator, drag a shared variable from the Project Explorer window to the front panel of your application.	- Does not enforce data flow. Therefore, the operations you can perform with the data you read and write are limited. - Does not indicate which controls and indicators access what shared variables. - Slower processing speed than using Shared Variable nodes. - Does not offer access to I/O variables.

Method of Reading and Writing Shared Variables	Recommended Uses	Advantages	Disadvantages
With <i>existing data sources</i>	Leveraging shared variable features for a data source that does not support those features. For example, real-time targets do not support DSC Module logging and alarming properties. Therefore, if you need to set DSC Module logging and alarming properties for a shared variable on a real-time target, you could create a new shared variable on your development computer, write that shared variable with the shared variable on the CompactRIO target, and configure the shared variable on the development computer for logging and alarming.	• Offers access to I/O variables. • Adds configuration options to shared variables.	If you do not need to leverage shared variable features for a data source or if the data source already supports the features you need, this method is inefficient because you can read and write the data source directly.
With the <i>DataSocket functions</i>	Reading and writing shared variables from Mac and Linux computers.	The DataSocket functions are the only programming tools you can use to read and write shared variables from Mac and Linux computers.	• Slower processing speed than the other methods of reading and writing shared variables. • Does not offer access to I/O variables.

Method of Reading and Writing Shared Variables	Recommended Uses	Advantages	Disadvantages
With the <i>variable Web service</i>	Reading and writing shared variables from Web-enabled platforms that support HTTP.	- The <i>variable Web service</i> is the only programming tool you can use to read and write shared variables from many Web-enabled platforms that support HTTP. - Offers access to I/O variables.	If you do not need to leverage shared variable features for Web-enabled platforms that support HTTP, this method is inefficient and requires more programming than other methods.



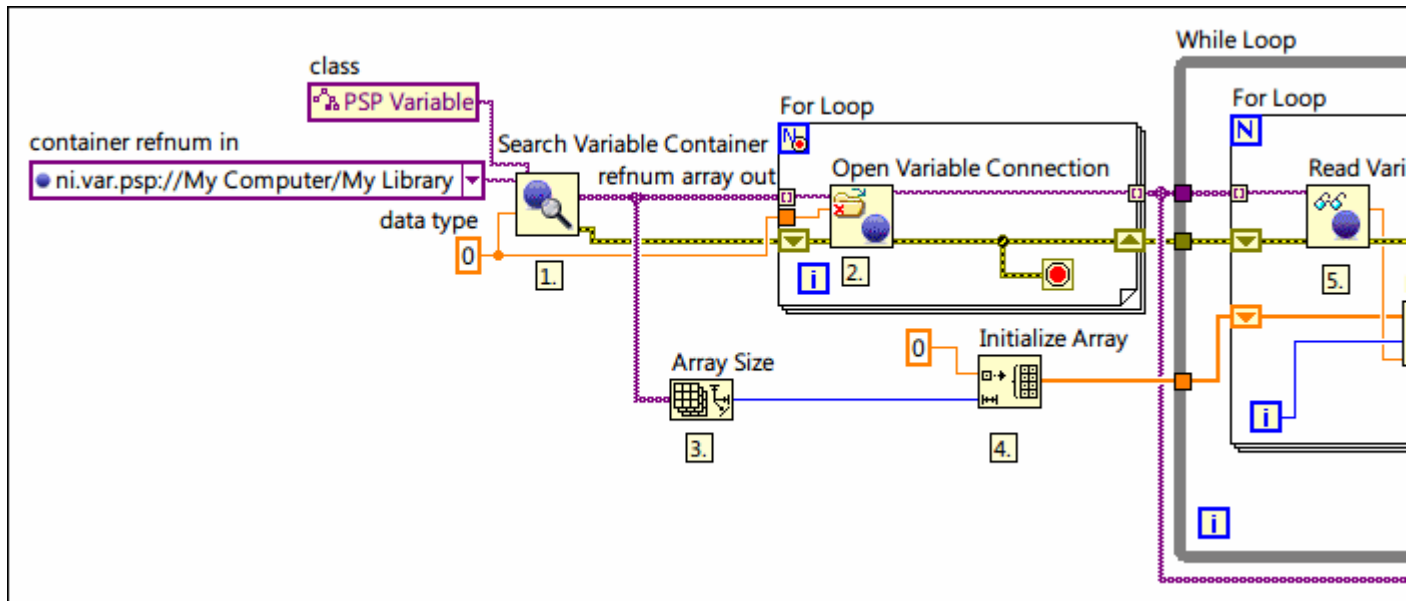
Reading and Writing Shared Variables Programmatically

If you have an application that accesses a large number of shared variables, you can read and write those shared variables programmatically with the Shared Variable functions. This programming technique often is cleaner and more scalable than *using Shared Variable nodes* because you can access multiple shared variables with a single Shared Variable function, whereas each Shared Variable node accesses a single shared variable only.



- NOTE**
- You can access the following types of shared variables programmatically:
- PSP Variables** —Data items published on the network using the *NI Publish-Subscribe Protocol (NI-PSP)*. Common examples of PSP variables include network-published shared variables, I/O variables with network-publishing enabled, and DAQ channels published through the NI-DAQmx I/O server.
 - I/O Variables** —Shared variables that use the *NI Scan Engine* for single-point access to I/O data.

The following figure shows an example of reading shared variables programmatically, placing those values in a preallocated array, and processing those values.



The following events occur in the previous figure.

1. The Search Variable Container function finds all of the shared variables of the double data type under **My Library**, and the **refnum array out** output returns an array of references to those shared variables.
2. The Open Variable Connection function uses the array of variable identifier URLs to open a connection to each shared variable. The For Loop stops if an error occurs.
3. The Array Size function uses the array of shared variable references to calculate the number of shared variables that the Search Variable Container function found.
4. The Initialize Array function uses the number of shared variables to create an array for the shared variable data. By initializing an array for the shared variable data outside of the While Loop, this application optimizes CPU usage.
5. The Read Variable function reads each of the shared variables that the Open Variable Connection function opened.
6. The Replace Array Subset function stores the shared variable data in the array that the Initialize Array function created.
7. The Process Data subVI processes the array of shared variable data.
8. The read and process operations run continuously in the While Loop until an error occurs or the user clicks the **Stop** button.
9. The Close Variable Connection function closes the shared variable connections.

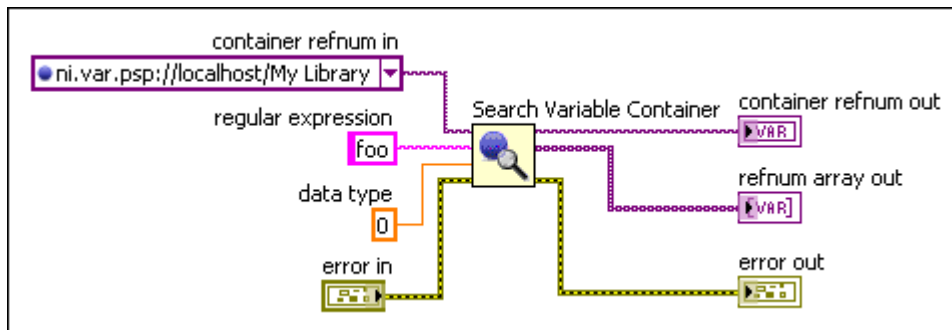
This application achieves the same functionality as one that uses Shared Variable nodes. However, because this application uses programmatic shared variable access, you can scale to hundreds or thousands of variables without updating the block diagram. For example, assume that **My Library** contains ten variables, Variable1 through Variable10. You could use the Batch Variable Creation dialog box to add Variable11 through Variable500 to **My Library**. After you redeploy **My Library**, the next time you run the VI, it reads Variable1 through Variable500 with no need to add Variable11 through Variable500 to the block diagram.

The following sections provide information about the programming techniques in the above example.

Finding Shared Variables

Use the Search Variable Container function to search for shared variables and variable containers that meet specific criteria. For example, the VI shown in the following block diagram uses this function to find all the shared variables under a given container that meet the following criteria:

- Exists in My Library on the local computer
- Double data type
- *Variable identifier URL* contains the string foo



Note You can use the Browse Variable Object dialog box to browse for shared variables and variable containers at edit-time either on the network or in the current project. Click the drop-down arrow on a Variable control or constant, such as the **container refnum in** constant in the figure above, and select **Browse** to display the **Browse Variable Object** dialog box.

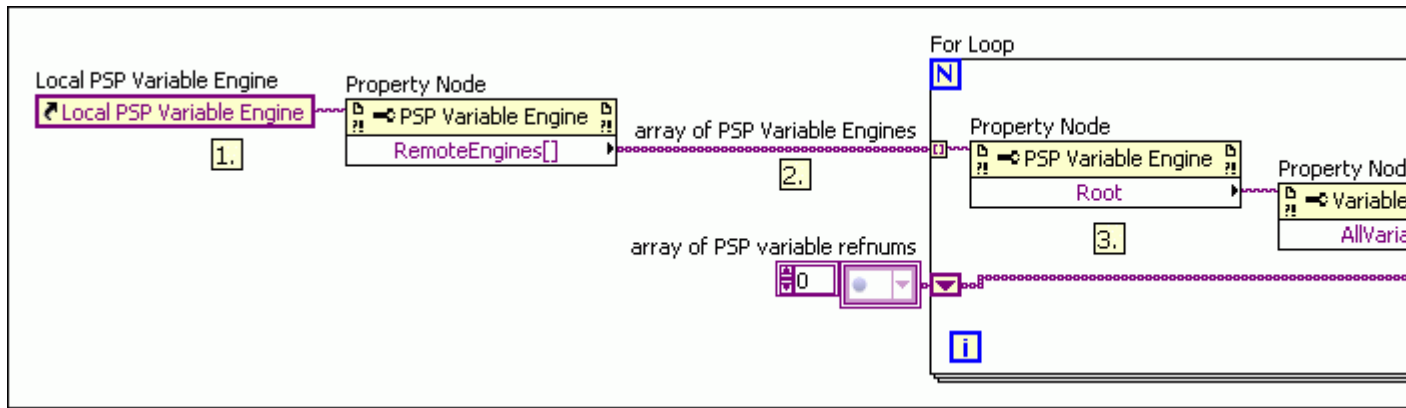


Caution When reading and writing shared variables that you find programmatically, you might read or write data of the wrong data type. Use the **data type** input of the Search Variable Container function to avoid reading or writing a shared variable of the wrong data type.

You also can use the following Variable Container properties and method to find shared variables programmatically.

- All Variables property
- Children property
- Find Descendants method

The VI in the following figure uses the All Variables property to find all the PSP variables hosted on the local network.



The following events occur in the previous figure.

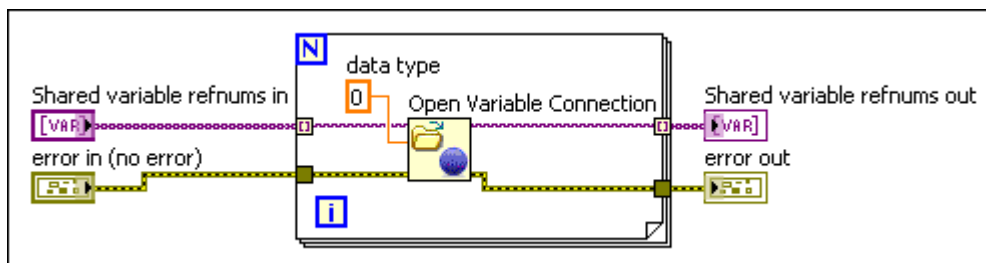
1. The Local Variable Object Reference constant passes a reference to the local PSP Variable Engine to the Remote Engines property.
2. The Remote Engines property returns the **array of PSP Variable Engines** hosted on other computers on the network. The VI then iterates over each PSP Variable Engine in a For Loop.
3. The Root property obtains a reference to the root PSP Variable Container on each remote computer.
4. The All Variables property obtains an array of references to all the variables hosted on each remote computer.
5. The VI adds all the variables found on the network to the **All PSP Variables** array.



Tip You can use the Variable Property Node, which is preconfigured with the Variable Object class, as a starting point for building the block diagram in the previous figure.

Opening and Closing Shared Variable Connections

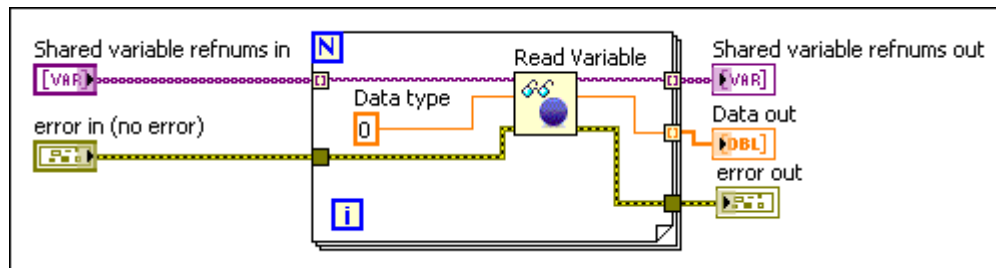
Use the Open Variable Connection function to open a connection to a variable programmatically, as shown in the following block diagram.



After the application finishes accessing the shared variables, use the Close Variable Connection function in a For Loop to close all the variable references. This function frees the memory resources allocated to each variable connection.

Reading and Writing Shared Variables

Use the Read Variable function to read live variable values programmatically. Unlike the Shared Variable node, which you must bind to a specific shared variable at edit time, the Read Variable function includes a **shared variable refnum in** input that you use to specify the shared variable to read at run time. For example, you can use the Read Variable function inside a For Loop to read multiple shared variables specified by an array of refnums, as shown in the following block diagram.



Use the Write Variable function to write live variable values programmatically. For example, you can use the Write Variable function inside a For Loop to write to multiple shared variables specified by an array of refnums.



Tip You can optimize performance and streamline the block diagram by *reading and writing I/O variable containers* as arrays.



Note If you write less than 8 kilobytes of data to a shared variable, LabVIEW can take up to 10 milliseconds to send that data over the network. To eliminate this delay, use the Flush Shared Variable Data VI to *send the shared variable data immediately*.



[Data Communication Methods Home](#)

Reading and Writing Shared Variables with Shared Variable Nodes

(Real-Time, Windows) A Shared Variable node is a block diagram object that points to a corresponding [shared variable](#). Use the Shared Variable node to read and write the value of a shared variable.



NOTE

- If your application accesses a large number of shared variables, [access the shared variables programmatically](#) to create a cleaner, more scalable block diagram.
- (Real-Time, Windows) Opening a VI containing a Shared Variable node in a project that does not contain the corresponding shared variable in the Project Explorer window causes the Shared Variable node to break. Any front panel controls associated with the missing shared variable also break. This behavior is specific to Windows and occurs only when you open the VI in a project. If you open the VI in the main application instance, LabVIEW does not notify you about missing shared variables.

Adding Shared Variable Nodes to the Block Diagram

You can add Shared Variable nodes to the block diagram in the following ways:

- Drag a shared variable from the Project Explorer window onto the block diagram of a VI in the same project.
- Add a Shared Variable node to the block diagram from the Shared Variable palette. You then must complete the following steps to point the Shared Variable node to a corresponding shared variable in the Project Explorer window.
 1. Double-click the Shared Variable node to display the Browse for Variable dialog box.
 2. Navigate to a shared variable in the **Project/Network Tree** list.
 3. Click the **OK** button.
- (Real-Time, FPGA, and DSC Modules) Right-click a control, indicator, or constant and select **Change to Shared Variable Node** from the shortcut menu. You also can select **Create»Shared Variable Node** from the shortcut menu when you right-click an output terminal.

Specifying Whether a Shared Variable Node Reads or Writes Data

Complete the following steps to specify whether a Shared Variable node reads or writes data to the corresponding shared variable in the Project Explorer window.

1. Right-click the Shared Variable node.
2. Choose between the following options in the shortcut menu:

- Select **Access Mode»Read** to read data from the shared variable with the Shared Variable node. When you select this option, an output appears on the right side of the Shared Variable, as shown in the following figure.



- Select **Access Mode»Write** to write data to the shared variable with the Shared Variable node. When you select this option, an input appears on the left side of the Shared Variable node, as shown in the following figure.



Note If you write less than 8 kilobytes of data to a shared variable, LabVIEW can take up to 10 milliseconds to send that data over the network. To eliminate this delay, use the Flush Shared Variable Data VI to *send the shared variable data immediately*.

Determining Whether a Shared Variable Value Has Been Updated Since the Last Read

Complete the following steps to use the **timestamp** indicator of a Shared Variable node to determine whether the shared variable value has been updated since the last read.

	Note • LabVIEW configures <i>network-published shared variables</i> for timestamping by default. To record a timestamp value for a single-process shared variable, you first must place a checkmark in the Enable Timestamping checkbox on the Variable page of the Shared Variable Properties dialog box. • LabVIEW disables timestamps by default on <i>I/O variable</i> nodes because timestamps require additional memory and CPU overhead that can affect the determinism of an application. Enable timestamps for I/O variable nodes only if you plan to use the timestamps.

1. Right-click a Shared Variable node and select **Timestamp»Show** from the shortcut menu. A **timestamp** output appears on the right side of the Shared Variable node, represented by the red hourglass shown in the following figure.



2. Right-click the **timestamp** output and select **Create»Indicator** from the shortcut menu to create a front-panel indicator that displays the timestamp data.

Determining Whether a Shared Variable Node Timed Out

Complete the following steps to specify a timeout period for the Shared Variable node and determine whether the Shared Variable node timed out.



Note You can enable the timeout control only for Shared Variable nodes configured to read data.

1. Right-click the Shared Variable node on the block diagram and select **Timeout»Show** from the shortcut menu
2. Specify a timeout in milliseconds with the **ms timeout** control, represented by the blue clock on the left side of the following figure. Use the **timed out?** indicator, represented by the red clock on the right side of the following figure, to determine whether the Shared Variable node read new data in the specified time period or timed out.



Specifying Whether Shared Variable Nodes Break When You Move Them to Different Targets

(Real-Time, Windows) To specify how a Shared Variable node connects to a shared variable, you can set the node to absolute or target-relative.

An absolute Shared Variable node always connects to the shared variable on the target on which you created the shared variable. Shared Variable nodes are absolute by default. If a VI contains an absolute Shared Variable node and you open the VI on a new target, the VI breaks because the node attempts to connect to the shared variable on the target on which you created the shared variable. Use absolute Shared Variable nodes when you do not expect to move VIs and variables to other targets.

A target-relative Shared Variable node always connects to the shared variable on the target that runs the VI that containing the Shared Variable node. If you move a VI that contains a target-relative Shared Variable node to a new target, you also must move the shared variable to the new target because target-relative Shared Variable nodes can connect to shared variables on the local host only. Use target-relative Shared Variable nodes when you are accessing shared variables on the local host and you expect to move VIs and variables to other targets.

To change an absolute Shared Variable node to target-relative, right-click the Shared Variable node on the block diagram and select **Reference Mode»Target Relative** from the shortcut menu. Target-relative Shared Variable nodes appear with a `\` glyph on the node. The following image shows a target-relative Shared Variable node.



To change a target-relative Shared Variable node to absolute, right-click the Shared Variable node and select **Reference Mode»Absolute** from the shortcut menu. The following image shows an absolute Shared Variable node.





Reading and Writing Shared Variables with the Variable Web Service

Using the variable Web service, you can create applications that access network-published shared variables through HTTP. Any shared variables that utilize the [NI Publish-Subscribe Protocol \(NI-PSP\)](#) can be accessed using the variable Web service. Using Open Data Protocol OData, the variable Web service provides responses in both XML and JSON formats.



NOTE

The variable Web service does not support the following OData parameters: \$filter, \$select, \$top, \$skip, \$links, \$orderby, \$inlinecount, and \$batch.

Installing the Variable Web Service on a Real-Time Target

To use the variable Web service on an RT target, install Variable Web Service 12.0.0 on the RT target using the LabVIEW Real-Time Software Wizard in NI Measurement & Automation Explorer (MAX).

Granting New Permissions to Users

To use the variable Web service, you must grant the appropriate permissions to the appropriate users in NI Web-based Configuration & Monitoring. The following permissions are included with the variable Web service:

- VariableBrowse
- VariableRead
- VariableWrite
- VariableCreateDeleteGroup
- VariableReadGroup

By default, these permissions are granted to the admin only.

To identify themselves and utilize these permissions, users must log in to NI-Auth when prompted. Alternatively, the admin can disable authentication by granting all five permissions to the everyone group.



Reading and Writing Shared Variables with Controls and Indicators

Read and write [shared variables](#) and other NI-PSP data items in human-machine interface (HMI) applications by binding those shared variables to controls and indicators. When you [bind a shared variable to a front panel control or indicator](#), that control or indicator directly writes values to, or reads values from, the shared variable.



NOTE

You can bind front panel objects to network-published shared variables only.

For example, if you need to monitor the temperature of the valves on a boiler, you can bind shared variables to controls and indicators to display and, to a limited extent, process the latest temperature values in the following ways:

- View the value of a shared variable. For example, if you bind a front panel indicator to one of the temperature variables, the indicator displays the latest value of the variable.
- Display or use data from a VI on one computer with a control or indicator in a VI on another computer. For example, if a VI on another computer calculates the average temperature of the valves and writes that data to a shared variable, you can display the same data in a VI on your computer by binding a front panel indicator to that same shared variable. Or, you can use the data by binding a control in your VI to the shared variable and wiring that control to a subVI or function.
- Manipulate and read a control from multiple computers. For example, if you bind the stop buttons in VIs on multiple computers to the same shared variable, each of those controls can shut down the temperature sensor.



Note You can [bind front panel objects to DataSocket server items](#) using the `opc`, `ftp`, and `http` [protocols](#) on Windows, and on all LabVIEW-supported platforms using the `dstp` and `file` [protocols](#). However, using front panel data binding to control a front panel object remotely through the [Shared Variable Engine](#) (SVE) provides faster and more reliable data transmission, as well as connections to shared variables and [NI-PSP](#) data items.



[Data Communication Methods Home](#)

Binding Shared Variables and Other NI-PSP Data Items to Controls and Indicators

To [read and write data with controls and indicators](#), drag a shared variable from the Project Explorer window to the front panel of a VI to create a control bound to the shared variable. You also can specify more configuration options or read and write a shared variable in a separate project by completing the following steps.

1. Right-click a front panel object and select **Properties** from the shortcut menu to display the Properties dialog box for the object.
 2. Navigate to the **Data Binding** page.
 3. Select **Shared Variable Engine (NI-PSP)** from the **Data Binding Selection** list.
 4. Select the shared variable or **NI-PSP** data item you want to bind the front panel object to.
 1. Click the **Browse** button to display the Select Source Item dialog box.
 2. Choose from the following options in the **Network-Published Source** pull-down menu:
 - Select **Project Items** to bind the front panel object to a shared variable in the current project.
 - Select **Network Items** to bind the front panel object to a shared variable or other NI-PSP data item in another project.
- 
NOTE
 Your VI will break if you move the shared variable or data item or it becomes unavailable on the network.
3. Select the shared variable or NI-PSP data item and click the **OK** button.
 5. Under the **Access Type** pull-down menu of the **Data Binding** page, choose from the following options to specify how the front panel object interacts with the shared variable or data item:
 - Select **Read only** to read from the shared variable or data item.
 - Select **Write only** to write to the shared variable or data item.
 - Select **Read/Write** to read from and write to the shared variable or data item.
 6. Refer to the **Troubleshooting Front Panel Data Binding** topic for information about ensuring that the connection is successful.

Variable Identifier URLs

LabVIEW identifies each shared variable with a variable identifier URL. To **read and write shared variables programmatically**, you can specify the variable identifier URLs of those shared variables with the **shared variable refnum in** input of the Shared Variable functions.



NOTE

You also can **search for shared variables programmatically** to read and write multiple shared variables with relatively little LabVIEW code.

Shared Variable Architecture

The following list summarizes the set of LabVIEW constructs that relate to shared variables and variable identifier URLs.

- **Shared Variable** —A memory space that you can read data from and write data to.

- **PSP Variable** —A data item published on the network using the [NI Publish-Subscribe Protocol \(NI-PSP\)](#). Common examples of PSP variables include network-published shared variables, I/O variables with network-publishing enabled, and DAQ channels published through the NI-DAQmx I/O server.
- **I/O Variables** —Shared variables that use the [NI Scan Engine](#) for single-point access to I/O data.
- **Variable Container** —A set of variables. The most common type of variable container is a [project library](#). However, other items such as I/O modules, Wireless Sensor Network (WSN) nodes, and EtherCAT devices also can serve as variable containers. You can use the Browse Variable Object dialog box to find all the variable containers currently deployed on the network.
- **Variable Engine** —A run-time component that manages a hierarchy of variable containers and variables.
- **Project Library** —A collection of VIs, type definitions, shared variables, palette files, and other files, including other project libraries. A project library that includes shared variables is a variable container, but not all variable containers are project libraries.
- **Target Name** —The name assigned to a target item, such as My Computer or an RT target, in a LabVIEW project.

Note (Real-Time Module) You can use the **Name** field on the General Properties page of the RT Target Properties dialog box to specify an RT target name.

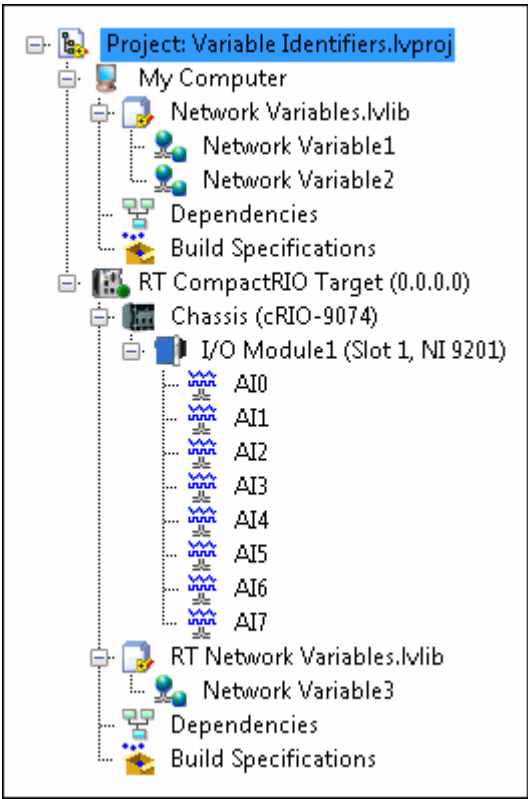
Variable Identifier URL Syntax

The URL syntax of a variable identifier is:

[Variable Engine]://[Host Name]/[Container Name]/[Variable Name]

You must specify the Container Name and Variable Name components for all variable identifier URLs. However, you can omit the Variable Engine if it is implied by the class. You also can omit the Host Name if the variables are hosted on the same computer where the VI runs.

For example, the following LabVIEW project contains network-published shared variables, which are listed under the **Network Variables.lvlib** container, and I/O variables, which are listed under the **I/O Module1** container.



The following table includes the full variable identifier URLs for several of the variables in this project.

Variable Name	Variable Type	Variable Identifier URL
Network Variable1	Network-Published Shared Variable	ni.var.psp://My Computer/Network Variables/Network Variable1
AI0	I/O Variable	ni.var.io://RT CompactRIO Target/I%2F0 Module1/AI0
Network Variable3	Network-Published Shared Variable	ni.var.psp://RT CompactRIO Target/RT Network Variables/Network Variable3

The variable identifier URL is not case sensitive. However, you must replace reserved characters with the corresponding *escape codes* to prevent parsing errors. For example, you must replace the `<` character with the escape code `%2F`.

The Variable Engine component of the URL depends on the type of variable you plan to read or write. The following table includes the Variable Engine URL for each type of shared variable that the Shared Variable functions support.

Type	Variable Engine URL
Network-published shared variable, I/O variable, or I/O alias (remote access)	<code>ni.var.psp</code>
I/O variable or I/O alias (local access)	<code>ni.var.io</code>

You can use either the PSP Variable Engine or the I/O Variable Engine to access a network-published I/O variable or I/O alias. Use the I/O Variable Engine `ni.var.io` to read or write a network-published I/O variable or I/O alias locally. Use the PSP Variable Engine `ni.var.psp` to read or write a network-published I/O variable or I/O alias remotely.

The Variable Engine component of the URL is implied and therefore optional when using one of the following classes:

- PSP Variable Engine
- I/O Variable Engine
- PSP Variable Container
- I/O Variable Container
- PSP Variable
- I/O Variable

In this case, you can omit the Variable Engine component of the URL and use the following URL syntax:

```
//[Host Name]/[Container Name]/[Variable Name]
```

If you are using a generic class that does not imply the Variable Engine and you do not specify the Variable Engine, LabVIEW attempts to resolve the variable identifier URL by searching variable engines in the following order of precedence.

1. I/O Variable Engine
2. PSP Variable Engine

In this case, if a matching URL exists in multiple engines, LabVIEW uses the first matching URL.

The [Host Name] component of the variable identifier URL is optional. You can specify the host name as a DNS name, a target name, or an IP address. However, if you do not specify a host name in the variable identifier URL, LabVIEW assumes the host is the same computer where you run the VI. You also can use the string `localhost` to explicitly specify the computer where you run the VI. To avoid name collisions, use unique names whenever possible.

If you do not need to specify a host name, you can use the following URL syntax:

```
/[Container Name]/[Variable Name]
```

If the host name you specify can be interpreted in more than one way, for example both as a project name and as a DNS name, LabVIEW searches for a valid interpretation of the specified host name in the following order of precedence.

- 1. Target name (as specified in the LabVIEW project)
- 2. DNS name
- 3. IP address

The name of each shared variable in a container must be unique. However, because the variable identity includes the container name, variables in different containers can share the same variable name and still maintain unique variable identities.

Variable Identifier URL Escape Codes

Use escape codes to substitute for reserved characters when specifying a *variable identifier URL*. The following table includes the escape codes for each reserved character.



NOTE
The set of reserved characters for variable identifier URLs is slightly different than the set of reserved characters specified by the Uniform Resource Identifier (URI) Generic Syntax specification (RFC 3986).

Character	Escape Code
#	%23
%	%25
[	%5B
]	%5D
;	%3B
/	%2F
?	%3F
:	%3A
@	%40
&	%26
\$	%24
!	%21
*	%2A
+	%2B

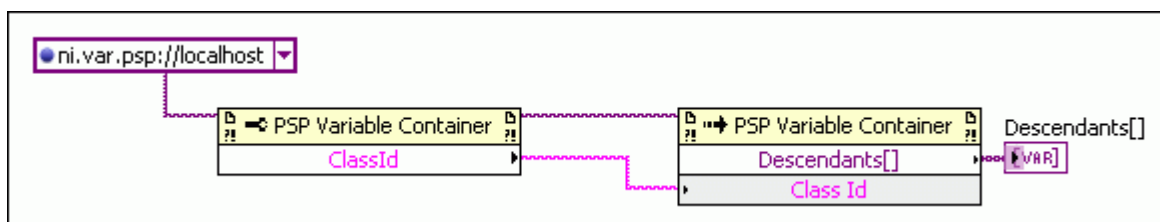
Character	Escape Code
=	%3D

Using Variable Methods

To use some variable methods, such as the Find Descendants and Is Kind Of methods, you must specify the Class Identifier of a Variable class with the **Class Identifier** input. The following table shows the Class Identifiers of the Variable classes included in LabVIEW.

Class Name	Class Identifier
Variable Object	VariableObject
Variable	Variable
PSP Variable	PSPVariable
I/O Variable	IOVariable
Variable	Variable
Variable Container	VariableContainer
PSP Variable Container	PSPVariableContainer
I/O Variable Container	IOVariableContainer
Variable Engine	VariableEngine
PSP Variable Engine	PSPVariableEngine
I/O Variable Engine	IOVariableEngine

You also can use the Class Identifier property of any class that inherits from Variable Object to obtain the Class Identifier programmatically. For example, in the following figure, the Class Identifier property passes the Class Identifier of the PSP Variable Container to the Find Descendants method. The Find Descendants method then returns an array of references to the PSP variable containers on the local host.



Writing Shared Variables with Existing Data Sources

When you bind a shared variable to an existing data source, the shared variable receives the same updates as that data source. Possible data sources for shared variables include DAQ channels; [NI Publish-Subscribe Protocol \(NI-PSP\)](#) data items, such as other shared variables; and data items from I/O servers that are defined in a separate project.



NOTE

- You can bind only network-published shared variables.
- You can bind shared variables only to data items that are currently [deployed](#).
- If you write less than 8 kilobytes of data to a shared variable, LabVIEW can take up to 10 milliseconds to send that data over the network. To eliminate this delay, use the Flush Shared Variable Data VI to [send the shared variable data immediately](#).

Bind a shared variable to an existing data source if you need to leverage shared variable features for a data source that does not support those features. For example, assume that you need to set logging and alarming properties for a shared variable on a CompactRIO target. Because real-time targets do not support logging and alarming, you must create a new shared variable on your development computer, bind that shared variable to the shared variable on the CompactRIO target, and configure the shared variable on the development computer for logging and alarming. When the shared variable on the CompactRIO target sends new data to the Shared Variable Engine (SVE), the SVE updates the value of the shared variable on your development computer, and that shared variable performs the logging and alarming tasks.



Note When you bind a shared variable to an existing data source across a network, LabVIEW does not track configuration changes to the data source. Therefore, in the example above, if you change the name of the variable on the CompactRIO target, you also must update the configuration of the shared variable on your development computer, or that shared variable breaks.

Complete the following steps to bind a shared variable to an existing data source.

1. [Create a shared variable](#) or right-click a shared variable in the [Project Explorer](#) window and select **Properties** from the shortcut menu to display the Shared Variable Properties dialog box.
2. Place a checkmark in the **Enable Aliasing** checkbox.
3. Choose from the following options in the **Bind to** pull-down menu:
 - Select **Project Variable** to bind the shared variable to an NI-PSP data item in the current project.
 - Select **PSP URL** to bind the shared variable to an NI-PSP data item on the network.

Note If the data item is on a computer that is not on the local subnet, you must select **Tools»Shared Variable»Register Computer** to register that computer. After you specify the computer name or the IP address of the computer, you can bind shared variables to data sources on that computer.

- Select **I/O Variable** to bind the shared variable to a data item on an I/O server, such as an OPC server.
- 4. Select the data source to which you want to bind the shared variable using one of the following methods:
 - Click the **Browse** button to browse for the data source with the Browse for Variable dialog box.
 - Enter the *network path* of the data source in the **Path** text box.
- 5. Click the **OK** button.



Sharing Live Data Programmatically

You can *use the DataSocket VI and functions* to programmatically configure, read, or write live data.

(Real-Time, Windows) National Instruments recommends that you *wire a Shared Variable node* into VIs and functions to share live data programmatically.



NOTE

You can create, configure, and host shared variables only on Windows or RT targets with the LabVIEW Real Time Module. Use the DataSocket VI and functions to read or write shared variables from other platforms.

Using Shared Variables

Use *shared variables* to share data among VIs or between locations on the block diagram that you cannot connect with wires. You can share data using shared variables with little or no block diagram programming.

For the DataSocket VI and functions, you read or write data by specifying a URL in much the same way you specify URLs in a Web browser.

Specifying a URL

URLs use communication protocols, such as `opc`, `ftp`, and `http` on Windows, and `psp`, `dstp`, and `file` on all LabVIEW-supported platforms, to transfer data. The protocol you use in a URL depends on the type of data you want to write and how you configure your network.

You can use the following protocols when you read or write data using the DataSocket VI and functions or when you *enable a connection for a control* :

- **NI Publish-Subscribe Protocol** `psp` —National Instruments technology for transmitting data between the network and your local computer. When you use this protocol, the VI communicates with the *Shared Variable Engine* (SVE). Use the `psp` protocol to connect to *shared variables* and server or device data items. You must provide a name for the data item, which is appended to the URL. The data connection uses that name to address a particular data item in the SVE. You also can use this protocol when *using front panel data binding*. You can use the `fieldpoint` protocol as an alias for NI Publish-Subscribe Protocol (NI-PSP).
- **DataSocket Transport Protocol** `dstp` —When you use this protocol, the VI communicates with the DataSocket Server. You must provide a named tag for the data, which is appended to the URL. The data connection uses the named tag to address a particular data item on a DataSocket Server. To use this protocol, you must run a DataSocket Server.
- (Windows) **OLE for Process Control** `opc` —Designed specifically for sharing real-time production data, such as data generated by industrial automation operations. To use this protocol, you must run an OPC server.
- (Windows) **File Transfer Protocol** `ftp` —You can use this protocol to specify a file from which to read data from an FTP server.

Note To read a text file from an FTP site using DataSocket functions, add [`text`] to the end of the URL.

- `file` —You can use this protocol to provide a link to a local or network file that contains data.
- (Windows) `http` —You can use this protocol to provide a link to a Web page that contains data.

Note To read a text file via the `http` protocol using DataSocket functions, add [`text`] to the end of the URL.

The following table shows examples of each protocol URL.

URL	Example
psp	For shared variables:
	<code>psp://computer/library/shared_variable</code>
	For NI-PSP data items, such as server and device data items:
	<code>psp://computer/process/data_item</code> <code>fieldpoint://host/FP/module/channel</code>

URL	Example
dstp	dstp://servername.com/numeric, where numeric is the named tag for the data
opc	opc:\National Instruments.OPCTest\item1 opc:\\computer\National Instruments.OPCModbus\Modbus Demo Box.4:0 opc:\\computer\National Instruments.OPCModbus\Modbus Demo Box.4:0?updaterate=100&deadband=0.7
ftp	ftp://ftp.ni.com/datasocket/ping.wav ftp://ftp.ni.com/support/00README.txt[text]
file	file:ping.wav file:c:\mydata\ping.wav file:\\computer\mydata\ping.wav
http	http://ni.com

Use the psp, dstp, and opc URLs to share live data because these protocols can update remote and local controls and indicators. Use the ftp and file URLs to read data from files because these protocols cannot update remote and local controls and indicators.

Data Formats Supported by the DataSocket VI and Functions

You can pass data of any LabVIEW data type using the DataSocket VI and functions. You also can use the DataSocket VI and functions to read and write the following data:

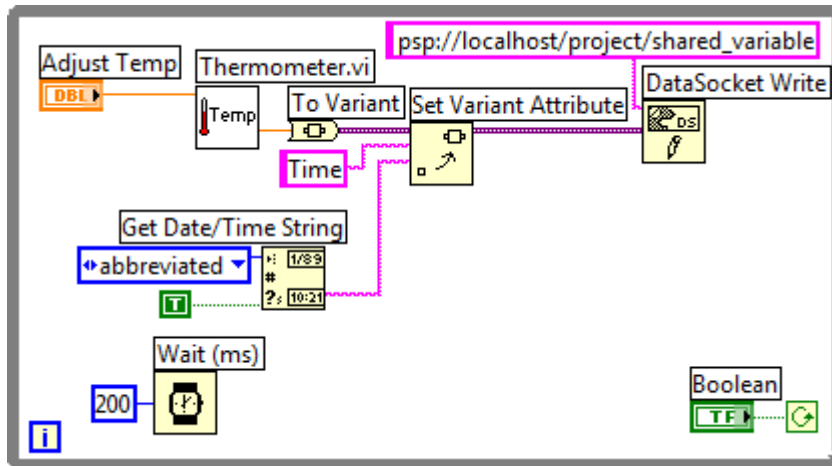
- **Raw text** —Use raw text to deliver a string to a string indicator.
- **Tabbed text** —Use tabbed text, as in a spreadsheet, to write data in arrays. LabVIEW interprets tabbed text as an array of data.
- **.wav data** —Use .wav data to write a sound to a VI or function.
- **Variant data** —Use variant data to read data from another application, such as a National Instruments Measurement Studio ActiveX control.

DataSocket Functions and Variant Data

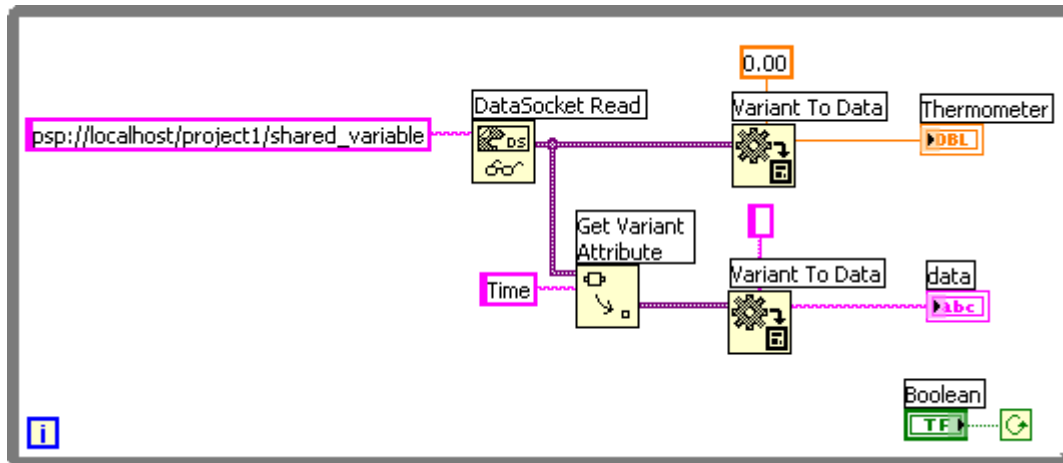
Use variant data when a VI or other application that programmatically reads data cannot convert the data back into its original data type, such as when you read data from another application. You also can use variant data to add an attribute, such as a time stamp or

warning, to data you read or write when the data types do not permit this additional attribute.

You can use the To Variant function to convert the data you write to a data connection to variant data programmatically. The following block diagram continuously acquires a temperature reading, converts the data to variant data, and adds a time stamp as an attribute to the data.



When another VI reads the live data, the VI must convert the variant data to a data type it can manipulate. The following block diagram continually reads temperature data, converts the variant data into a temperature reading, retrieves the time stamp attribute associated with each reading, and displays the temperature and the time stamp on the front panel.

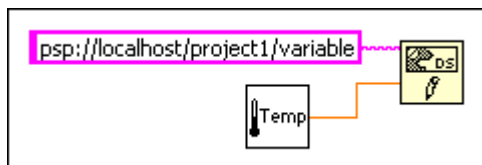


Reading and Writing Live Data Using DataSocket Functions

From the block diagram, you can programmatically read or write data using the DataSocket functions or (Real-Time, Windows) using a Shared Variable node, which is useful for customizing or dynamically changing the data connection. You can use the DataSocket functions with shared variables to start and stop reading or writing shared variables programmatically, to specify a buffer size, or to dynamically determine which

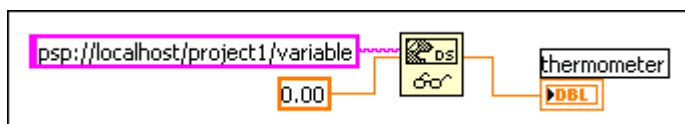
shared variable to read. You can wire a shared variable control into DataSocket functions to select the input shared variable.

Use the DataSocket Write function to write live data programmatically. The following block diagram shows how to write a numeric value.



The DataSocket Write function is polymorphic, so the data you write to the connection can be in any format or the LabVIEW data type.

Use the DataSocket Read function to read live data programmatically. The following block diagram shows how to read data and convert it to a double-precision floating-point number.



Convert live data to a specific data type by wiring a control or constant to the **type** input of the DataSocket Read function. If you do not specify a type, the **data** output of the DataSocket Read function returns variant data, which you can manipulate with the Variant To Data function.

Programmatically Opening and Closing Data Connections

Use the DataSocket Open and DataSocket Close functions to control when a data connection opens and closes. When you open a data connection using the DataSocket Open function, the connection remains open until one of the following conditions is met: you explicitly close the connection using the DataSocket Close function, you close the VI, or the VI stops running. If a VI stops running, the data connections of any subVI also will close. The **URL** input of the DataSocket Open function accepts only valid URLs. The DataSocket Open function returns a data connection refnum that you can use as the **URL** input for the DataSocket Read and DataSocket Write functions.

Refer to the VIs in the `labview\examples\Data Communication\DataSocket\Simple DataSocket\Simple DataSocket.lvproj` for examples of using DataSocket functions for data connections.



RELATED INFORMATION

[Reading and Writing Shared Variables with Shared Variable Nodes](#) on page 629

[Publishing Latest Values with Shared Variables](#) on page 612

[I/O Controls and Indicators](#) on page 1048

[Writing Live Data Programmatically](#) on page 1529

[Block Diagram Objects](#) on page 126

[Sharing Live Data Programmatically](#) on page 641

[Reading Live Data Programmatically](#) on page 1532

[Programmatically Opening and Closing Data Connections](#) on page 1529

[Sharing Live Data Programmatically](#) on page 641

Buffering Data

If you use network-published [shared variables](#) or a [psp](#) or [dstpURL](#) to share [data](#) programmatically, LabVIEW by default writes only the most recent value to all readers. When one client writes values to the server faster than another client reads them, newer values overwrite older, unprocessed values before the clients read them. If the reader does not receive a value before receiving the following value, the data is lost. This loss of unprocessed data can occur at the server or at the client. This loss of data might not be a problem if you are reading data and you want to receive only the most recent value written to the server. However, if you want to receive every value written to the server or if lossy transfers are unacceptable, you must buffer the data on the client. Place a checkmark in the **Use Buffering** checkbox on the Network page of the Shared Variable Properties dialog box to enable buffering for the shared variable.

Client-Side Buffering

Buffering allows you to lessen the potential timing difference between the reader and writer but does not guarantee data delivery. If the data in a buffer at the server or client exceeds the buffer size, the buffer discards older values in place of newer values. You can set the size of a buffer for a shared variable on the **Network** page of the Shared Variable Properties dialog box to set how many data packets the shared variable retains before overwriting old data.

You also can specify programmatically the maximum number of bytes and the maximum number of values, in terms of packets, that the buffer for each item can hold. National Instruments recommends that you specify both the Buffer Maximum Bytes and Buffer Maximum Packets properties when you buffer data. If incoming data exceeds the maximum number of bytes or the maximum number of values, LabVIEW removes older data from the buffer to accommodate the new data.

When configuring buffering programmatically, wire the written data to the Set Variant Attribute function to detect discarded values in a data stream by uniquely identifying each value in the writer and checking for discarded sequence IDs in the reader.

If you publish a shared variable with less than 8 kilobytes of data, it can take up to 10 milliseconds for LabVIEW to send that data over the network. The Flush Shared Variable Data VI allows you to flush the buffer immediately and eliminate this delay.

If you want the buffer-enabled client to receive all data, use the DataSocket Write function to *write live data* programmatically. Writing data from the front panel can result in data loss.



NOTE

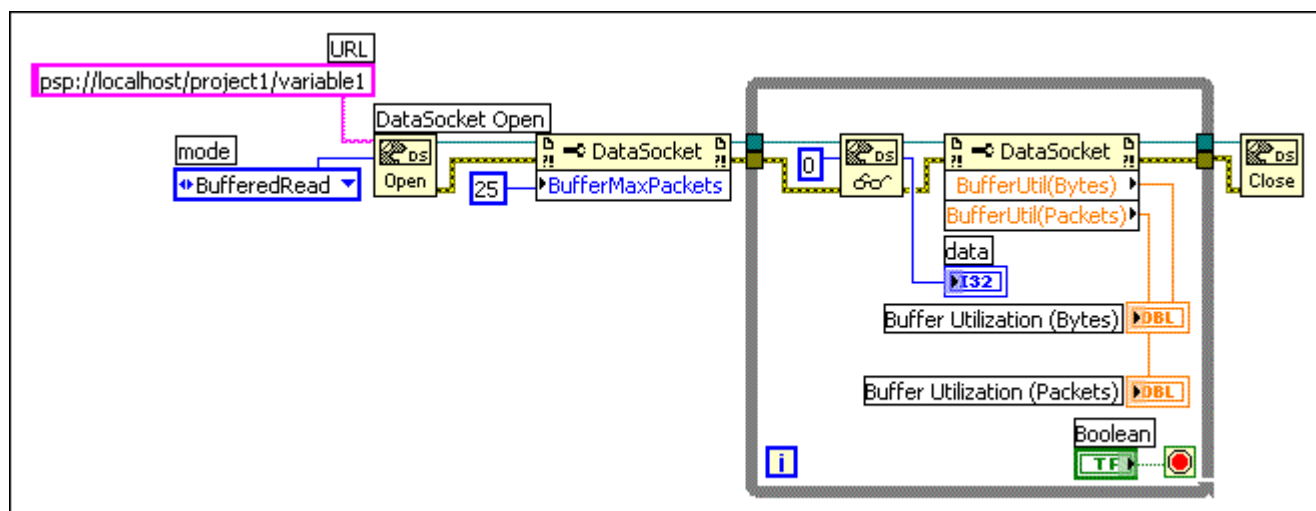
Client-side buffering also applies to other protocols, such as `opc` and `file`. However, in some cases these servers might *lose data*.

Set the **mode** input of the DataSocket Open function to **BufferedRead** or **BufferedReadWrite** and use a Property Node to *set the DataSocket properties for the size* of a first-in-first-out (FIFO) buffer. Doing so ensures that LabVIEW stores the values the client receives in a buffer rather than overwriting them every time the value changes.



Note If you use DataSocket properties to set the size of a FIFO buffer, you must set the mode input of the DataSocket Open function to **BufferedRead** or **BufferedReadWrite**. Otherwise, the item at the server is not buffered for the connection.

The following block diagram uses buffering programmatically.



You also can enable buffering when sharing data using the DataSocket functions by adding the string `?maxbytes=1000&maxpackets=10`, to the end of the URL, where 1000 is the maximum number of bytes the buffer contains and 10 is the maximum number of packets the buffer contains.



Note Buffering applies when you *use the DataSocket Read function* to read data a server writes. Buffering is not available when you use *front panel DataSocket data binding* to read data. Buffering is available for *front panel data binding* through the *Shared Variable Engine* if you bind controls to shared variables with buffering enabled on the *Network* page of the Shared Variable Properties dialog box.

Reporting Diagnostics Using DataSocket Properties

Use the Buffer Utilization (Bytes) property or the Buffer Utilization (Packets) property to *request diagnostic information* about the buffers you specify. Use these properties to check the percentage of buffer in use on the client to determine whether the current buffer size is sufficient. If the value of either of these properties approaches the maximum value of the buffer, *increase the buffer size* to make sure you receive all values the server writes. You also can *increase the rate* at which the While Loop containing the DataSocket functions executes to increase the read rate.

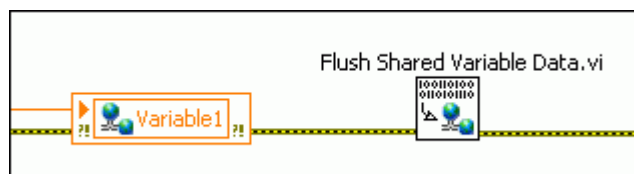
Server-Side Buffering

When you configure client-side buffering, servers might lose data in some cases if the server sends data faster than the network can accept. You can use server-side buffering for *NI Publish-Subscribe Protocol (NI-PSP)* or *ds tp* buffering. To use NI-PSP buffering, you must connect to an existing shared variable with buffering enabled. To use *ds tp* buffering, you also must use the DataSocket Server Manager to configure server-side buffering. Refer to the DataSocket Server Help for more information about server-side buffering for the DataSocket Server. For other protocols, refer to the server documentation for more information about server-side buffering for that server.

Sending Shared Variable Data over the Network Immediately

LabVIEW temporarily stores the data that you write to a shared variable in an 8 kilobyte buffer. LabVIEW sends that data over the network when the buffer is full or 10 milliseconds have passed. Because you incur overhead each time you send a data packet over the network, this design increases throughput by decreasing the number of data packets you send.

However, this design increases latency when you write less than 8 kilobytes of data to a shared variable. To eliminate the 10 millisecond delay and minimize latency, send all shared variable data over the network immediately by using the Flush Shared Variable Data VI after you write to the shared variable, as shown in the following figure.



The previous figure flushes the data that a Shared Variable node writes, but you also can flush shared variable data that you write with other methods. For example, if you are [writing shared variables programmatically](#), use the Flush Shared Variable Data VI after the Write Variable function.



NOTE

When you use the Flush Shared Variable Data VI, LabVIEW sends all of the shared variable data that has not yet transmitted over the network, including data that you write with all of the shared variables in other applications and other parts of the same application. Therefore, the Flush Shared Variable Data VI is useful when you need to minimize latency for all of the shared variable writers on a single computer. However, if some of those shared variables require high throughput rates, the Flush Shared Variable Data VI could affect those shared variables adversely.



[Data Communication Methods Home](#)

Making Shared Variables Available on a Network

You must deploy [shared variables](#) to the [Shared Variable Engine \(SVE\)](#) before you can read and write those shared variables over a network. Deploying a shared variable creates a memory space for the shared variable in the SVE. Undeploying a shared variable frees the memory resources allocated to that shared variable.

When you run a VI that reads or writes a shared variable with a Shared Variable node, LabVIEW automatically deploys the [project library](#) that contains that shared variable. However, LabVIEW does not automatically undeploy the project library when the application terminates. Therefore, you do not need to deploy shared variables before [reading and writing them with Shared Variable nodes](#), but you do need to undeploy those shared variables to free the memory resources allocated to those shared variables. Additionally, you must deploy shared variables explicitly if you are [reading and writing shared variables with other methods](#). The following list describes the methods you can use to deploy and undeploy shared variables.

- [Edit-time method](#) —Deploy and undeploy shared variables at edit-time when you are developing an application that accesses a small number of shared variables.
- [Run-time method](#) —Deploy and undeploy shared variables at run-time when you plan to distribute an application as a stand-alone application. Deploying and undeploying shared variables at run-time also is useful if your application accesses a large number shared variables.



NOTE

If you plan to distribute a stand-alone application that uses shared variables, do not include the `.lvlib` file in an LLB or in the executable. Use the Source File Settings page of the Application Properties dialog box to change the **Destination** of the `.lvlib` file to a destination outside the executable or LLB.

Deploying and Undeploying Shared Variables at Edit-Time

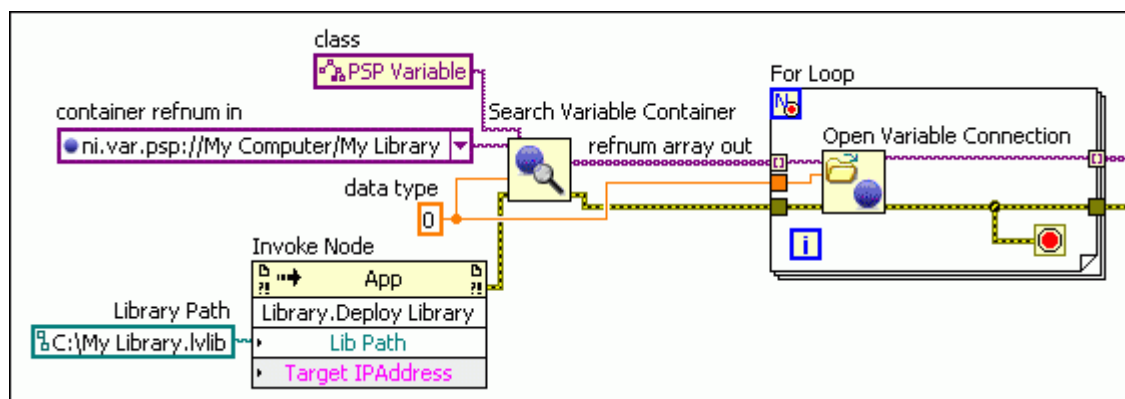
To deploy a shared variable, right-click the project library in the [Project Explorer](#) window that contains the shared variable and select **Deploy**. This action deploys all of the shared variables in the project library to the SVE. Perform this action before running applications that read and write shared variables with methods other than using Shared Variable nodes.

To undeploy a shared variable, right-click the project library that contains the shared variable and select **Undeploy**.

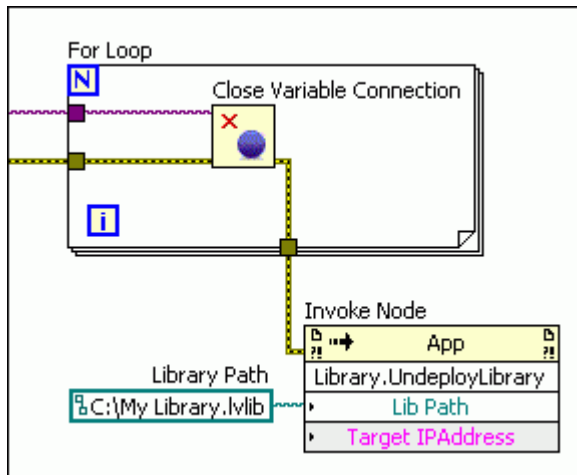
Deploying and Undeploying Shared Variables at Run-Time

(Windows) To deploy shared variables at run-time, place a checkmark in the **Deploy shared variable libraries at application execution** checkbox on the Shared Variable Deployment page of the Application Properties dialog box.

You also can use the Deploy Library and Undeploy Library methods on any platform to deploy and undeploy libraries that contain shared variables. For example, in the following figure, the Deploy Library method deploys `My Library.lvlib` before the Search Variable Container function searches for the shared variables in that project library. The Open Variable Connection function then opens connections to those shared variables.



In the following figure, the Undeploy Library method undeploys `My Library.lvlib` after the Close Variable Connection function closes connections to the shared variables in that project library.



Data Transfer in LabVIEW

Transferring Data

Refer to the [Data Communication Methods in LabVIEW](#) topic for guidance on selecting a data communication method in LabVIEW. You also can refer to the following books in the LabVIEW Help for information about specific data transfer use cases:

- [Transferring Data over a Network](#)
- [Communicating Data between Parallel Sections of Code Using Channel Wires](#)
- [Sharing Data in Deterministic Applications \(Real-Time Module\)](#)
- [Storing and Transferring Data \(FPGA Module\)](#)



RELATED INFORMATION

[Data Communication Methods in LabVIEW](#) on page 651

[Choosing a Method of Reading and Writing Shared Variables](#) on page 619

[Communicating Data between Parallel Sections of Code Using Channel Wires](#) on page 742

Data Communication Methods in LabVIEW

LabVIEW contains many data communication methods, each suited for a certain use case. To ensure you implement the correct data communication method for your application, refer to the overview presented in this topic. For detailed information about a particular data communication method, click the title of the data communication method you want to learn more about.

Data Communication Type:	<i>Dataflow Elements</i>	<i>Buffer Interfaces</i>	<i>Variable Interfaces</i>
Use Case:	Sending data between most block diagram objects.	Sending data between locations when it is important to avoid overwriting or losing any value.	Storing a readily-accessible value in memory where the latest value is of primary interest. <i>Use carefully to avoid race conditions.</i>
Examples:	Passing data from a VI output to a VI input.	Acquiring a waveform; sending a message or command; streaming an image.	Monitoring the current temperature; monitoring the system state.
Categories:	— Wire — Shift register — Feedback Node — Channel Wire	<i>Intra-Target or PC</i> — Queue — RT FIFO — User events — Asynchronous Message Communication (AMC) Reference Library <i>Networked</i> — Network streams — TCP — Simple TCP Messaging (STM) Reference Library <i>Intra-FPGA</i> — Target-scoped FIFO — VI-defined FIFO — DRAM <i>FPGA to/from Host</i> — DMA FIFO <i>FPGA to FPGA</i> — Peer-to-Peer (P2P) FIFO	<i>Intra-Target or PC</i> — Local variable — Global variable — Functional global variable (FGV) — Single-process shared variable — Data value reference — Current Value Table (CVT) <i>Networked</i> — Network-published shared variable — Programmatic Shared Variable API — Web services — UDP <i>Intra-FPGA</i> — Global variable — Memory item — Register item <i>FPGA to/from Host</i> — Read/Write Control — NI Scan Engine and Variables — User-defined I/O variable

Dataflow Elements

To take advantage of the *dataflow programming environment* in LabVIEW, you should use wires, shift registers, feedback nodes, and channel wires as the primary data communication methods in your application.

Method	Attributes
<i>Wire</i>	Primary data transfer between block diagram objects
<i>Shift register</i>	Primary data transfer between loop iterations
<i>Feedback Node</i>	Primary data transfer between iterations of code (does not require a loop)
<i>Channel Wire</i>	Primary data transfer for asynchronous communication between parallel sections of code

Buffer Interfaces

Use buffer interfaces when you must send data from one location to another and you do not want to overwrite or lose any value. A buffer interface is often represented as a FIFO (first-in, first-out) single or multi-element buffer where one or more writers add data onto one end of the buffer and a reader retrieves data from the other end.

Intra-Target or PC

Method	Attributes
Queue	- Flexible sizing and access options - Non-deterministic
RT FIFO (Real-Time Module)	- Deterministic - Polling mode (default) handles read/write wait times deterministically, but creates high CPU overhead - Blocking mode optimizes CPU utilization during read/write wait times, but does not handle the operations deterministically - Less flexible than queues because of fixed size and limited access options
<i>User events</i>	<i>Event-driven communication</i> between different parts of an application - Respond to programmatic changes on objects
Asynchronous Messaging Communication (AMC) Library (Available at ni.com)	- Message communication within a process and between processes - Greater ease of use compared to queues when designing a messaging framework - Includes the Queued Message Handler (QMH) template Note: This is community content subject to the Sample Code License Terms available at ni.com/samplecode/license

Networked

Method	Attributes
<i>Network streams</i>	• 1:1 communication • Simpler to implement than TCP • Automatically handle disconnections and reconnections • Unidirectional • Available only for Windows and Real-Time systems
<i>TCP</i>	• Broad functionality for applications that require network communication to devices without network streams support • Low-level protocol introduces implementation complexity
Simple TCP Messaging (STM) Reference Library (Available at ni.com)	• Message wrapper around low-level TCP functions • Used to implement 1:N, single server to multiple client communication Note: This is community content subject to the Sample Code License Terms available at ni.com/samplecode/license

Intra-FPGA

Method	Attributes
Target-scoped FIFO (FPGA Module)	• Implementation options specify the type of storage the FIFO uses: Flip-flops, Look-Up Table, or Block Memory
VI-defined FIFO (FPGA Module)	• Implementation options specify the type of storage the FIFO uses: Flip-flops, Look-Up Table, or Block Memory • Avoids resource conflicts in reentrant subVIs by replicating the resource for each instance • Facilitates reusing code for VIs in multiple projects • Not viewable or customizable from the Project Explorer window
DRAM (FPGA Module)	• Interface with dynamic random access memory (DRAM) using a memory item or CLIP interface • Provides a large amount of buffered memory space compared to Target-scoped and VI-defined FIFOs • Nondeterministic access latency • Not available on all targets

FPGA to/from Host

Method	Attributes
DMA FIFO (FPGA Module)	Transfers large amounts of data to/from host using Direct Memory Access (DMA)

FPGA to FPGA

Method	Attributes
Peer-to-Peer (P2P) FIFO (FPGA Module)	1:1 communication between targets

Variable Interfaces

Use variable interfaces when you must store a value in memory that readers and writers can access with minimal or no flow control. Note that you should use variables in cases where the latest value is of primary interest and it is acceptable to overwrite previous values. *Use carefully to avoid race conditions.*

Intra-Target or PC

Method	Attributes
<i>Local variable</i>	Stores data accessible from a single VI
<i>Global variable</i>	Stores data accessible from multiple VIs in the same LabVIEW application instance
<i>Functional global variable (FGV)</i>	- VI that uses uninitialized shift registers to store global data - Capable of implementing simple global variables, as well as more complex data structures, such as stacks or queue buffers - Helps prevent race conditions by defining specific operations that calling code can perform on the variable
<i>Single-process shared variable</i>	- Deterministic option available - Scalable to network-published use case
<i>Data value reference</i>	- Stores large data sets - Helps manage memory and avoid data copies

Method	Attributes
Current Value Table (CVT) (Available at ni.com)	- Common data repository for variables in applications with separate components - Uses less computing resources than shared variables by centralizing I/O operations Note: This is community content subject to the Sample Code License Terms available at ni.com/samplecode/license

Networked

Method	Attributes
<i>Network-published shared variable</i>	- Deterministic option available - Ideal for monitoring values - Stores values in a network data server (Shared Variable Engine) - Built-in connection manager for 1:N and N:1 network configurations
<i>Programmatic Shared Variable API</i>	- Improves design scalability by providing dynamic access to variables instead of static node-based binding - Access multiple variables with a single variable API function - Access network-published I/O variables
<i>Web services</i>	- Network data communication using standard HTTP protocols - Stream standard MIME data types, such as text, images, and videos - Build and deploy web services using LabVIEW VIs
<i>UDP</i>	- Supports single-cast (1:1) and multicast (1:N) for low-level communication among processes on devices - Higher performance than TCP in applications that do not require lossless data transfer - Low-level protocol introduces implementation complexity - Does not guarantee clients receive data

Intra-FPGA

Method	Attributes
Global variable	Stores data accessible from multiple VIs in the same LabVIEW application instance

Method	Attributes
Memory item (FPGA Module)	• Capable of storing values as multiple data points • Implementation options specify the type of storage the memory item uses: Block Memory, Look-Up Table, or DRAM • Target-scoped (configured in the project) and VI-defined (configured in a VI for reentrant use) options
Register item (FPGA Module)	• Capable of posting a status every clock cycle • Target-scoped (configured in the project) and VI-defined (configured in a VI for reentrant use) options • Consume fewer FPGA resources than FIFOs

FPGA to/from Host

Method	Attributes
Read/Write Control (FPGA Module)	• Reads or writes to a control or indicator in the top-level FPGA VI on the FPGA target.
NI Scan Engine and Variables (FPGA Module + NI Scan Engine support on your target)	• Simple, periodic communication of FPGA I/O data to/from an RT host • Reduces code required to transfer I/O data between an FPGA target and RT host
<i>User-defined I/O variable</i> (FPGA Module)	• Send user-defined data when using the Scan Engine in Hybrid mode • Data transfer between an FPGA VI and LabVIEW RT host VI • Transfer custom FPGA-processed I/O data

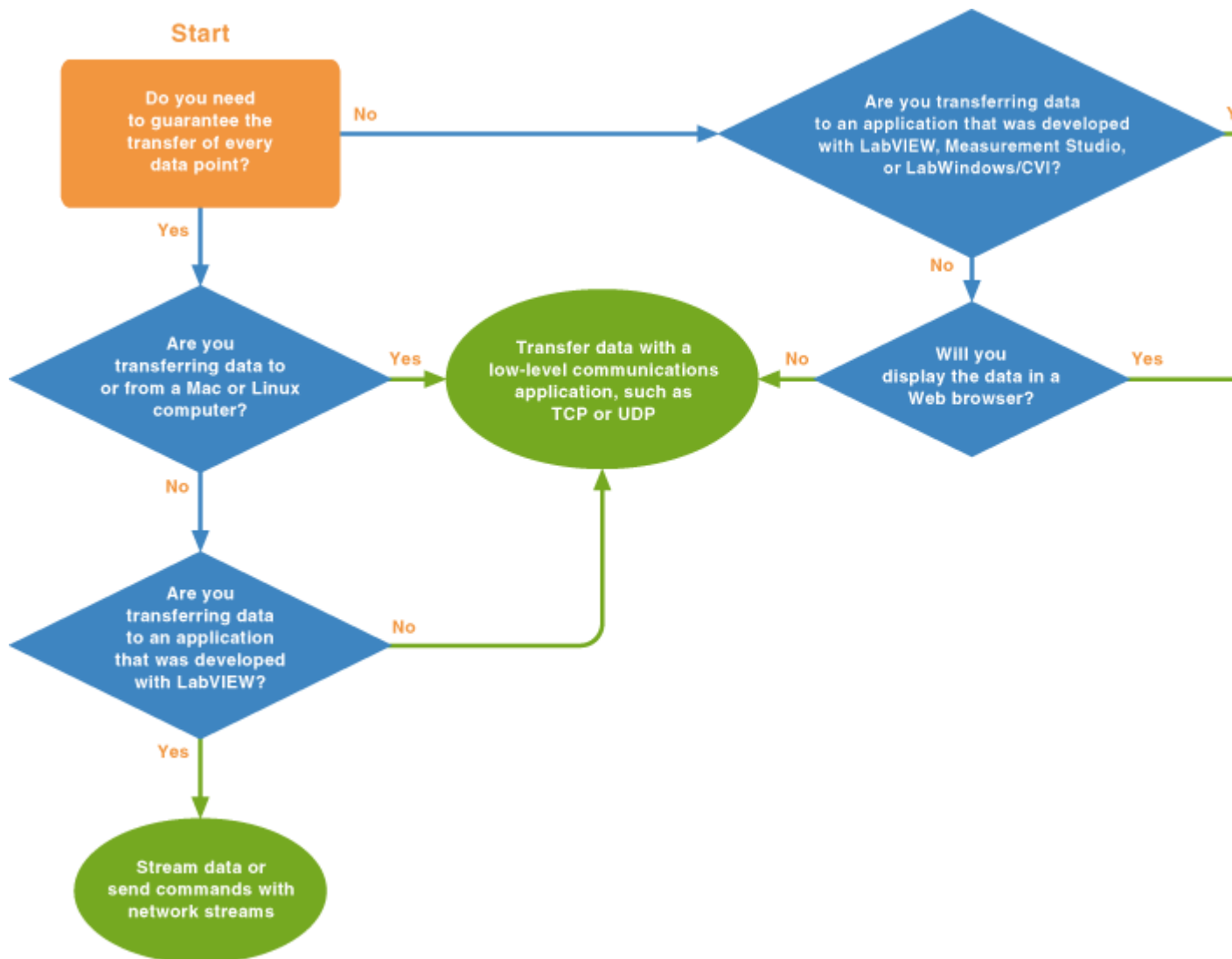
Transferring Data over a Network

When choosing networking features for your applications, the most important factors are the communication models that your applications use. Different communication models have different data transfer requirements that different networking features are designed to fulfill. The following table describes the most common communication models and recommends a networking feature for each.

Communication Model	Description	Example	Recommended Networking Feature
Process data	The transfer of latest values only, typically from one or more targets to a host computer.	An application in which an embedded controller acquires temperature data and periodically transfers a data point to the host computer so users can monitor the state of the system.	<i>Shared variables</i>
<i>Streaming</i>	The high-throughput transfer of every data point, typically from one computer that writes data to another computer that reads and displays or processes that data.	An application in which an embedded controller acquires vibration data from a motor and streams every point of that data to a host computer for analysis and processing.	<i>Network streams</i> *
<i>Command-based</i>	Low-latency data transfer from one computer, known as the commander, that triggers a specific event on another computer, the worker. Command-based communication is typically infrequent and requires that you do not miss any data points.	When a user clicks the stop button in a human-machine interface (HMI) application, that application shuts down a conveyer belt by sending the stop command to the CompactRIO target controlling the conveyer belt.	Network streams*

* Available only for Windows and Real-Time systems.

Depending on the OSes your computers run and the applications you are transferring data to, you might need to use networking features other than those outlined in the table above. Use the following flowchart to determine the networking features that best fit your application needs.



You also can use the following networking features with LabVIEW:

- Establish a wireless connection to a Bluetooth device using the Bluetooth VI and functions.
- Establish a wireless connection to an Infrared Data Association (IrDA) device using the IrDA functions.
- View and control a VI front panel remotely using [Remote Front Panels](#) on the [LabVIEW Web Server](#).
- Share live data from an industrial network by [connecting to OPC systems](#).



NOTE

If your application automatically loads on system startup, as is typical for real-time applications, your application may start before the network is fully configured. Network connections may fail until the system can route connection requests to a network interface. Consider using the Wait for Configured Network VI to delay automatic network connection requests until the system can route requests.

LabVIEW as a Network Client and Server

You can use LabVIEW as a client to subscribe to data and use features in other applications or as a server to make LabVIEW features available to other applications. You can use the [VI Server](#) to control VIs on local and remote computers by accessing properties and invoking methods using the Property Node and the Invoke Node, respectively.

Before you can communicate between computers, you must establish the network protocol you use to communicate. Protocols you can use include HTTP and TCP/IP. The protocol you select depends on the application. For example, the HTTP protocol is ideal for publishing on the Web, but you cannot use the HTTP protocol to build a VI that listens for data that another VI creates. To do that, use the TCP protocol. LabVIEW supports several [low-level protocols](#) you can use to communicate between computers.

(Windows) You can use [ActiveX technology](#) with LabVIEW as an ActiveX server or client.

Streaming Data and Sending Commands between Applications

[Streaming data](#) and [sending commands](#) are [models of communication](#) for sending data between two LabVIEW applications. Use network streams to accomplish these tasks. A network stream is a lossless, unidirectional, one-to-one communication channel that consists of a writer and a reader [endpoint](#). Use the Network Streams functions and the Network Stream Endpoint properties to design applications using network streams.



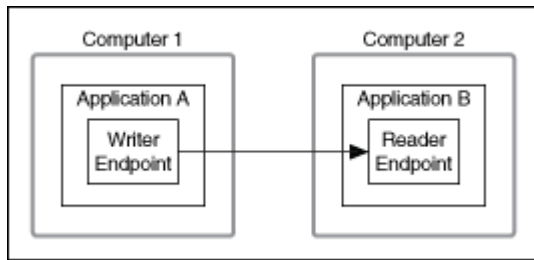
NOTE

You can use network streams on Windows and Real-Time systems only.

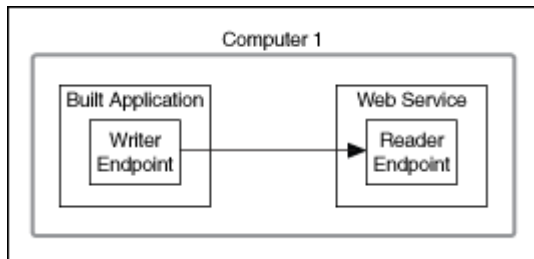
You can use network streams to transfer any LabVIEW data type except LabVIEW classes and most refnums. The only refnum you can transfer with network streams is the Vision Image data type. The following data types transfer at the fastest rates:

- Numeric scalars
- Booleans
- 1D arrays of numeric scalars
- 1D arrays of Booleans

The following figures illustrate some of the ways you can configure network streams to transfer data.



In the figure above, a network stream transfers data between two applications on different computers.

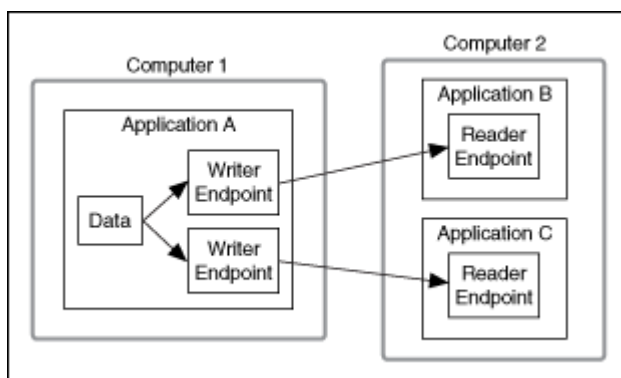


In the figure above, a network stream transfers data between a built application and a Web service.

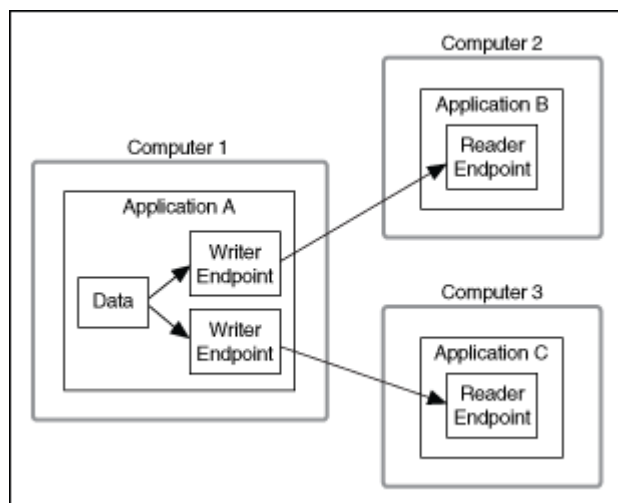


Note To facilitate network communication, network streams require more overhead than non-networked data transfer methods. To transfer data between loops within a single application, National Instruments recommends using the following methods:

- Queue Operations functions
- (Real-Time Module) RT FIFO functions
- (Real-Time Module) Shared variables with RT FIFO enabled



In the figure above, a network stream transfers data between an application on one computer to two applications on another computer.

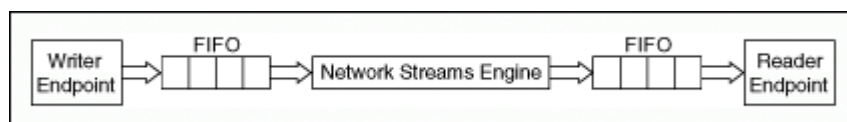


In the figure above, a network stream transfers data from an application on one computer to two applications on different computers.

Network Streams Engine

Each endpoint uses a FIFO buffer to transfer data. The Network Streams Engine (NSE) uses LogosXT to transfer data from the FIFO buffer on the writer endpoint to the FIFO buffer on the reader endpoint.

The following figure illustrates the flow of data in a network stream.



In the figure above, data flows in the following order.

1. The writer endpoint writes data to a FIFO buffer.
2. The NSE transfers data over a network to another FIFO buffer on the reader endpoint.
3. The reader endpoint reads the data from that FIFO buffer.

Determining When to Use Network Streams Instead of Shared Variables

Use [shared variables](#) to publish the latest value in a data set to many computers. Conversely, use network streams to log every point of data on one computer.

For example, assume that you are using an accelerometer to detect the vibrations of a pump that is re-pressuring natural gas in a pipeline. You are processing the vibration data on a CompactRIO target to monitor for bearing fault to ensure that the pump does not fail. However, the CompactRIO target does not have enough memory to analyze the data. Therefore, you must send the data to a desktop computer that has enough memory to store, analyze, and display that data.

Because shared variables are optimized for publishing the latest value of data only, they could miss a critical data point. However, network streams would stream every point of data to the desktop computer so you could monitor the condition of the engine.



Note Network streams can induce jitter in real-time (RT), time-critical loops. Therefore, if you want to stream data from a time-critical loop with network streams, National Instruments recommends that you first share the data with a lower-priority loop. Then, use network streams to stream the data to another application.

Designing Applications Using Network Streams

You can design applications for Windows and Real-Time systems to perform the following tasks using network streams:

- Stream data
- Send commands
- Display information about endpoints

Refer to the `Simple Network Streams.lvproj` in the `labview\examples\Data Communication\Network Streams\Simple Network Streams` directory for an example of designing applications using network streams.



RELATED INFORMATION

[Streaming Data between Applications](#) on page 663

[Sending Commands between Applications](#) on page 665

[Displaying Information about Endpoints](#) on page 668

[Data Communication Methods in LabVIEW](#) on page 651

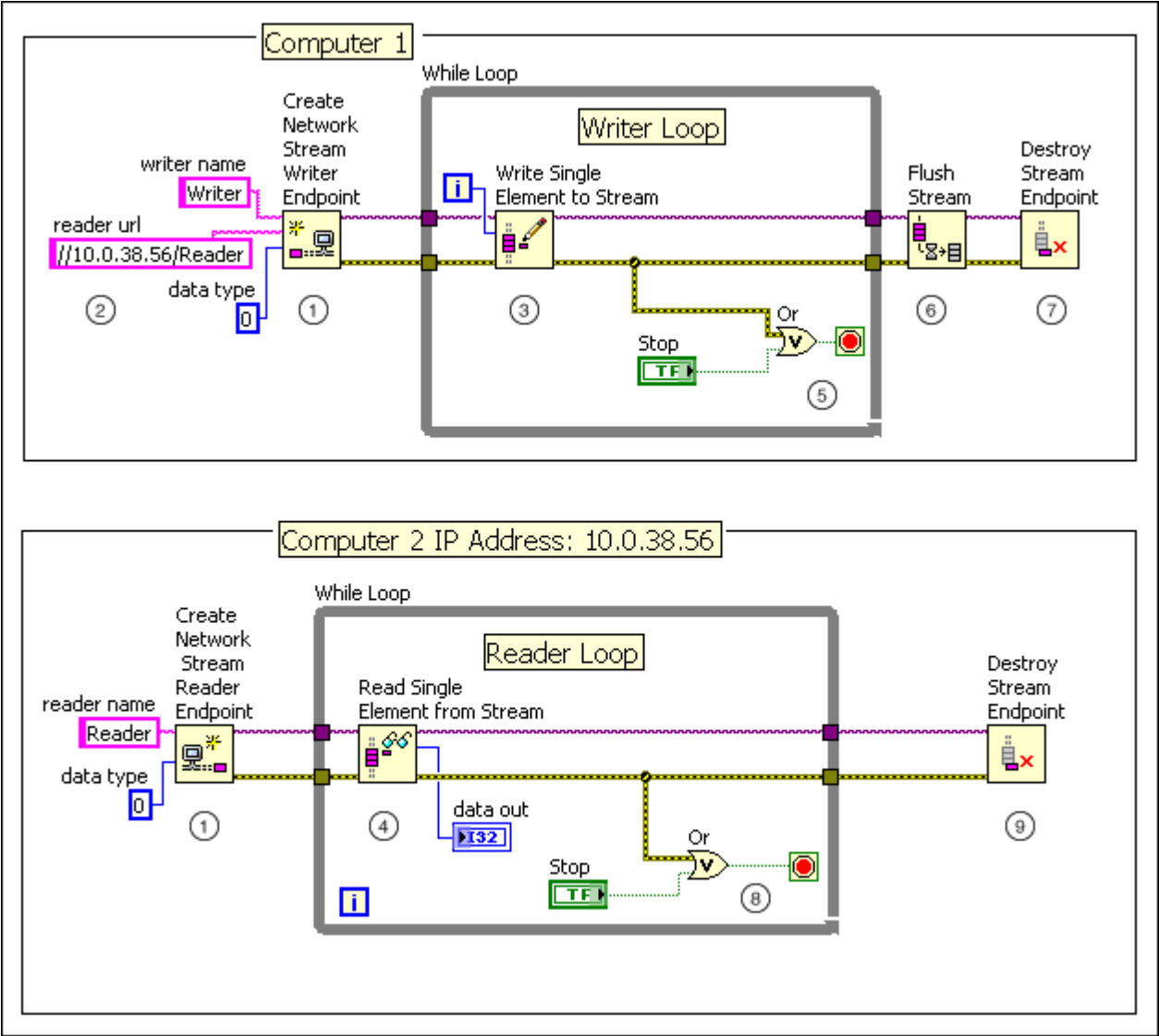
Streaming Data between Applications

The following figure shows an example of using the Network Streams functions to stream data between two applications on different computers.



NOTE

You can use network streams on Windows and Real-Time systems only.



The following events occur in the previous figure.

- 1

NOTE

You can run the Create Network Stream Writer Endpoint and Create Network Stream Reader Endpoint functions in any order. These functions will create a network stream unless the **timeout in ms** terminal on one function expires before the other function runs.
- 2

The writer endpoint establishes a connection with the reader endpoint using the **endpoint URL** of the reader endpoint.

Within the **Writer Loop**, the Write Single Element to Stream function continuously writes the value of the iteration **i** terminal of the While Loop to the stream.

③

**NOTE**

You also can use the Write Multiple Elements to Stream function to stream more than one data point at a time. For example, use this function if you need to stream two samples of waveform data every second.

Within the **Reader Loop**, the Read Single Element from Stream function continuously reads the stream.

④

**NOTE**

You also can use the Read Multiple Elements from Stream function to read a stream. You can use a Read Multiple Elements from Stream function even if you used a Write Single Element to Stream function in the other application, and vice versa.

⑤

Data streams continuously until the user clicks the **Stop** button on **Computer 1**.

⑥

The Flush Stream function transfers all remaining data to the reader endpoint.

⑦

The Destroy Stream Endpoint function destroys the writer endpoint.

⑧

The reader endpoint receives an error due to disconnection, and data flow exits the While Loop.

⑨

The Destroy Stream Endpoint function destroys the reader endpoint, which destroys the stream.

Sending Commands between Applications

Complete the following steps to send **commands** from one computer to another using the Network Streams functions.

**NOTE**

You can use network streams on Windows and Real-Time systems only.

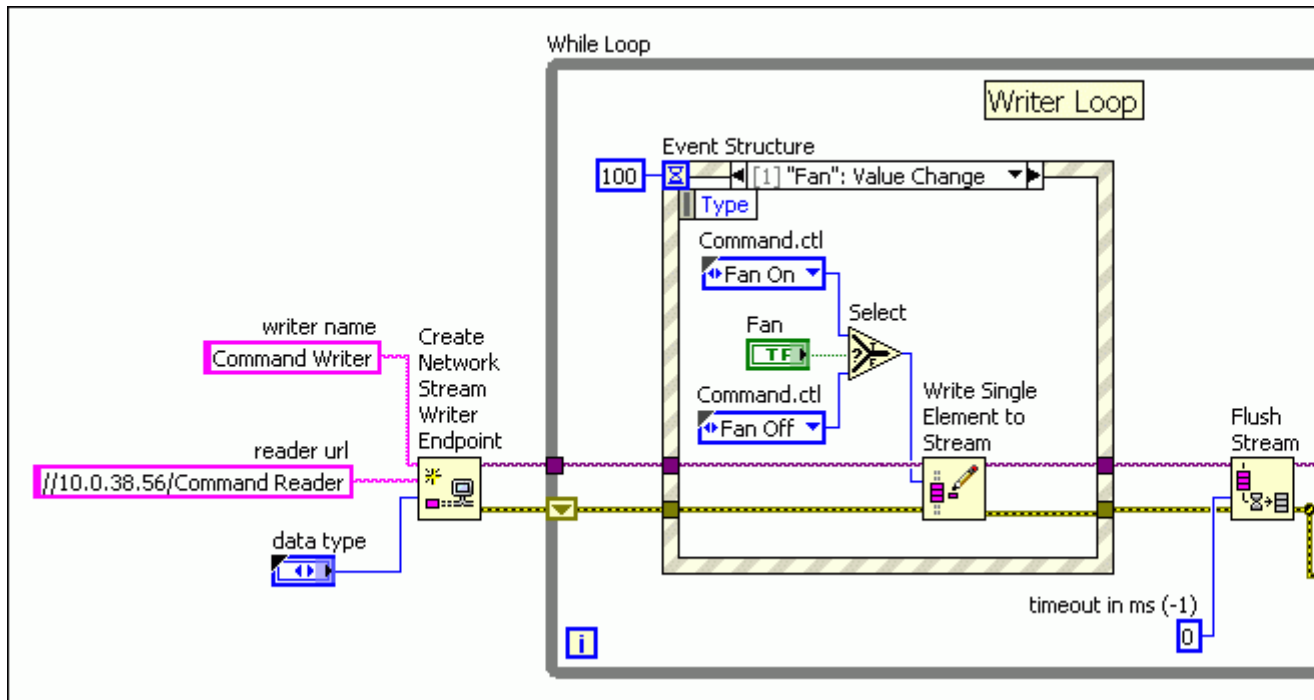
1. **Create a type definition** of an **enumerated type control** with an entry for each command. For example, the following figure shows an example of a type definition of an enumerated type control with commands to turn a fan on, turn a fan off, and stop the application that controls the fan.

Note Creating a type definition ensures that if you update one instance of the control, the other instances update as well.

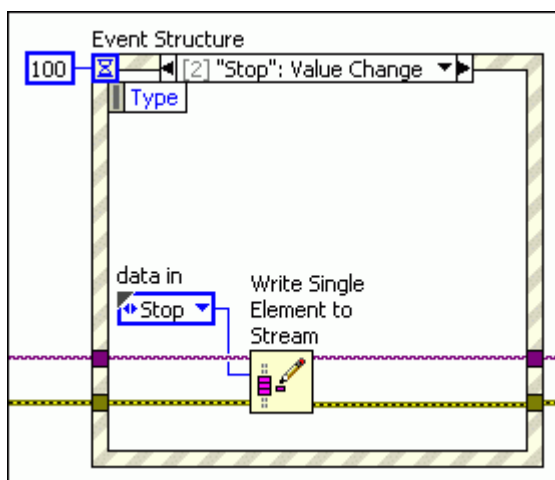


2. Create a **Writer Loop** and a **Reader Loop**, just as you do when *streaming data*.
3. Within the **Writer Loop**, *configure an Event structure* to send the commands based on user input. For example, in the following figure, the Write Single Element to Stream function sends the **Fan On** or **Fan Off** command when the user clicks the **Fan** button.

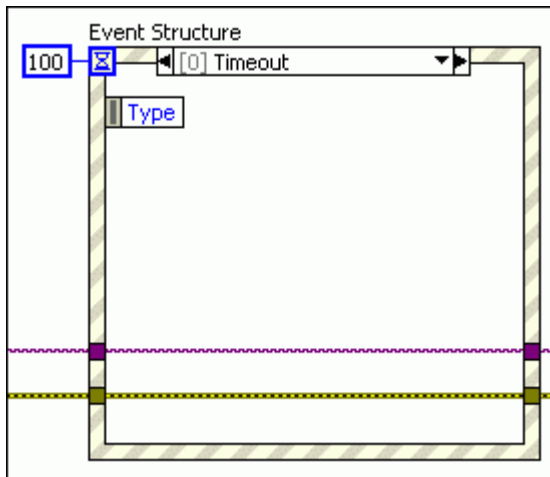
Note Use the Flush Stream function within the **Writer Loop** to minimize latency and ensure that commands propagate over the network immediately, as shown in the following figure.



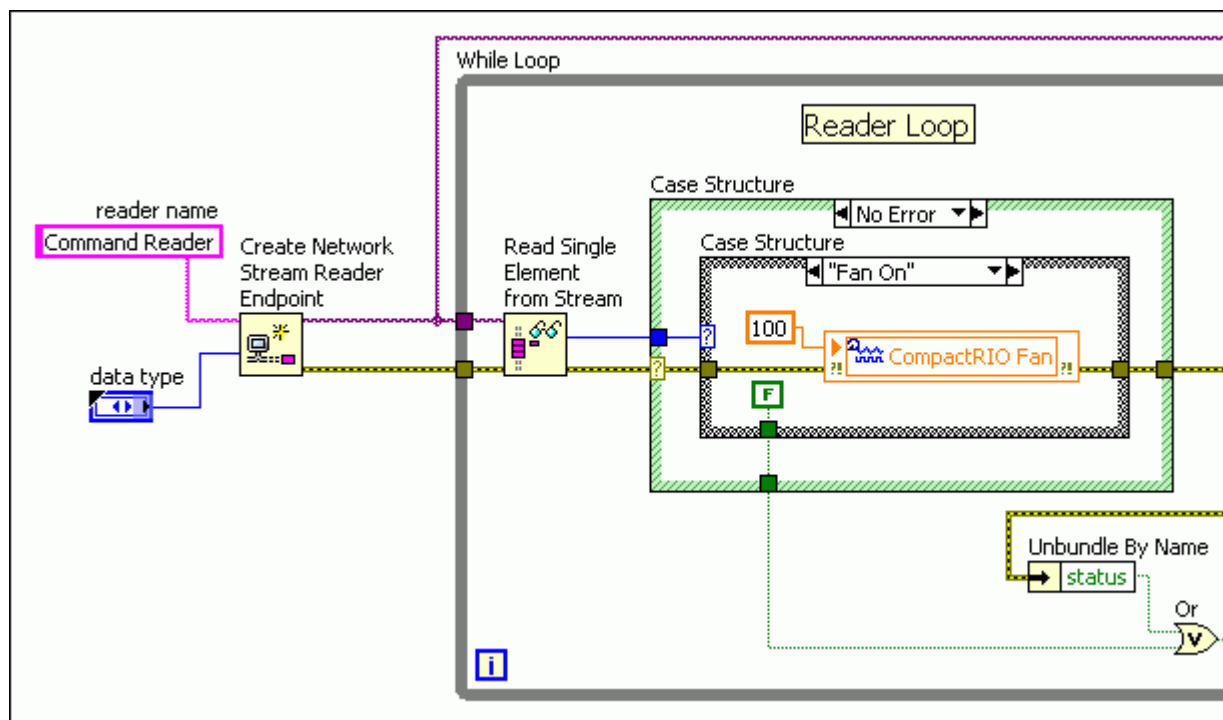
Similarly, in the following figure, the Write Single Element to Stream function sends the **Stop** command when the user clicks the **Stop** button.



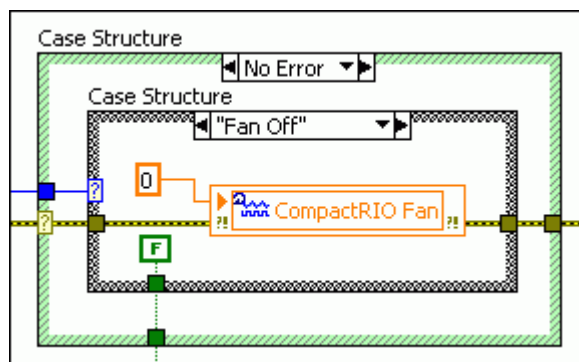
4. Configure the Event structure to do nothing if the Event structure times out, as shown in the following figure.



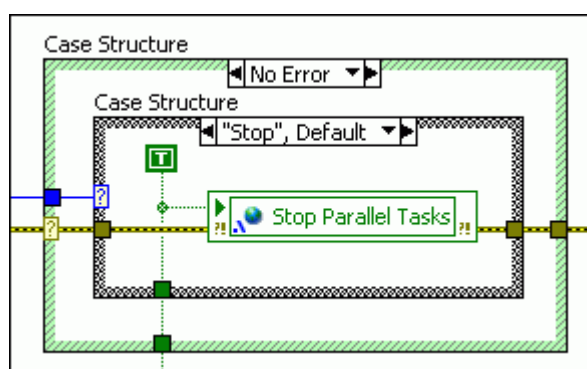
5. Within the **Reader Loop**, parse the commands and handle error conditions.
 1. Place a Case structure inside the **Reader Loop** and wire the **error out** terminal of the Read Single Element from Stream function to the case selector.
 2. Place a Case structure within the **No Error** case and wire the **data out** terminal of the Read Single Element from Stream function to the case selector. Use this Case structure to parse the commands you send from the **Writer Loop**. For example, in the following figure, when the Read Single Element from Stream function receives the **Fan On** command, the Case structure switches to the **Fan On** case and sets the **I/O variable** that controls the fan to an intensity of 100, which activates the fan.



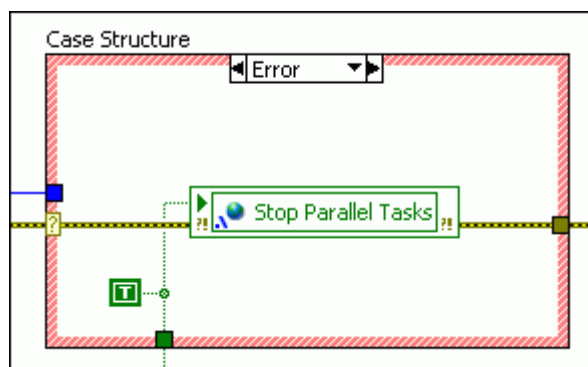
Similarly, in the following figure, when the Read Single Element from Stream function receives the **Fan Off** command, the Case structure switches to the **Fan Off** case and sets the I/O variable that controls the fan to an intensity of 0, which deactivates the fan.



Finally, in the following figure, when the Read Single Element from Stream function receives the **Stop** command, the Case structure switches to the **Stop** case, stops the **Reader Loop** with a TRUE constant, and uses a *single-process shared variable* to stop any parallel tasks within the VI.



3. Within the **Error** case, write code to stop the **Reader Loop** and any parallel tasks within the VI if an error occurs, as shown in the following figure.



Displaying Information about Endpoints

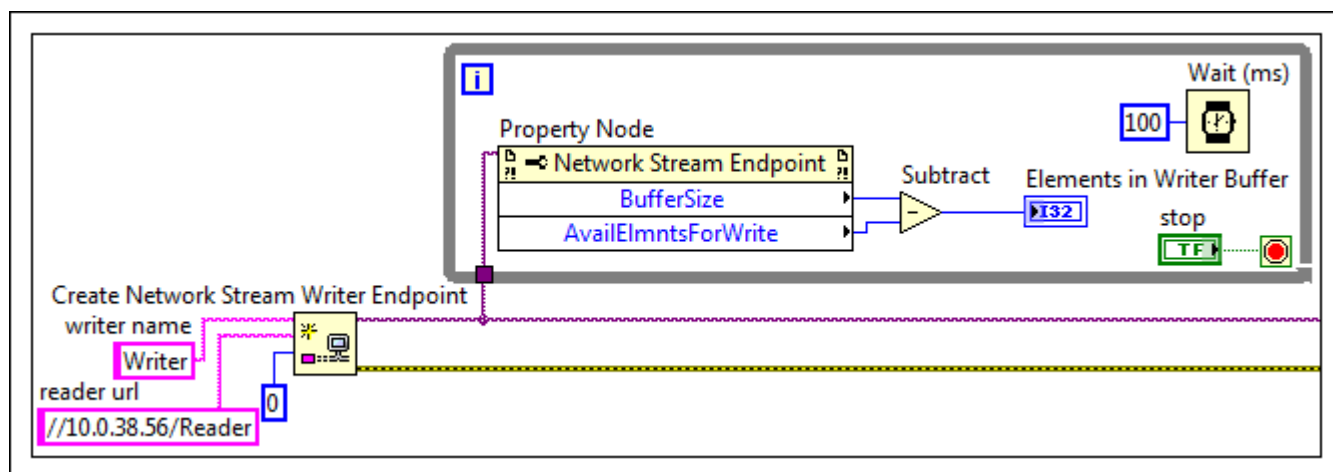
You can use the Network Stream Endpoint properties to display information about *endpoints*. Complete the following steps to use Network Stream Endpoint properties.

1. Add the Network Streams Property Node to the block diagram.
2. Wire an endpoint refnum wire to the **reference** input on the property node. Obtain this reference from the **reader endpoint** reference of the Create Network Stream Reader Endpoint function, the **writer endpoint** terminal of the Create Network

Stream Writer function, or the **endpoint out** terminal of any of the other Network Streams functions.

3. Right-click the **Property** terminal of the Network Stream Endpoint Property Node and select **Select Property** from the shortcut menu to select the property you want to use.
4. (Optional) Use the Positioning tool to *resize* the Network Stream Endpoint Property Node to display more than one property simultaneously.

The following figure shows an example of using the Network Stream Endpoint properties.



In this example, subtracting the Available Elements for Writing property from the Buffer Size property indicates the number of elements in the buffer of the writer endpoint.

Network Stream Endpoints

Endpoints are the parts of applications that exchange data. Every network stream must have a writer endpoint and a reader endpoint. Writer endpoints write data to the stream. Reader endpoints read data from the stream. LabVIEW identifies each endpoint with an endpoint URL, which you use to establish connections between endpoints. When two endpoints connect, they create a network stream. You use the Network Streams functions to create endpoints and stream data between them.



RELATED INFORMATION

[Specifying Network Stream Endpoint URLs](#) on page 670

[Connecting Network Stream Endpoints Together](#) on page 674

[Designing Applications Using Network Streams](#) on page 663

[Data Communication Methods in LabVIEW](#) on page 651

Specifying Network Stream Endpoint URLs

LabVIEW identifies every network stream endpoint with a URL. When you specify a name for an endpoint with the **writer name** terminal of the Create Network Stream Writer Endpoint function or the **reader name** terminal of the Create Network Stream Reader Endpoint function, LabVIEW uses that name to create a URL with the following syntax:

```
ni.dex:// host_name : context_name / endpoint_name
```

The following table describes each component of this URL.



NOTE

Only one application on each computer can specify the default context. Therefore, if you have *multiple applications on a single computer that use network streams*, you must assign a URL instead of a name to each endpoint in those applications.

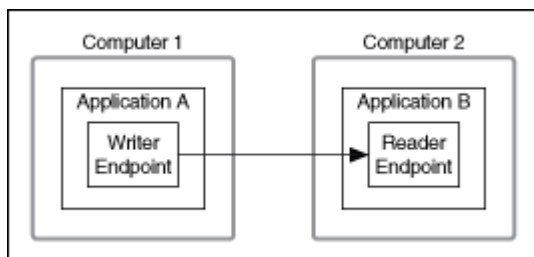
To create a valid network stream, use endpoint URLs to prompt a writer and a reader endpoint to connect to each other. Perform this task by wiring the URL of a remote endpoint to the **reader url** input on the Create Network Stream Writer Endpoint function or the **writer url** input on the Create Network Stream Reader Endpoint function.

The URL you must specify in these terminals varies depending on the network location of the remote endpoint. The following sections show examples of prompting endpoints to connect with each other.



Note Endpoint URLs are not case sensitive. However, when you specify an endpoint URL, you must replace any reserved characters that you use with the corresponding *escape codes* to prevent parsing errors.

Streaming to a Computer That Runs One Application Using Network Streams

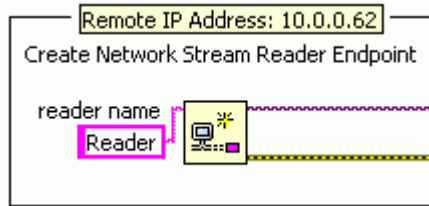


When you connect to an endpoint on a remote computer that runs one application that uses network streams, as shown in the figure above, you must use a URL with the following syntax to prompt a connection between the endpoints:

```
// host_name / endpoint_name
```

Complete the following steps to use a URL with this syntax to prompt a connection between two endpoints.

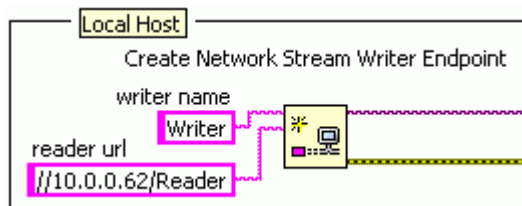
1. Create a reader endpoint on the remote computer with the Create Network Stream Reader Endpoint function, as shown in the following figure.



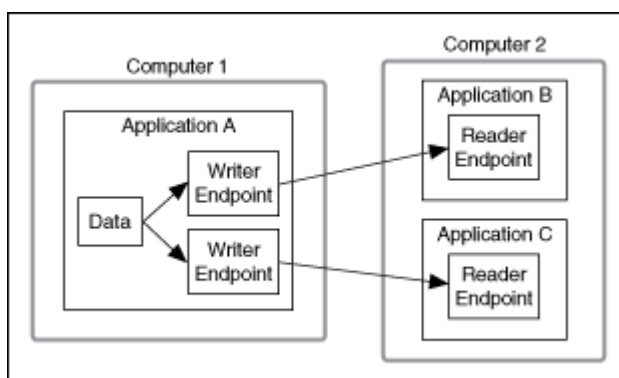
In the figure above, 10.0.0.62 is the host_name and Reader is the endpoint_name. Therefore, the URL of this endpoint is //10.0.0.62/Reader.

Note LabVIEW searches for a valid interpretation of the host name in the following order:

1. Target name (as specified in the LabVIEW project)
 2. DNS name
 3. IP address
2. Create a writer endpoint on the local host with the Create Network Stream Writer Endpoint function and specify the URL of the reader endpoint with the **reader url** terminal, as shown in the following figure.



Streaming to a Computer That Runs Multiple Applications Using Network Streams

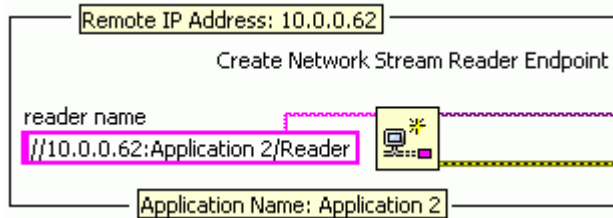


When you connect to an endpoint on a remote computer that runs multiple applications that use network streams, as shown in the figure above, you must use a URL with the following syntax to prompt a connection between the endpoints:

```
// host_name : context_name / endpoint_name
```

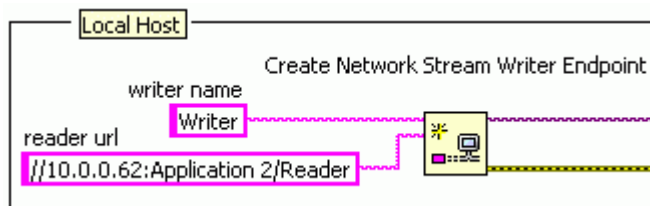
Complete the following steps to use a URL with this syntax to prompt a connection between two endpoints.

1. Create a reader endpoint on the remote computer with the Create Network Stream Reader Endpoint function, as shown in the following figure.

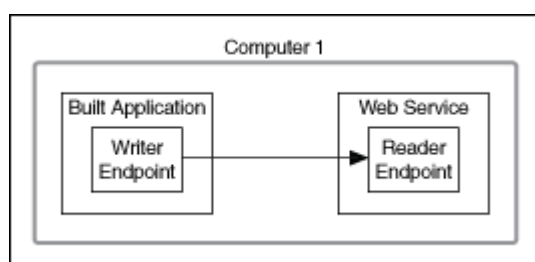


When you create an endpoint on a computer that runs multiple applications that use network streams, you must assign a URL that includes the `host_name`, a `context_name`, and an `endpoint_name`, as shown in the figure above. Assigning a URL instead of a name in this instance ensures that multiple applications on one computer do not use the same context, which would prevent two endpoints from connecting.

2. Create a writer endpoint on the local host with the Create Network Stream Writer Endpoint function and specify the URL of the reader endpoint with the **reader url** terminal, as shown in the following figure.



Streaming to an Application on the Local Host

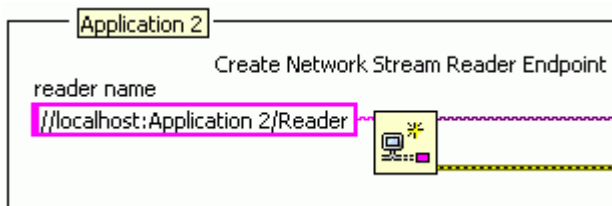


When you connect to an endpoint within another application on the local host, as shown in the figure above, you must use a URL with the following syntax to prompt a connection between the endpoints:

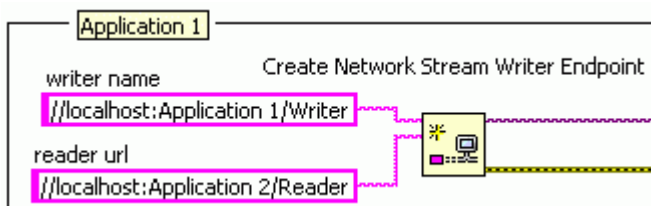
```
// localhost : context_name / endpoint_name
```

Complete the following steps to use a URL with this syntax to prompt a connection between two endpoints.

1. Create a reader endpoint within one application with the Create Network Stream Reader Endpoint function, as shown in the following figure.



2. Create a writer endpoint within the other application with the Create Network Stream Writer Endpoint function and specify the URL of the reader endpoint with the **reader url** terminal, as shown in the following figure.



Note Because the local host runs two applications that use network streams, you must specify a URL instead of a name for each endpoint.

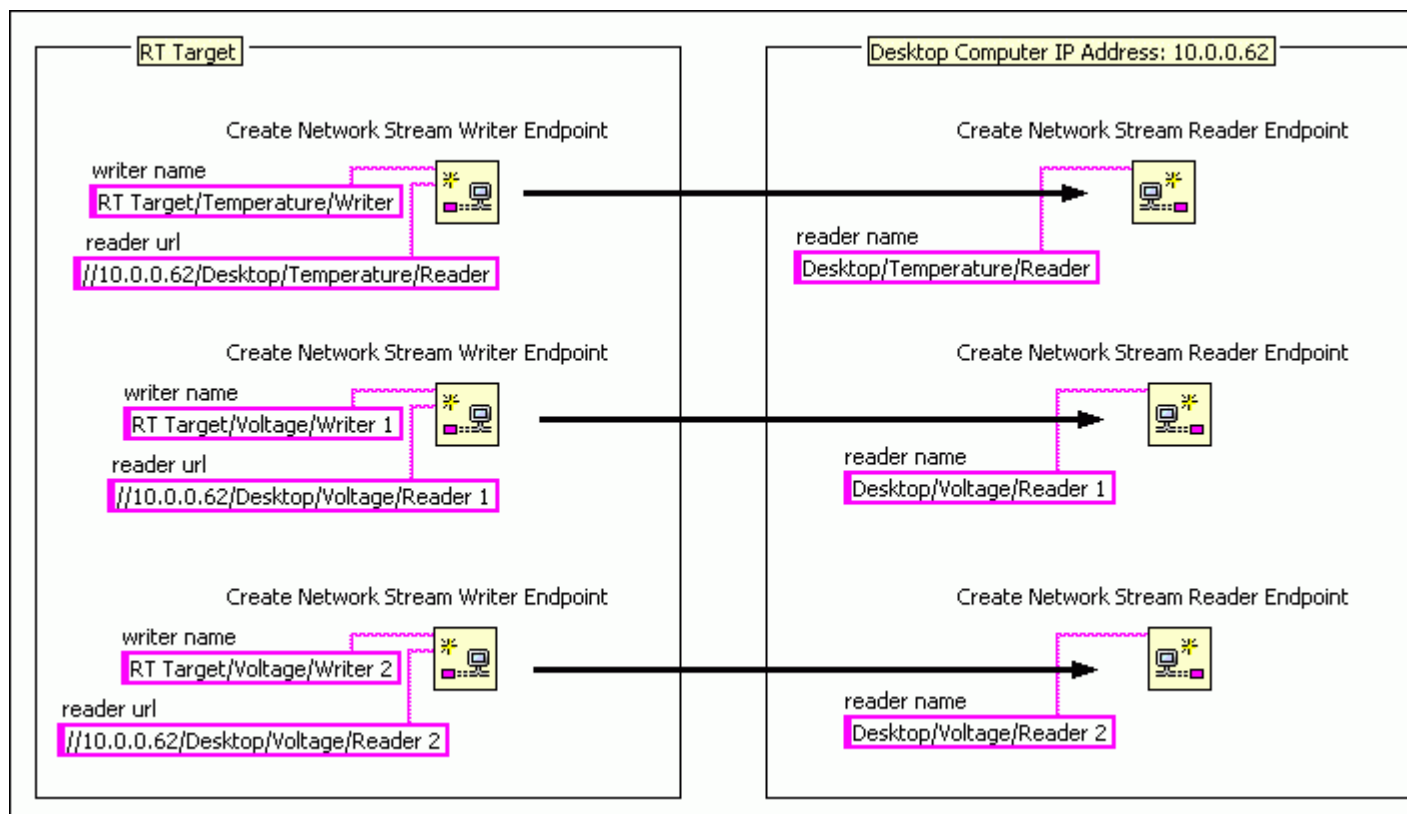
Organizing Network Stream Endpoint Names

You can organize endpoints by adding segments to their names that describe the data they stream, the computer on which they reside, or other characteristics. When you have multiple endpoints on different computers, organizing endpoints in this way helps ensure that you connect each writer endpoint to its corresponding reader endpoint.

Complete the following steps to organize endpoints by name.

1. Wire the **writer name** terminal of the Create Network Stream Writer Endpoint function or the **reader name** terminal of the Create Network Stream Reader Endpoint function.
2. Place a slash between each segment of the name that you specify in these terminals.

For example, assume you have three writer endpoints within the same application on a real-time (RT) target. One measures temperature, and the other two measure voltage signals. These endpoints connect to three corresponding reader endpoints in an application on a desktop computer. The following figure shows examples of names that you could assign to each endpoint. The arrows indicate the reader endpoint to which each writer endpoint connects.



In the figure above, the name of each endpoint describes the target the endpoint resides on, the type of data the endpoint streams, and whether the endpoint reads or writes that data. This naming scheme shows which endpoints correspond with each other.



NOTE

The name of one endpoint cannot be the partial name of another endpoint within the same application. For instance, in the above example, if you name one of the writer endpoints `RT Target/Voltage` and the other `RT Target/Voltage/Writer 1`, these endpoints will return an error instead of creating network streams.



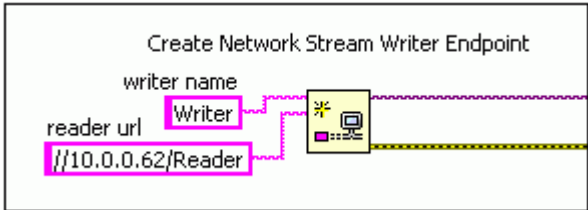
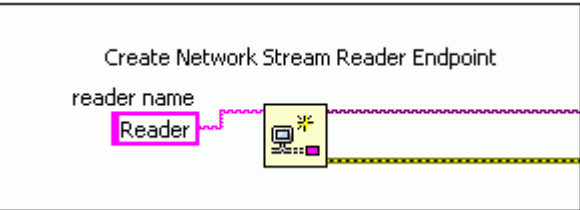
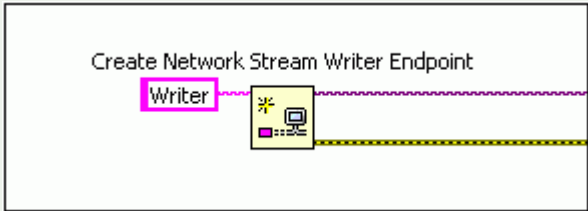
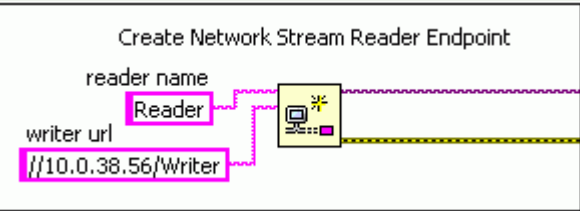
[Data Communication Methods Home](#)

Connecting Network Stream Endpoints Together

When a **Create Network Stream Writer Endpoint** function runs, it creates a writer endpoint and uses the **URL** you specify in the **reader url** input to connect to a reader endpoint that you create with the **Create Network Stream Reader Endpoint** function. This connection results in a valid network stream.

Wiring this input is just one way of creating a valid network stream. You also can leave the **reader url** input unwired. However, in this situation, you must wire the URL of a writer endpoint to the **writer url** input of the **Create Network Stream Reader Endpoint** function. Both of these methods are equivalent and result in a valid network stream by connecting the reader and writer endpoints together.

The following table shows examples of each of these methods.

Writer Endpoint	Reader Endpoint	Notes
Create Network Stream Writer Endpoint 	Create Network Stream Reader Endpoint 	If you configure the endpoints in this manner, the writer endpoint establishes a connection with the reader endpoint. Therefore, you can change the writer endpoint that connects to the reader endpoint.
Create Network Stream Writer Endpoint 	Create Network Stream Reader Endpoint 	If you configure the endpoints in this manner, the reader endpoint establishes a connection with the writer endpoint. Therefore, you can change the reader endpoint that connects to the writer endpoint.

Allocating Memory for Network Stream Endpoint Buffers

Network stream *endpoints* use a FIFO buffer to transfer data. The **data type** and **buffer size** inputs of the Create Network Stream Writer Endpoint and Create Network Stream Reader Endpoint functions determine how much memory the buffer requires. The **element allocation mode** input of these functions determines when initial buffer memory is allocated for non-scalar data types.

Regardless of the allocation mode you select, buffer size affects the amount of memory your application uses. Always *monitor memory usage* when testing your applications.

Specifying the Allocation Mode for Network Stream Endpoint Buffers

Use the **element allocation mode** input on the Create Network Stream Writer Endpoint and Create Network Stream Reader Endpoint functions to specify allocation mode.



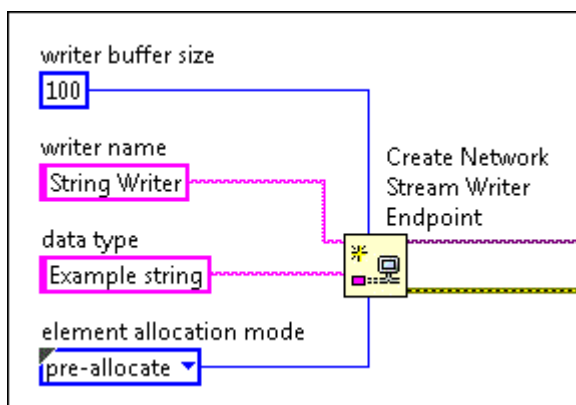
NOTE

The **element allocation mode** input does not affect the transfer of scalar data types. Scalar data transfers operate in **pre-allocate** mode.

Use **allocate as needed** mode if your application transfers non-scalar data and you cannot determine the maximum size of each data element before run-time. In **allocate as needed** mode, LabVIEW allocates only enough memory to create a pointer or handle for each data element when the network stream is created. LabVIEW then dynamically allocates additional memory for each element as it enters the buffer.

Use **pre-allocate** mode if your application transfers non-scalar data and you can determine the maximum size of each data element at edit-time. In **pre-allocate** mode, the Create Network Stream Writer Endpoint and Create Network Stream Reader Endpoint functions allocate initial buffer memory based on the **data type** input and **writer buffer size** or **reader buffer size** inputs.

To pre-allocate an endpoint buffer, you must wire an element of the maximum expected size to the **data type** input. The following figure shows a Create Network Stream Writer Endpoint function that allocates memory for 100 string elements when the endpoint is created, each with a maximum size equivalent to Example string.





Note Additional memory is allocated dynamically if an element requires more than the pre-allocated amount of memory at runtime.

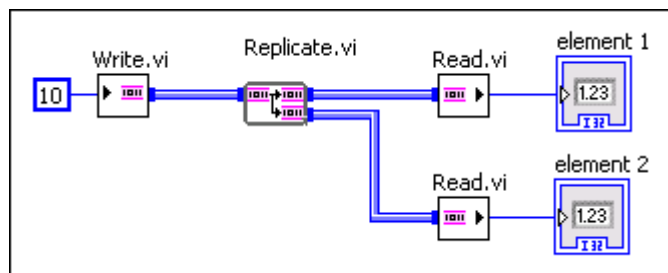


[Data Communication Methods Home](#)

Replicating Channels

Replicating a channel creates a copy of the data in a channel and splits the channel into two outgoing channels that connect to two different reader endpoints. LabVIEW represents the copy version of reader endpoint as a **Replicate** endpoint of the same channel template. A replicate endpoint has one channel input and two channel outputs. Each of the two channel outputs receives a separate copy of the data from the channel input, and each reads its own separate copy of data.

The following illustration demonstrates a replicate channel of the Stream template.



NOTE

To replicate a channel, right-click a channel wire and select **Insert»Replicate**. The **Insert»Replicate** option is only available for some of the template channels.

Type Propagation

Type propagation is an analysis step in the LabVIEW compilation process that resolves data types for terminals that adapt to different data types and detects syntax errors. This step runs each time you edit a VI.

For example, if you wire an integer to the input of the Decrement function, the data type of the output is an integer. If you wire a floating-point number to the input, the output is a floating-point number. The same pattern exists for arrays and clusters. This behavior of determining the output data type based on the input data type through the block diagram is called type propagation. If you wire an enum to the input of the Decrement function, the type fails to propagate through the block diagram and the VI reports a syntax error because decrementing an enum is not a legal operation for this function.

Type Descriptors

LabVIEW associates each wire and terminal on the block diagram with a data type. LabVIEW keeps track of this type with a structure in memory called a type descriptor. A type descriptor is a sequence of 32-bit integers that can describe any data type in LabVIEW. Numeric values are written in hexadecimal format, unless otherwise noted.

A type can be aggregate, meaning it can consist of different types. For example, in an array of Booleans, the array is a type, and its Boolean elements are another type. In LabVIEW, all type descriptors for such complex types are stored in a list.

Observable Formats

You can observe type descriptors in different formats using the Flatten To String function or the Variant To Flattened String function.



NOTE

The format in which LabVIEW stores type descriptors changed in LabVIEW 8.0. LabVIEW 7. x and earlier store type descriptors in 16-bit flat representation. LabVIEW 8.0 and later store type descriptors in 32-bit flat representation. The information for LabVIEW 7. x and earlier applies if you use the **Convert 7.x Data** mode of the Flatten To String function. The following information applies to LabVIEW 8.0 or later. NI recommends that you use the *variant* data type to avoid any conflicts these format changes might cause.

When you use the Flatten To String function to flatten variant data, LabVIEW flattens the variant and all its contents, including attributes. Using this function, the type descriptor save buffer has the following format:

32 bits	nTDs TDs
var size	nTypesUsed
var size	TypesUsed

where nTDs	is the number of elemental type descriptors contained in the buffer
TDs	is the list of nTDs type descriptors
var size	is the variable size and stored in a variable number, either 16-bits or 32-bits, depending on the value being represented. If the value is less than 32768, it is stored in two bytes. If the value is greater than or equal to 32768, it is stored in four bytes with the high bit set to one.
nTypesUsed	is the actual number of type descriptors types being used by the entity referencing the buffer
TypesUsed	is the list of indexes for all the types of TDs being used

When you use the Variant To Flattened String function, LabVIEW flattens only the variant and discards any attributes. Using this function, a type descriptor has the following format:

[size][type code][type specific info][name (if applicable)]

where [size]	is the first word, in bytes, and includes the size word. The minimum value of [size] is 4. You can append a name (a Pascal string) to any type descriptor, which increases the value of [size] by the length of the name rounded up to a multiple of two. The size is 16 bits.
[type code]	is the second word. LabVIEW reserves the high-order byte of the type code for internal use. When comparing two type descriptors for equality, you should ignore this byte. Two type descriptors are equal even if the high-order bytes of the type codes are not. The type code is 16 bits.
[type specific info]	is additional information about the type descriptor. Arrays and clusters are structured or aggregate data types because they include references to other types. For example, the cluster type contains additional information about the type of each of its elements.
[name]	is a Pascal string. Name may or may not be present.

The type code encodes the actual type information, such as single-precision or extended-precision, floating-point number, as listed in the following table. These type code values might change in future versions of LabVIEW.

Data Types

The following table lists numeric and non-numeric data types, type codes, and type descriptors. The minimum value of [size] appears before the type code. Information on the possible values of [type specific info] appears after the type code for some data types.



Note Name may be appended to any type descriptor. Name is a Pascal string.

Data Type	Type Descriptor (numbers in hexadecimal)
8-bit Integer	0004 xx 01
16-bit Integer	0004 xx 02
32-bit Integer	0004 xx 03
64-bit Integer	0004 xx 04
Unsigned 8-bit Integer	0004 xx 05
Unsigned 16-bit Integer	0004 xx 06

Data Type	Type Descriptor (numbers in hexadecimal)
Unsigned 32-bit Integer	0004 xx 07
Unsigned 64-bit Integer	0004 xx 08
Single-Precision, Floating-Point Number	0004 xx 09
Double-Precision, Floating-Point Number	0004 xx 0A
Extended-Precision, Floating-Point Number	0004 xx 0B
Complex Single-Precision, Floating-Point Number	0004 xx 0C
Complex Double-Precision, Floating-Point Number	0004 xx 0D
Complex Extended-Precision, Floating-Point Number	0004 xx 0E
Enumerated 8-bit Integer	<nn> xx 15 <k><k pstrs>
Enumerated 16-bit Integer	<nn> xx 16 <k><k pstrs>
Enumerated 32-bit Integer	<nn> xx 17 <k><k pstrs>
Single-Precision Physical Quantity	<nn> xx 19 <k><k base-exp>
Double-Precision Physical Quantity	<nn> xx 1A <k><k base-exp>
Extended-Precision Physical Quantity	<nn> xx 1B <k><k base-exp>
Complex Single-Precision Physical Quantity	<nn> xx 1C <k><k base-exp>
Complex Double-Precision Physical Quantity	<nn> xx 1D <k><k base-exp>
Complex Extended-Precision Physical Quantity	<nn> xx 1E <k><k base-exp>
Boolean	0004 xx 21
String	0008 xx 30 <dim>
Path	0008 xx 32 <dim>
Picture	0008 xx 33 <dim>
Array	<nn> xx 40 <k dims><element type descriptor>
Cluster	<nn> xx 50 <k elems><element type descriptors>

Data Type	Type Descriptor (numbers in hexadecimal)
Waveform	<nn> xx 54 <waveform type><element type descriptors>
Refnum	<nn> <refnum type code>
Variant	<nn> xx 53
nn = length; xx = reserved by LabVIEW; k = number; k pstrs = number of Pascal strings; k base-exp = number of base-exponent pairs ; k dims = number of dimensions; k elems = number of elements; dim is a 32-bit integer.	

The minimum value in the size field of a type descriptor is 4, as shown in the previous table. However, any type descriptor can have a name (a Pascal string) appended, in which case the size field is larger by the length of the name (as a Pascal string) rounded up to a multiple of two.

Enumerated 8-Bit Integer

In the following example of an enumerated 8-bit integer without a name for the items `am`, `fm`, and `fm stereo`, each group of characters represents a 16-bit word expressed in hexadecimal notation.

```
0016 0015 0003 0261 6D02 666D 0966 6D20 7374 6572 656F
```

The type descriptor has the following words:

- 0016—indicates 22 bytes total
- 0015—indicates the type code for an enumerated 8-bit integer
- 0003—indicates there are three items
- 0261 6D02 666D 0966 6D20 7374 6572 656F—indicates the type code specific information. For example, 0261 6D is the ASCII encoding for `am` with the 02 prefix indicating the size of the string in hexadecimal.

If the same enumerated 8-bit integer was named `radio`, its type descriptor can be interpreted as:

```
001C 4015 0003 0261 6D02 666D 0966 6D20 7374 6572 656F 0572 6164 696F
```

Below is a comparison of the two descriptors:

Physical Quantity

```
000E 001A 0002 0002 FFFF 0003 0001
```

- 000E—indicates 14 bytes total
- 001A—indicates the type code for a double-precision physical quantity
- 0002—indicates two base-exponent pairs
- 0002—denotes the seconds base index
- FFFF (-1)—indicates the exponent of seconds
- 0003—denotes the meters base index
- 0001—indicates the exponent of meters

The following table shows the nine bases LabVIEW uses and their representation by indexes 0 through 8.

om

Quantity Name	Unit	Abbreviation	Base Value
solid angle	steradian	sr	1
time	second	s	2
length	meter	m	3
mass	kilogram	kg	4
electric current	ampere	A	5
thermodynamic temperature	kelvin	K	6
amount of substance	mole	mol	7
luminous intensity	candela	cd	8

String, Path, and Picture Data Types

The string, path, and picture data types have a 32-bit length, similar to the array dimension size. However, the only value currently encoded is hexadecimal FFFFFFFF, which indicates variable sized. Currently, all strings, paths, and pictures are variable size. The actual length is stored with the data.

Array and Cluster Data Types

The array and cluster data types each have their own type code. They also contain additional information about the data types of their elements and the dimensionality for arrays or number of elements for clusters.

Array

The type code for an array is 40. The type descriptor for an array has the following format:

`<nn> xx 40 <k dims><element type descriptor>`

Consider the following example of a type descriptor for a 1D array of double-precision, floating-point numbers, where each group of characters represents a 16-bit word expressed in hexadecimal notation:

`000E 0040 0001 FFFF FFFF 0004 000A [type specific info]`

The type descriptor has the following words:

- 000E—indicates 12 bytes total, which is the length of the entire type descriptor including the element type index
- 0040—indicates the type code for an array

- 0001—indicates one dimension, the number of dimensions in the array
- FFFF FFFF—indicates the size of the dimension, which is variable-sized. LabVIEW stores the actual dimension size, which is always greater than or equal to zero, with the data.
- 0004 000A—indicates the type descriptor for the element. The element type can be any type except an array. In this example, the elements are double-precision, floating-point numbers and have the following words:
 - 0004—indicates 4 bytes, which is the size of the element
 - 000A—indicates the type code for a double-precision, floating point number

The following is the format of the type descriptor save buffer for this example:

Number of type descriptors	0000 0002	The number of unique type descriptors is 2.
Type descriptor 0	0004 000A	The first type descriptor is for the double-precision, floating-point number.
Type descriptor 1	000E 0040 0001 FFFF FFFF	The second type descriptor is for the array. 000E is the length of the entire type descriptor, including the element type index. The array is variable sized, so the dimension is FFFFFFFF.
Number of types used	0002	The total number of types used is 2.
Types used	0000 0001	The types used are 0 and 1.

Consider the following example of a type descriptor for a 2D array of Boolean values:

0012 0040 0002 FFFF FFFF FFFF FFFF 0004 0021 [type specific info]

The type descriptor has the following words:

- 0012—indicates 18 bytes total, which is the length of the entire type descriptor including the element type index
- 0040—indicates the type code for an array
- 0002—indicates two dimensions, the number of dimensions in the array
- FFFF FFFF (-1)—indicates the size of the first dimension, which is variable-sized
- FFFF FFFF (-1)—indicates the size of the second dimension, which is variable-sized
- 0004 0021—indicates the type descriptor for the element. In this example, the elements are Boolean and have the following words:
 - 0004—indicates 4 bytes, which is the size of the element
 - 0021—indicates the type code for a Boolean

The following is the format of the type descriptor save buffer for this example:

Number of type descriptors	0000 0002	The number of unique type descriptors is 2.
Type descriptor 0	0004 0021	The first type descriptor is for the Boolean.
Type descriptor 1	0012 0040 0002 FFFF FFFF FFFF FFFF	The second type descriptor is for the array. 0012 is the length of the entire type descriptor, including the element type index. The array is variable sized, so the dimension is FFFFFFFF.
Number of types used	0002	The total number of types used is 2.
Types used	0000 0001	The types used are 0 and 1.

Cluster

The type code for a cluster is 50. The type descriptor for a cluster has the following format:

<nn> xx 50 <k elems><element type descriptors>

Consider the following example of a type descriptor for a cluster containing two integers, a signed 16-bit integer and an unsigned 32-bit integer, where each group of characters represents a 16-bit word expressed in hexadecimal notation:

000E 0050 0002 0004 0002 0004 0007

The type descriptor has the following words:

- 000E—indicates 14 bytes total
- 0050—indicates the type code for a cluster
- 0002—indicates two elements, the number of elements in the cluster
- 0004 0002—indicates the type descriptor of the first element, a signed 16-bit integer:
 - 0004—indicates 4 bytes
 - 0002—indicates the type code for a signed 16-bit integer
- 0004 0007—indicates the type descriptor of the second element, an unsigned 32-bit integer:
 - 0004—indicates 4 bytes
 - 0007—indicates the type code for an unsigned 32-bit integer

The following is the format of the type descriptor save buffer format:

Number of type descriptors	0000 0003	The number of unique type descriptors is 3.

Type descriptor 0	0004 0002	The first type descriptor is for the signed 16-bit integer.
Type descriptor 1	0004 0007	The second type descriptor is for the unsigned 32-bit integer.
Type descriptor 2	000E 0050 0002	The third type descriptor is for the cluster. 000E is the length of the entire type descriptor, including the cluster type index.
Number of types used	0003	The total number of types used is 3.
Types used	0000 0001 0002	The types used are 0, 1, and 2.

Because array and cluster type descriptors contain other type descriptors, they might become deeply nested.

Note that duplicate types are not included in the type list. The following example is a type descriptor list of a 1D array of double-precision, floating-point numbers and a cluster with one double-precision, floating-point number.

Number of type descriptors	0000 0003	The number of unique type descriptors is 3.
Type descriptor 0	0005 000A 00	The first type descriptor is for the element type and appears exactly as it does for a scalar of the same type.
Type descriptor 1	000C 0040 0001 FFFF FFFF 0000	The second type descriptor is for the array. 000C is the length of the entire type descriptor, including the element type index. The array is variable sized, so the dimension is FFFFFFFF.
Type descriptor 2	0008 0050 0001 0002	The third type descriptor is for the cluster. 0008 is the length of the entire type descriptor, including the cluster type index. The cluster contains one element by the type at index 2.
Number of types used	0003	The total number of types used is 4.
Types used	0000 0001 0000 0002	The types used are 0, 1, 0, and 2.

Note that the type referenced in the cluster at index 2 must be looked up in the types used array to make sense. The second element in types used is 0, which corresponds to type descriptor 0.

Waveform

The type code for a waveform is 54. A word that contains the type of waveform immediately follows the type code. The type descriptor of the cluster of the waveforms elements follows this word. There are five categories of waveforms:

- Analog
- Digital
- Digital Data
- Time Stamp
- Dynamic

Analog Waveform

The following table lists the types of analog waveforms.

Type	Subtype Code
8-bit signed	14
16-bit signed	2
32-bit signed	15
64-bit signed	19
Double-Precision	3
Single-Precision	5
Extended-Precision	10
8-bit unsigned	11
16-bit unsigned	12
32-bit unsigned	13
64-bit unsigned	20
Complex Single-Precision	16
Complex Double-Precision	17
Complex Extended-Precision	18

For example, consider a double-precision waveform, where each group of characters represents a 16-bit word expressed in hexadecimal notation:

```
0006 0054 0003 //size of waveform, waveform type code, and double-precision subtype code 3
```

0080 0050 0005 //size of cluster, cluster code, and number of elements

<type descriptor of time stamp>

<type descriptor of dt >

<type descriptor of array of element type>

<type descriptor of unused error cluster>

<type descriptor of attributes>

The type descriptor of the other types of analog waveforms is similar to the double-precision waveform. The difference is a different type for the array element and a different subtype.

Digital Waveform and Digital Data

The digital waveform is the only type of waveform that has a subtype code 8.

For example, consider the following, where each group of characters represents a 16-bit word expressed in hexadecimal notation:

00B6 0054 0008 //size of digital waveform, waveform type code, and subtype code 8

<type descriptor of time stamp>

<type descriptor of dt >

<type descriptor of digital data>

<type descriptor of unused error cluster>

<type descriptor of attributes>

Digital Data

Digital data is the only data type with a waveform type code and a subtype code of 7.

For example, consider the following where each group of characters represents a 16-bit word expressed in hexadecimal notation:

003E 0054 0007 //size of digital data, waveform type code, and subtype code 7

<type descriptor of transitions>

<type descriptor of data>

Time Stamp

The time stamp has a subtype code of 6. The time stamp is a 128-bit fixed-point number with a 64-bit radix. LabVIEW stores a time stamp as a cluster of four integers where the first two signed integers (64 bits) represent the number of whole seconds after 12:00 a.m., Friday, January 1, 1904, Universal Time [01-01-1904 00:00:00]. The next two unsigned integers (64 bits) represent the fractions of seconds.

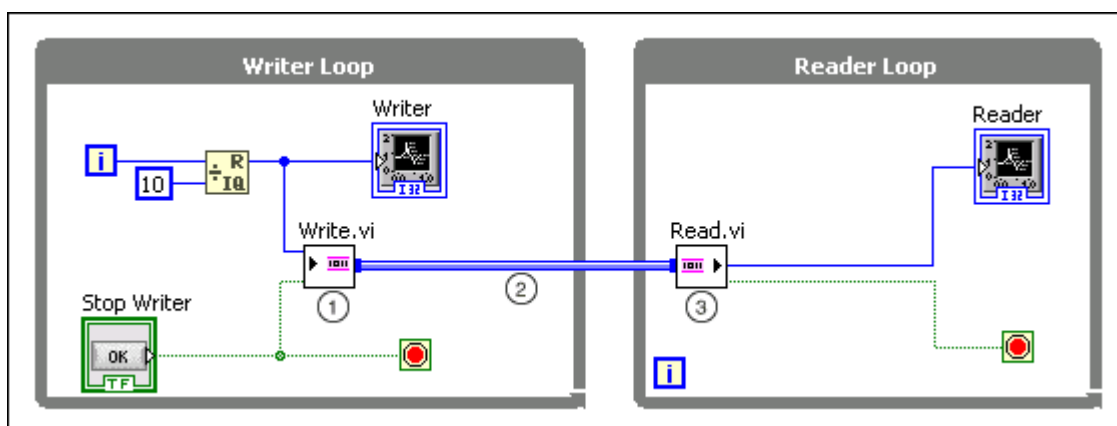
```
001C 0054 0006 //size of time stamp, waveform type code, and time
stamp subtype code 6
```

<type descriptor of an analog waveform>

Transferring Data between Two Loops Running in Parallel

Using a channel wire is similar to using a refnum or a variable that one loop writes into and the other loop reads from. Instead of using a refnum or a variable, you represent data transfer graphically as a wire. Channel wires are asynchronous wires that connect two parallel sections of code without forcing an execution order. Thus, they avoid creating a data dependency between the two sections of code.

The following illustration demonstrates transferring data between two loops running in parallel using channel wires. Within **Writer Loop**, the **Write** endpoint continuously writes the value of the iteration terminal to the channel. Within **Reader Loop**, the **Read** endpoint continuously reads data from the channel. The **Reader Loop** does not need to wait for the **Writer Loop** to complete before the **Reader Loop** starts running. Instead, the two loops run in parallel. When the user clicks the **Stop Writer** button in **Writer Loop**, the **Write** endpoint stops writing data to the channel immediately, and the **Read** endpoint stops reading after it reads all the remaining data in the channel.



The following list describes important details about the previous diagram:



NOTE

If you see a coercion dot or if the wire is broken because of different data types in the channel, right-click the endpoint or front panel terminal and select **Adapt To Element Type** or **Adapt To Channel Type** from the shortcut menu to resolve data type issues.

Caveats and Recommendations

The following list describes some of the caveats and recommendations to consider when you debug channel wires:

- When probing a channel wire, the **Debug** window only serves as a flag that you want to probe that channel. A separate window for channel probes appears when the VI starts running. You can view the ongoing state of the channel in the channel probe window as different endpoints read or write data.
- Selecting the **Pause channel when wire probe activates for a new channel** option in the **Debug** window causes the channel probe to pause as soon as the channel probe window opens. If you select the option early during the execution of your overall program, you can pause the channel itself before any endpoint executes. Pausing the channel itself allows you to watch closely what is happening in your channel.
- The channel probe window provides debugging capabilities such as pausing, conditional pausing, and single-stepping. The debugging options are customized for each channel template, allowing for fine-grain debugging of channels.
- Debugging capabilities are not available for High Speed Stream channels. You cannot probe the High Speed Stream channel even when debugging is enabled in a caller VI. Such settings ensure that you can transfer data in a fast and highly efficient manner using High Speed Stream channels. This should not deter you from using the other template channels because debugging has marginal impact on the performance of channels and the vast majority of caller VIs.



RELATED INFORMATION

[Debug Window - LabVIEW Programming Reference Manual](#)

Formulas and Equations in LabVIEW

Formulas and Equations

When you want to use a complicated equation in LabVIEW, you do not have to wire together various arithmetic functions on the block diagram. You can develop equations in

a familiar, mathematical environment and then integrate the equations into an application.



RELATED INFORMATION

[Formula Nodes](#) on page 691

[Expression Nodes](#) on page 692

[Formula Node and Expression Node Functions](#) on page 1118

[Precedence of Operators in Formula Nodes and Expression Nodes](#) on page 1120

[Script Nodes](#) on page 1131

Formula Nodes

The Formula Node is a convenient text-based node you can use to perform mathematical operations on the block diagram. You do not have to access any external code or applications, and you do not have to wire low-level arithmetic functions to create equations. In addition to text-based equation expressions, the Formula Node can accept text-based versions of if statements, while loops, for loops, and do loops, which are familiar to C programmers. These programming elements are similar to what you find in C programming but are not identical.



NOTE

You can use Formula Node most effectively if you are familiar with the C programming language. LabVIEW Help assumes you are familiar with C and does not contain any topics related to programming in C. Refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie or *Teach Yourself C* by Herbert Schildt for more information.

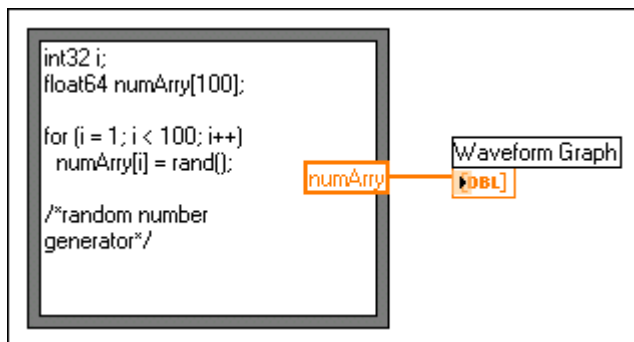
Formula Nodes are useful for equations that have many variables or are otherwise complicated and for using existing text-based code. You can copy and paste the existing text-based code into a Formula Node rather than recreating it graphically.

Formula Nodes use type checking to make sure that array indexes are numeric data and that operands to the bit operations are integer data. Formula Nodes also check to make sure array indexes are in range. For arrays, an out-of-range value defaults to zero, and an out-of-range assignment defaults to nop to indicate no operation occurs.

Formula Nodes also perform automatic type conversion.

Using the Formula Node

The Formula Node, shown as follows, is a resizable box similar to the For Loop, While Loop, Case structure, Stacked Sequence structure, and Flat Sequence structure. However, instead of containing a subdiagram, the Formula Node contains one or more C-like statements delimited by semicolons, as in the following example. As with C, add comments by enclosing them inside a slash/asterisk pair `/*comment*/` or by preceding them with two slashes `//comment`.



Variables in the Formula Node

When you work with variables, remember the following points:

- There is no limit to the number of variables or equations in a Formula Node.
- No two inputs and no two outputs can have the same name, but an output can have the same name as an input.
- Declare an input variable by right-clicking the Formula Node border and selecting **Add Input** from the shortcut menu. You cannot declare input variables inside the Formula Node.
- Declare an output variable by right-clicking the Formula Node border and selecting **Add Output** from the shortcut menu. The output variable name must match either an input variable name or the name of a variable you declare inside the Formula Node.
- You can change whether a variable is an input or an output by right-clicking it and selecting **Change to Input** or **Change to Output** from the shortcut menu.
- You can declare and use a variable inside the Formula Node without relating it to an input or output wire.
- You must wire all input terminals.
- Variables can be floating-point numeric scalars, whose precision depends on the configuration of your computer. You also can use integers and arrays of numeric values for variables.



Note The Formula Node does not support complex data types or the matrix data type. If you want to use these data types, use a script node.

- Variables cannot have units.

Expression Nodes

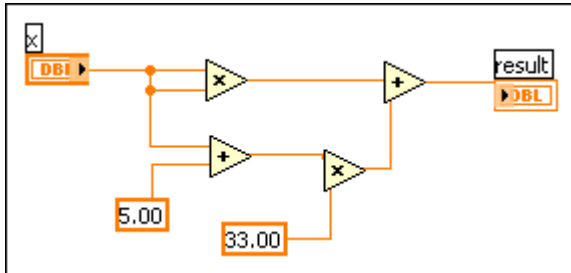
Use the Expression Node to calculate expressions that contain a single variable. Expression Nodes are useful when an expression has only one variable, but is otherwise complicated.

Expression Nodes use the value you pass to the input terminal as the value of the variable. The output terminal returns the value of the calculation.

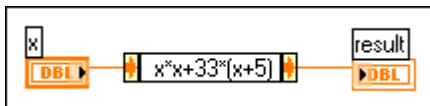
For example, consider this simple expression:

$$x \times x + 33 \times x + 5)$$

The following block diagram uses Numeric functions to represent this expression.



Use an Expression Node, shown as follows, to create a much simpler block diagram.



Polymorphism in Expression Nodes

The input terminal of an Expression Node is the same data type as the control or constant you wire to it. The output terminal is the same data type as the input terminal. The data type of the input can be any non-complex scalar number, array of non-complex scalar numbers, or cluster of non-complex scalar numbers. With arrays and clusters, the expression node applies the expression to each element of an input array or cluster.

Feedback Node

Use a Feedback Node to store data from previous block diagram executions or loop iterations. The Feedback Node is useful in storing and retrieving state information from the previous execution.

A Feedback Node does not perform any action on the data it receives. Instead, it receives a value from the *initializer terminal* and transfers the value to the next input terminal. When a Feedback Node receives a new value, the node stores the value until the node passes that value to the next input terminal. A Feedback Node is analogous to a z^{-1} block in feedback control theory and digital signal processing.

Creating Feedback Nodes

A Feedback Node, shown as follows, automatically appears if you wire the output of a subVI, function, or group of subVIs and functions to the input of that same VI, function, or group and if you enable **Auto-insert Feedback Node in cycles** on the Block Diagram page in the Options dialog box. LabVIEW enables **Auto-insert Feedback Node in cycles** by default.



You also can manually select the Feedback Node on the Functions palette.

Refer to the Feedback Node - Building Array VI in the Structures \ Feedback Node directory for examples of the Feedback Node.

Initializer Terminals

The initializer terminal sets the initial value of the Feedback Node. The appearance of the initializer terminal, shown as follows, changes depending on how you configure the Feedback Node.



When you place a Feedback Node on the block diagram, the initializer terminal remains attached to the node and the node globally initializes each time the VI compiles or loads. The initial input of the Feedback Node for the first execution is the *default value for the data type*. However, you can *set an initial value* for the Feedback Node. If you set an initial value, the Feedback Node initializes to that value on the first call of the VI in an execution. Each time the VI runs after the first execution, the initial value is the last value from the previous execution. If you *use the Feedback Node in a loop*, you can move the initializer terminal to the left edge of the loop to configure the node to initialize every time the loop executes.



NOTE

(FPGA Module) When you initialize a Feedback Node inside a single-cycle Timed Loop, the output value is the same as the initial value until the enable terminal is TRUE. If the enable terminal remains FALSE for more than one cycle and the initial value changes from cycle to cycle, the output value also changes.

Enable Terminal

Use the enable terminal, shown as follows, to enable or disable the Feedback Node.



You can display the enable terminal by right-clicking the Feedback Node and selecting **Enable Terminal** from the shortcut menu. If the enable terminal is set to TRUE, the Feedback Node runs as you configure it to run in the Properties dialog box or from the shortcut menu of the node. If the enable terminal is set to FALSE, the Feedback Node continues to output the last value from the previous execution or iteration until the value of the terminal is set to TRUE again.

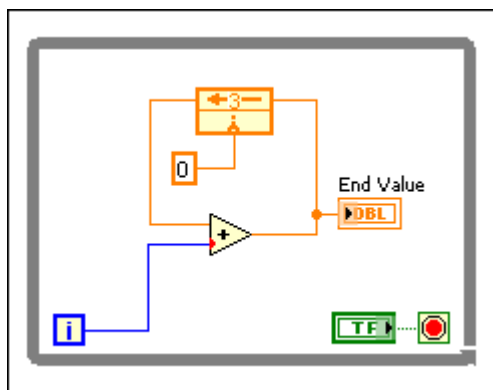
Samples of Data

By default, the Feedback Node stores data from only the previous execution or iteration. However, you can use the Configuration page of the Properties dialog box to configure the Feedback Node to store samples of data from multiple subsequent executions or iterations by increasing the number of executions or iterations to delay the output of the Feedback Node. If you increase **Delay** to more than one execution or iteration, the Feedback Node outputs only the initializer value until the delay you specify is complete. The Feedback Node then begins to output the values it stored one at a time and in subsequent order.



Note (FPGA Module) If you place a checkmark in the **Ignore FPGA reset method** checkbox on the FPGA Implementation page of the Properties dialog box, the LabVIEW FPGA Module removes the reset from the underlying register instantiations. This removal gives the compiler the option to implement the delays using shift register lookup tables (SRLs) instead of flip-flops. SRLs combine many delays into a single lookup table (LUT), which can reduce FPGA resource usage significantly compared to flip-flops.

For example, if you specify a delay of 3, the Feedback Node outputs the initializer value for three loop iterations. On the next iteration, the Feedback Node outputs the value from the first iteration, which is iteration 0, and then the value from the second iteration, and so on. The following block diagram illustrates the behavior of a Feedback Node with an initial value of 0 and a delay of 3.



Loop Iteration	Value the Feedback Node Outputs	End Value	Data the Feedback Node Stores
Before execution or iteration 0	N/A	N/A	0, 0, 0
0	0	0	0, 0, 0

Loop Iteration	Value the Feedback Node Outputs	End Value	Data the Feedback Node Stores
1	0	1	0, 0, 1
2	0	2	0, 1, 2
3	0	3	1, 2, 3
4	1	5	2, 3, 5
5	2	7	3, 5, 7
6	3	9	5, 7, 9
7	5	12	7, 9, 12

In the previous block diagram, the Feedback Node outputs the initializer value, 0, three times before it outputs the first value it received, which was 0.



Note When you set **Delay** to two or more executions or iterations, you can wire an array to the initializer terminal to initialize each delay element to an arbitrary value.

Arrow Direction

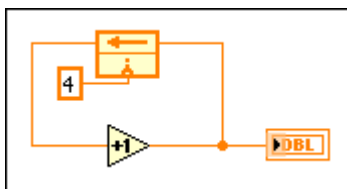
You can change the arrow direction of a Feedback Node to indicate the flow of data along the wire by right-clicking the node and selecting **Change Direction** from the shortcut menu. Changing the arrow direction changes only the cosmetic appearance of the Feedback Node and the wires connected to it on the block diagram and does not change the direction of the flow of data.

Z-Transform View

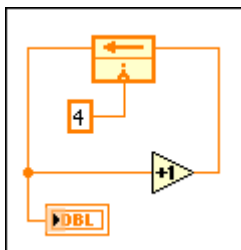
You can change a Feedback Node to display in z-transform view by right-clicking the node and selecting **Z-Transform View** from the shortcut menu. This view changes the appearance of the Feedback Node to look like that of a z^{-1} block in feedback control theory and signal processing. Changing the node to z-transform view changes only the cosmetic appearance of the Feedback Node and does not change the functionality of the node.

Front Panel Indicators and Feedback Nodes

Front panel indicators connected to the same cycle of wires as the Feedback Node display different values depending on their location on the wire relative to the Feedback Node. In the following block diagram, the Feedback Node passes the initial value to the next input node, which operates on the initial value and sends the final value to the indicator and back to the Feedback Node.



In the previous block diagram, the Increment function operates on the initial value of 4 before passing it to the indicator. Thus, the front panel indicator displays 5. However, in the following block diagram, the Feedback Node sends the same data value to the Increment function, but the indicator displays a different value.



In the previous block diagram, the Feedback Node initial value, 4, reaches the indicator before it reaches the Increment function. Although the final value sent to the Feedback Node from the Increment function is 5, the front panel indicator displays the initial value of 4.

Communication With LabVIEW Applications

Communicating with a LabVIEW Application from Web Clients (Real-Time, Windows)

When a LabVIEW stand-alone application runs remotely, you can allow communication with the application by a web client, such as a web browser or another application via HTTP requests. LabVIEW Web services enable the communication between application and client. A Web service runs on the same target as the application and responds to HTTP requests from clients by executing VIs, referred to as HTTP method VIs, designed to transfer data to and from the application.

Web services can also run independently of a LabVIEW application.

Use the following help topics to get started with LabVIEW Web services.

Getting Started - Overview: Web-based Communication with a LabVIEW Application - Tutorial: Creating and Publishing a LabVIEW Web Service Developing a Web Service - Components of a Web Service - Developing HTTP Method VIs - Developing Startup VIs - Integrating Static Content into a Web Service	Running a Web Service - Hosting Web Services - Publishing Web Services Communicating with a Web Service - Transferring Data between Web Services and Clients Creating User Interface for a Web Service - Using WebVIs to Create User Interfaces for Web Services

Getting Started

- Overview: Web-based Communication with a LabVIEW Application
- Tutorial: Creating and Publishing a LabVIEW Web Service

Developing a Web Service

- Components of a Web Service
- Developing HTTP Method VIs
- Developing Startup VIs
- Integrating Static Content into a Web Service
-

Running a Web Service

- Hosting Web Services
- Publishing Web Services

Communicating with a Web Service

- Transferring Data between Web Services and Clients

Creating User Interface for a Web Service

- Using WebVIs to Create User Interfaces for Web Services



RELATED INFORMATION

Overview: Web-based Communication with a LabVIEW Application (Real-Time, Windows) on page 700

Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server (Windows) on page 1552

Components of a Web Service (Real-Time, Windows) on page 704

Developing HTTP Method VIs (Real-Time, Windows) on page 707

Developing Startup VIs to Initialize and Communicate with HTTP Method VIs (Real-Time, Windows) on page 712

Integrating Static Content into a Web Service (Real-Time, Windows) on page 1537

Hosting Web Services (Real-Time, Windows) on page 715

Publishing Web Services (Real-Time, Windows) on page 718

Reading and Writing Shared Variables with the Variable Web Service on page 632

Data Communication Methods in LabVIEW on page 651

Overview: Web-based Communication with a LabVIEW Application (Real-Time, Windows) on page 700

Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server (Windows) on page 1552

Components of a Web Service (Real-Time, Windows) on page 704

Developing HTTP Method VIs (Real-Time, Windows) on page 707

Developing Startup VIs to Initialize and Communicate with HTTP Method VIs (Real-Time, Windows) on page 712

Integrating Static Content into a Web Service (Real-Time, Windows) on page 1537

Hosting Web Services (Real-Time, Windows) on page 715

Publishing Web Services (Real-Time, Windows) on page 718

Reading and Writing Shared Variables with the Variable Web Service on page 632

Overview: Web-based Communication with a LabVIEW Application (Real-Time, Windows)

A web client can exchange data with a remote LabVIEW stand-alone application over a network through LabVIEW Web services. A Web service consists of VIs and other files running on a server that respond to HTTP requests from clients.



NOTE

LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Web services are useful in the following situations:

- Users can invoke the Web service VIs with any HTTP-capable web client, including a standard web browser, to exchange data using a URL and standard HTTP methods such as POST. For example:
 - Uploading new parameters to an application
 - Retrieving current state or status
- Users can remotely monitor and control embedded applications using custom thin clients. For example:
 - Updating application state
 - Starting or stopping a process
- You can conduct application-to-application data exchange between numerous HTTP-capable devices and software from both National Instruments and third parties. For example:
 - Connecting to non-LabVIEW clients
 - Implementing security and authentication

Do not use Web services in the following situations:

- Continuous data streaming
- Rapid polling
- Real-time processing

The following figure shows the common processes involved in LabVIEW Web service applications.



(1) Developing a Web Service

Web services enable communication between LabVIEW applications and web clients. Perform the following actions to develop a Web service:

- On a host computer with LabVIEW installed, create and develop a Web service in a LabVIEW project.
- Develop the fundamental components of a Web service, such as the HTTP method VIs and startup VIs.
- Integrate static content files into the Web service.
- Test and debug the Web service on a web server.

Related Information

[Developing Web Services](#)

(2) Hosting the Web Service

LabVIEW uses the NI Web Server or Application Web Server to host Web services on the network. The web server support for Web services differs depending on how you publish or run Web services.



NOTE

The NI Web Server supports the Windows operating system only. The Application Web Server supports both the Windows operating system and RT targets.

You need to enable and configure the web servers before using them to host Web services. You can also establish secure communication between Web clients and LabVIEW Web services.

Related Information

[Hosting Web Services](#)

(3) Publishing the Web Service

From the project, you can perform the following actions to publish the Web service so that it can run without depending on the LabVIEW Development System:

- Publish the Web service as a stand-alone Web service to the Application Web Server on the host computer or a connected RT target.
- (Windows) Publish the Web service through an NI package to the NI Web Server.
- Publish the Web service through a stand-alone application to the NI Web Server or Application Web Server on a Windows computer or an RT target.

- Publish the Web service through a Windows installer to the Application Web Server on a Windows computer.

A Web service starts and stops running under different conditions, depending on how you publish it to the host server.

Related Information

Publishing Web Services

(4) Transferring Data between the Web Service and Clients

A web client, such as a browser, exchanges data with a Web service by sending an HTTP request to a specific URL. LabVIEW maps a URL to each HTTP method VI, so the specific URL that the client uses determines which HTTP method VI receives the HTTP request. The request might contain values to assign to specific parameters in the HTTP method VI. After each request, the HTTP method VI can process those values and return a response. The Web service returns data to the web client in a particular format that you configure, such as JSON.

Supported web clients include any HTTP-enabled platform, such as standard web browsers, HTML forms, third-party software, and VIs that use the HTTP Client palette.

You can implement communication between any Windows or Embedded LabVIEW application and a Web Service using standard inter-process communication methods. For example, you can use the Network Streams functions to locally transfer data from a LabVIEW application to a Web Service.

In a simple example, a web client might submit an HTTP request that contains two numbers to an HTTP method VI that calculates and returns the sum.

Related Information

Transferring Data between the Web Service and Clients

(5) Creating User Interfaces for Web Services

If you want to create user interfaces to view and interact with data from Web services in a web browser, you can develop Web User Interface (Web UI) applications using the G Web Development Tool. Use the G Web Development tool to create WebVIs, which are specialized VIs that generate HTML, JavaScript, and CSS files when you build your web application. WebVIs can interact with the web services using HTTP and display data in an HTML web page.



NOTE

Although the G Web Development Tool simplifies the process of creating Web UIs, you can use third-party languages that support HTTP communication to create client user interfaces.

Developing Web Services (Real-Time, Windows)

Parent Topic: Overview: Web-based Communication with a LabVIEW Application

Web services enable communication between LabVIEW applications and web clients. You can develop a Web service in a LabVIEW project.



NOTE

For more information about concepts in this topic, refer to the Web services introduction and tutorial. LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

A Web service consists of four major components: HTTP method VIs, HTTP method VI dependencies, startup VIs, and static content files:

- **HTTP method VIs** —Top-level VIs in a Web service. They contain the core functionality of the Web service and exchange data web clients over a network.
- **HTTP method VI dependencies** —SubVIs, shared variables, DAQ channels, and other components that appear within an HTTP method VI.
- **Startup VIs** —VIs that exchange data with HTTP method VIs without using network resources and do not exchange data with a web client.
- **Static content files** —HTML files, JavaScript files, stylesheets, images, and videos. For example, files that WebVIs generate for web pages.

Before you publish a Web service, test the Web service to ensure that clients can invoke and communicate with HTTP method VIs and static content as expected. If you notice unexpected behavior or errors, you can correct them and then test the Web service again.



RELATED INFORMATION

[Overview: Web-based Communication with a LabVIEW Application \(Real-Time, Windows\)](#) on page 700

[Overview: Web-based Communication with a LabVIEW Application \(Real-Time, Windows\)](#) on page 700

[Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server \(Windows\)](#) on page 1552

[Components of a Web Service \(Real-Time, Windows\)](#) on page 704

[Developing HTTP Method VIs \(Real-Time, Windows\)](#) on page 707

[Developing Startup VIs to Initialize and Communicate with HTTP Method VIs \(Real-Time, Windows\)](#) on page 712

[Integrating Static Content into a Web Service \(Real-Time, Windows\)](#) on page 1537

[Testing and Debugging a Web Service on the NI Web Server \(Windows\)](#) on page 1539

[Understanding the Web Services Architecture and File Locations \(Real-Time, Windows\)](#) on page 714

Components of a Web Service (Real-Time, Windows)

Parent Topic: [Developing Web Services](#)

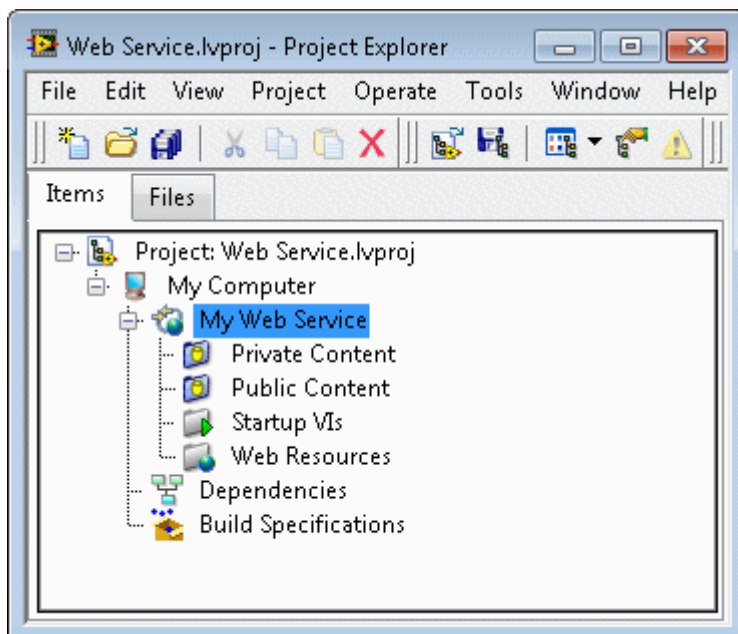
A Web service can include HTTP method VIs, HTTP method VI dependencies, startup VIs, and static content files. Organize Web service files in a Web service project item.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

The following LabVIEW project contains a Web service project item and the subitems that you use to organize and configure Web service files. To add the top-level Web service project item, right-click a target, such as **My Computer**, and select **New»Web Service**.



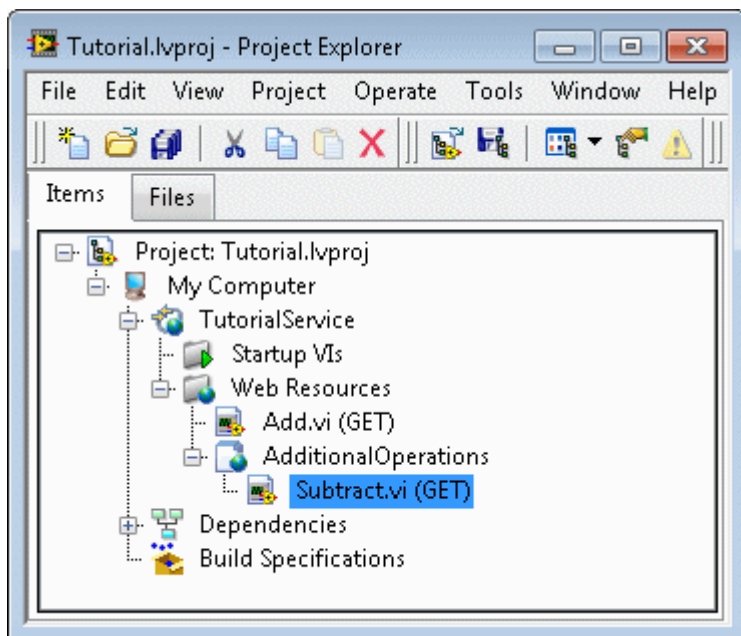
HTTP Method VIs

HTTP method VIs are the top-level VIs in a Web service that contain the core functionality of that Web service. When a web client sends an HTTP request to a Web service, the Web service accepts the request and routes it to a specific HTTP method VI. HTTP method VIs and web clients exchange data over a network.

Right-click the **Web Resources** project item and select **New VI** to create a new HTTP method VI.

You can add sub-items under the **Web Resources** project item to organize HTTP method VIs. These subfolders are useful in RESTful design for grouping HTTP method VIs that perform actions on the same resource, such as a collection of users. RESTful refers to the Representational State Transfer (REST) architecture.

The name of a subfolder also appears in the URL that clients use to invoke HTTP method VIs you place in the subfolder. For example, in the following figure, the URL for the `Subtract.vi` HTTP method VI includes `AdditionalOperations` after the service name: `http://127.0.0.1/TutorialService/AdditionalOperations/Subtract`.



Related Information

[Developing HTTP Method VIs](#)

[Sending Data to a Deployed Application using URLs](#)

HTTP Method VI Dependencies

HTTP method VI dependencies are necessary components of the HTTP method VIs you develop. Dependencies can include subVIs, shared variables, DAQ channels, and other components that appear within an HTTP method VI.

LabVIEW Web services automatically package some HTTP method VI dependencies, such as subVIs, when you publish the Web service. Published Web services do not include HTTP method VI dependencies such as shared variables.

Startup VIs

Startup VIs run as part of a Web service, but unlike HTTP method VIs, they do not exchange data with web clients. You might use a startup VI to perform one or more of the following tasks in a Web service:

- Load data from a file to initialize a value.

- Process and manipulate data the Web service receives from clients.
- Periodically save data to a file while the Web service runs.

A LabVIEW application that includes a Web service also can perform the previous tasks. However, for stand-alone Web services that are not part of an application, use startup VIs instead. Note that startup VIs can exchange data with HTTP method VIs without using network resources, which can increase application efficiency.

Right-click the **Startup VIs** project item and select **New VI** to create a new startup VI. Each Web service can include multiple startup VIs.

Related Information

[Developing Startup VIs to Initialize and Communicate with HTTP Method VIs](#)

Static Content

You can publish static content alongside HTTP method VIs in a Web service. Examples of static content are HTML files, JavaScript files, stylesheets, images, and videos. When you use WebVIs to create user interfaces for Web services, the web page files that WebVIs generate are static content.

Right-click the Web service and select **Add Public Content Folder** or **Add Private Content Folder** to add folders to the project and map them to folders of static content on disk. Organize static content on disk in two main folders to mirror the organization in the project tree—one main folder that contains public content and one that contains private content:

- **Private Content** —Clients cannot access files in the private folder via HTTP requests. Private files do not have URL mappings. Only VIs in the Web service can use private static content. For example, if your Web service relies on configuration files to execute, you might want to make those configuration files private.
- **Public Content** —Clients can access public static content via HTTP requests. Examples of files you might want to make public are HTML pages and images. LabVIEW automatically assigns each public file a URL mapping.

The main folders can contain subfolders, such as `Public Folder\html` and `Public Folder\images`. The folders in the project tree are auto-populating by default, which means they update in real time to reflect the contents of folders on disk.

Related Information

[Integrating Static Content into a Web Service](#)

Developing HTTP Method VIs (Real-Time, Windows)

Parent Topic: [Developing Web Services](#)



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

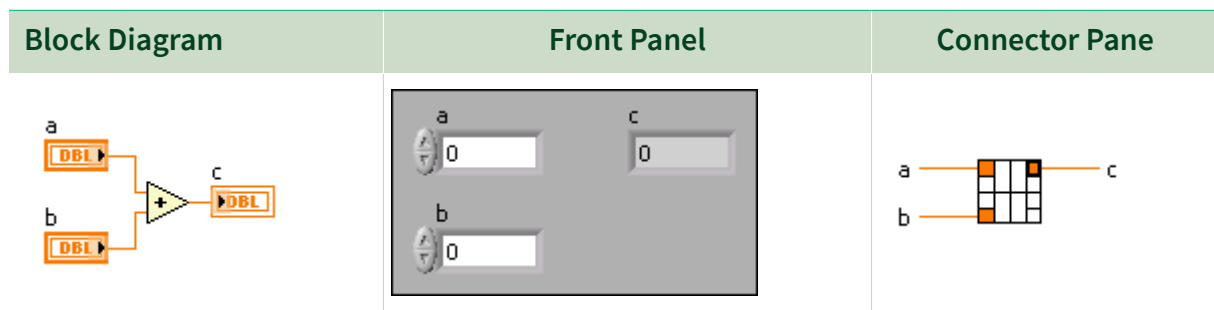
To create an HTTP method VI, right-click **Web Resources** or a web resource sub-item in the Project Explorer window and select **New VI**. LabVIEW opens the new VI, which contains a [LabVIEW Web Service Request](#) control you commonly use in HTTP method VIs.

Consider the guidelines in the following sections for developing HTTP method VIs.

Transferring Data between HTTP Method VIs and Web Clients

[HTTP method VIs](#) receive and send data using their connector pane terminals. When a web client sends an HTTP request to an HTTP method VI, the VI receives values for controls on its connector pane, processes those values, and returns a response to the client.

The following figure displays an example front panel, block diagram, and connector pane for an HTTP method VI that accepts two values from a web client and returns their sum.



Labels of controls and indicators that receive and return values in Web services can contain only letters, numbers, hyphens, and underscores available in the US ASCII character set. Labels cannot contain spaces or special characters.

Related Information

[Sending Data to a Deployed Application using URLs](#)

[Assigning Terminals to Controls and Indicators](#)

Choosing the Format of Output Data

By default, an HTTP method VI returns data to web clients as a JSON string. The JSON string includes a summary of the indicator values assigned to the connector pane of the HTTP method VI. You also can configure an HTTP method VI to return data as an XML-, HTML-, or plain text-formatted response. Configure the output format type on the HTTP Method VI Settings page of the **Web Service Properties** dialog box.



Note Instead of returning data to a client via the connector pane, an HTTP method VI can stream data to a client in a custom format, such as a customized HTML page or image data.

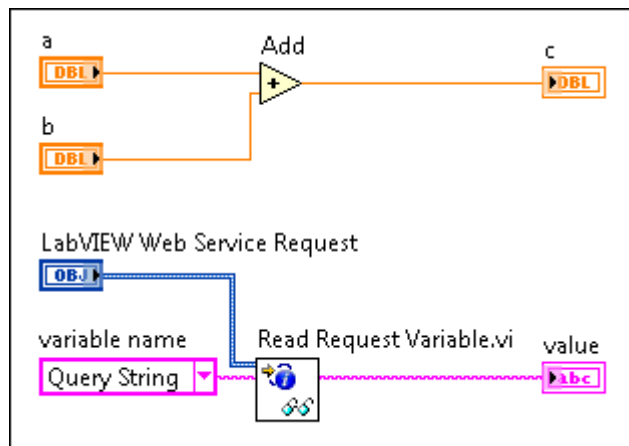
Related Information

[Configuring HTTP Headers, Streaming, and Buffering](#)

Using the VIs on the Web Services Palette

The VIs on the Web Services palette can handle POST data, form data, HTTP sessions, cookies, and other Web-specific functionality. When you include VIs from the Web Services palette within an HTTP method VI, you must wire the **LabVIEW Web Service Request** input on each VI so they uniformly handle the current HTTP request. The **LabVIEW Web Service Request** input functions similarly to a refnum in LabVIEW.

The following figure shows the use of the **LabVIEW Web Service Request** input to identify the HTTP request for which the Read Request Variable VI returns data:



Note You must assign the control wired to **LabVIEW Web Service Request** to the connector pane of the top-level HTTP method VI.

If an HTTP method VI contains multiple VIs from the Web Services palette, you can wire a single control to the **LabVIEW Web Service Request** input on the first Web Services VI, and then wire the subsequent VIs together using the **LabVIEW Web Service Request** input and **LabVIEW Web Service Request out** output of each VI.

Using Other Built-in LabVIEW Nodes

HTTP method VIs can include all of the available functionality of LabVIEW that uses *supported data types*.

Related Information

- [Processing POST Data, Form Data, and Uploaded Files](#)
- [Storing Values During HTTP Sessions when Using the Application Web Server](#)
- [Accessing HTTP Request Information](#)
- [Configuring HTTP Headers, Streaming, and Buffering](#)



Processing POST Data, Form Data, and Uploaded Files (Real-Time, Windows)

Parent Topic: Developing Web Services

When a web client sends an HTTP request to a Web service, the Web service accepts the request and routes it to the HTTP method VI specified in the URL. That HTTP method VI uses the connector pane and VIs on the Web Services palette to process the request and return a response. A web client might send data by appending values to a URL or by using standard HTML forms configured for HTTP methods such as POST.



NOTE

For more information about concepts in this topic, refer to the Web services introduction and tutorial. LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Configuring the HTTP Method VI

To configure an HTTP method VI to accept POST data, set its HTTP method as POST. Right-click the VI in the LabVIEW project and select **POST**.

Developing the POST HTTP Method VI

By default, a web client submits a URL with values that correspond directly to controls of the HTTP method VIs. The HTTP method VIs use the connector pane to route the values to the controls. You also can configure the HTTP method VIs to return the stream output data on the HTTP Method VI Settings page of the **Web Service Properties** dialog box. For more advanced data, you can use the VIs from the Web Services palette to process the data from the web client's request. Use the following VIs to accept form and POST data:

- Read All Form Data
- Read Form Data
- Read Postdata

You must wire a **LabVIEW Web Service Request** control to the connector pane of any HTTP method VI that receives POST data and includes VIs from the Web Services palette.

Accepting Uploaded Files from a Web Client

LabVIEW stores information about each file uploaded by a web client in an array of clusters. The information for each uploaded file includes the client filename, the path to the temporary location, the content type, and the size (bytes). When you develop HTTP method VIs, include the Read Uploaded Files Info VI to return information regarding all uploaded files associated with an HTTP request. Use this information to perform actions on the uploaded files, such as moving them to a permanent location.

You must wire a **LabVIEW Web Service Request** control to the connector pane of any HTTP method VI that receives uploaded files and includes the Read Uploaded Files Info VI.



RELATED INFORMATION

[Developing Web Services \(Real-Time, Windows\)](#) on page 703

[Sending Data to a Deployed Application Using URLs \(Real-Time, Windows\)](#) on page 721

[Using the POST HTTP Method \(Real-Time, Windows\)](#) on page 724

[Overview: Web-based Communication with a LabVIEW Application \(Real-Time, Windows\)](#) on page 700

[Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server \(Windows\)](#) on page 1552

[Uploading Files Using Web Services \(Real-Time, Windows\)](#) on page 725

[Data Communication Methods in LabVIEW](#) on page 651

Accessing HTTP Request Information (Real-Time, Windows)

Parent Topic: Developing HTTP Method VIs

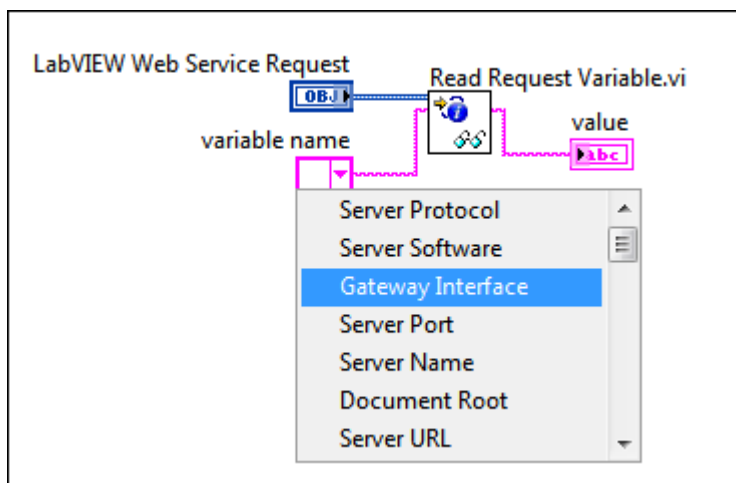
HTTP environment variables, or request variables, provide information regarding over 30 aspects of the Web service including server label, remote address, and cache control. Use the **variable name** control on the Read Request Variable VI to select a specific request variable from a pull-down menu. Use the Read All Request Variables VI to return all request variables as an array for the current HTTP request.



NOTE

For more information about concepts in this topic, refer to the Web services introduction and tutorial. LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

The following figure shows code that uses the Read Request Variable VI to access one of the available request variables that describe characteristics of the HTTP request.



RELATED INFORMATION

[Developing HTTP Method VIs \(Real-Time, Windows\)](#) on page 707

[Overview: Web-based Communication with a LabVIEW Application \(Real-Time, Windows\)](#) on page 700

[Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server \(Windows\)](#) on page 1552

Configuring HTTP Headers, Streaming, and Buffering (Real-Time, Windows)

Parent Topic: Developing HTTP Method VIs



NOTE

For more information about concepts in this topic, refer to the Web services introduction and tutorial. LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Streaming Data

You can stream output data to a client from an HTTP method VI. Use the stream output type to return data from an HTTP method VI in a customized format. You can configure the HTTP method VI to use the stream output type on the HTTP Method VI Settings page of the **Web Service Properties** dialog box.

Use the Write Response VI to create the stream output data, such as a customized HTML page or image data.

Use the Set HTTP Response Code VI and the Set HTTP Response MIME Type VI to customize the response code and MIME type of the stream output data.



Note Instead of streaming data to a client in a custom format, such as a customized HTML page or image data, an HTTP method VI can return data to a client via the connector pane.

Configuring HTTP Headers

You can customize the HTTP headers of the response from an HTTP method VI to a web client. Use the Set HTTP Header VI and the Set HTTP Redirect VI to customize HTTP headers for HTTP method VIs using the stream output type. For example, you can create a custom header that refreshes the web page displayed in a web browser every two seconds. If you do not customize the HTTP headers of the HTTP method VI, the HTTP method VI uses default headers for the response.

Buffering Streaming Data

You can configure an HTTP method VI to buffer the stream output data. Use the Write Response VI to create either buffered or unbuffered stream output data. Use the HTTP Method VI Settings page of the **Web Service Properties** dialog box to configure an HTTP method VI to buffer the stream output data.



RELATED INFORMATION

[Developing HTTP Method VIs \(Real-Time, Windows\)](#) on page 707

[Overview: Web-based Communication with a LabVIEW Application \(Real-Time, Windows\)](#) on page 700

[Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server \(Windows\)](#) on page 1552

[Data Communication Methods in LabVIEW](#) on page 651

Developing Startup VIs to Initialize and Communicate with HTTP Method VIs (Real-Time, Windows)



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Startup VIs run as part of a Web service application but do not exchange data with a web client. A startup VI runs once, when the Web service loads, and can continue to run as long as the Web service is published. Consider the following template block diagram for startup VIs:



To create a startup VI that uses the previous template, right-click **Startup VIs** under a Web service project item Project Explorer window and select **New VI**.

Programming Guidelines

Modify the startup VI template to meet the needs of your application. For example, consider the following changes you might make:

- If you only need to perform an initialization action, you can delete the processing and clean-up code.
- Remove or adjust the Wait (ms) function in the processing loop according to the timing requirements of the VI. Or, if the startup VI waits on data from a queue, use a timeout instead of a wait.



[Data Communication Methods Home](#)

Understanding the Web Services Architecture and File Locations (Real-Time, Windows)

Parent Topic: [Developing Web Services](#)

This topic describes the Web services architecture, supported data types in Web services, and the Web services file system.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Web Services Architecture

LabVIEW Web services use RESTful Web service architecture, which is based upon the Representational State Transfer (REST) architecture. RESTful provides a lightweight protocol accessible to a wide variety of HTTP-enabled clients and does not require complex message parsing.

Supported Data Types in Web Services

Web services support the following data types:

- string
- path
- Boolean
- numeric (except complex numbers)
- enumeration (unsigned 8-bit, 16-bit, and 32-bit)
- array (JSON and XML output only)

- cluster (JSON and XML output only)
- digital/analog waveform (JSON and XML output only)
- timestamp, formatted as string (JSON and XML output only)

Web Services File System and LVWS Files



Note The following section applies only to Web services you include in LabVIEW stand-alone applications.

LabVIEW uses a .lvws file to publish Web services to the web server that hosts them. When you publish a Web service, LabVIEW packages all necessary files into a zip file with the .lvws extension and deploys the file to the target web server. LabVIEW then unzips the file and installs the Web service components into a unique directory on the target web server.

Hosting Web Services (Real-Time, Windows)

Parent Topic: [Overview: Web-based Communication with a LabVIEW Application](#)

LabVIEW uses one of two types of web servers to host Web services on the network:

- **NI Web Server**—The NI Web Server is a production-grade web server that can host user-authored services, such as LabVIEW Web services, and SystemLink services created by NI. The NI Web Server protects web applications against common web security threats, provides high scalability to many enterprise-grade data services, and allows device management.
- **Application Web Server**—The Application Web Server is developed based on a third-party, open-source web server which is fast, secure, and compact.

The web server support for Web services differs depending on how you publish or run Web services. The following table shows which methods of publishing or running a Web service are supported for each type of web server.

Method for Publishing/Running	NI Web Server	Application Web Server
Testing and debugging	✓	✓ (Debug Server)
Publishing through a package	✓	✗
Publishing through an installer	✗	✓

Method for Publishing/Running	NI Web Server	Application Web Server
Publishing through a stand-alone application	✓	✓ (Application-specific)
Publishing as a stand-alone Web service	✗	✓

Understanding the NI Web Server

The NI Web Server can host stand-alone Web services running on the host computer and Web services included in a package or a stand-alone application. You can run the NI Web Server without running the LabVIEW Development System. Once you enable the NI Web Server, it launches whenever you boot up the system. Before you run or publish a Web service, you might need to set up the NI Web Server.



NOTE
Checking the status of Web services hosted on the NI Web Server is unsupported yet.

Related Information

- [Setting up the NI Web Server for Web Services](#)
- [Configuring Web Services Security when Using the NI Web Server](#)

Understanding the Application Web Server

The Application Web Server has three different modes depending on how you use it.

- [System-Wide Application Web Server](#)
- [Application-Specific Application Web Server](#)
- [Debugging Server](#)

System-Wide Application Web Server

The system-wide Application Web Server can host stand-alone Web services and Web services included in an installer. The Application Web Server runs as an independent system service. You can enable and run the Application Web Server without running the LabVIEW Development System. Once you enable the Application Web Server, it launches whenever you boot up the system as `ApplicationWebServer.exe`. Before you run or publish a Web service, you might need to set up the Application Web Server.

Complete the following steps to check the status of Web services hosted on the Application Web Server:

1. Right-click the Web service project item in the project tree and select **Application Web Server»Manage Web Server**. The NI Web-based Configuration & Monitoring utility opens in a web browser.



2. Click the **Web Services Management** button on the left-hand sidebar.
3. Any published Web services appear in the **Published Web Services** list. Click the **Refresh** button to update the list.

Related Information

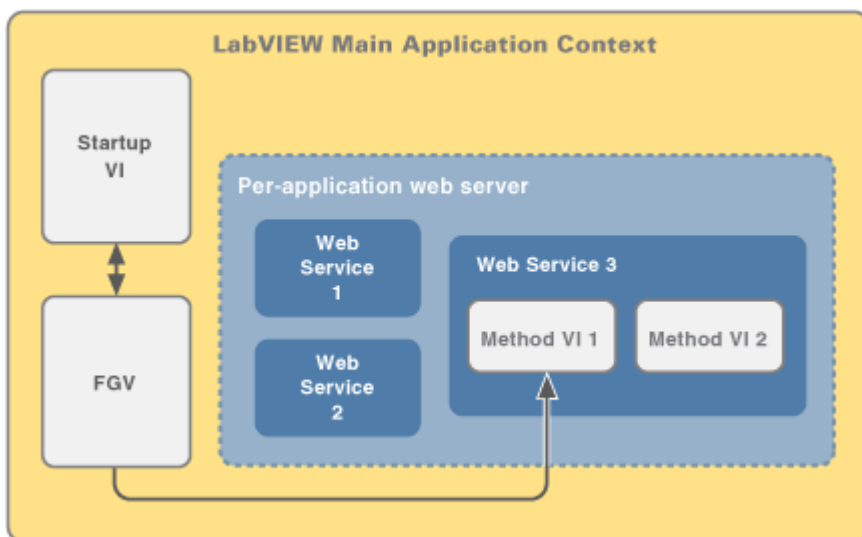
[Setting up the Application Web Server for Web Services](#)

[Configuring Web Services Security when Using the Application Web Server](#)

Application-Specific Application Web Server

When you include a Web service in a LabVIEW stand-alone application, the application acts as a web server that hosts the Web service directly. LabVIEW unpublishes the Web service when the application exits.

The LabVIEW built application starts its own web server that hosts the included Web services. Because each VI in the Web service is running in the same context as the application's startup VI, you can use simple function global variables (FGV) to pass data between them. The following illustration shows an example of how a LabVIEW built application starts its own web server.



You can configure SSL and the port you want to use to accept requests for the Web service when you specify to include the Web service in the application build specification.

Related Information

[Publishing a Web Service through a Stand-Alone Application](#)

Debugging Server

When you start debugging a Web service, LabVIEW places the Web service on a debugging server, where it behaves as if published on a target.

The default port that web clients use to send requests to the Web service in debugging mode is 8001. To use a different port for debugging, select **Tools»Options**, browse to the Web Server page, and modify the **Debug HTTP Port** option.

Related Information

[Testing and Debugging a Web Service on the Application Web Server](#)

Other LabVIEW Web Servers

LabVIEW also includes a System Web Server for internal Web service processes. The System Web Server also runs as an independent system service. After you install LabVIEW, the System Web Server launches whenever you boot up the system as `SystemWebServer.exe`. You cannot disable the System Web Server.

LabVIEW includes a separate web server that hosts *[remote front panels](#)*.

Related Information

[Publishing Web Services](#)

Publishing Web Services (Real-Time, Windows)

You can publish a Web service to the host computer, an RT target, or another Windows computer, so that the Web service can run without depending on the LabVIEW Development System. The process of publishing a Web service differs depending on which platform you use to host the Web service and how you want to distribute the Web service to the deployment target. The lifespan of the Web service also differs according to the method you use to publish the Web service.



NOTE

For more information about concepts in this topic, refer to the *[Web services introduction](#)* and *[tutorial](#)*. LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Refer to the following table to choose the appropriate method for publishing a Web service.

Use Case	Deployment Target	Hosting Server	Method for Publishing	Starts/Stops Running	Context
The target is a Windows computer with NI Package Manager installed.	Windows	NI Web Server	Publish the Web service through a package .nipkg.	After you install/remove the package	A unique context for each Web service within the AQMP Web Service Host process
The target is a Windows computer without NI software installed.	Windows	NI Web Server	Publish the Web service through a package installer.	After you finish the installation/removal process	A unique context for each Web service within the AQMP Web Service Host process
Application Web Server	Publish the Web service through an installer.	After you finish the installation/removal process	A unique context for each Web service within Application Web Server process		
The Web service needs to communicate with a stand-alone application and runs only within the application life time.	Windows	NI Web Server	Publish the Web service through a stand-alone application .exe.	When the application starts/stops running	Main application instance, where the startup VI of the stand-alone application runs
Windows and Real-Time	Application Web Server				

Use Case	Deployment Target	Hosting Server	Method for Publishing	Starts/Stops Running	Context
The target is the host computer or a connected RT target. The Web service does not communicate with a stand-alone application.	Windows and Real-Time	Application Web Server	Publish as a stand-alone Web service.	When you publish/unpublish the Web service in your LabVIEW project	A unique context for each Web service within Application Web Server process



Note The NI Web Server supports the Windows operating system only. The Application Web Server supports both the Windows operating system and RT targets.

Transferring Data between Web Services and Clients (Real-Time, Windows)

By default, web clients use URLs and HTTP methods to exchange data with controls on the connector pane of HTTP method VIs. For example, if you create an HTTP method VI that contains two numeric controls assigned to the connector pane, a web client can send values to those controls using a URL or HTTP method. The HTTP method VI can then return a response such as the sum of the numeric controls. The default response format is a JSON string. The JSON string includes a summary of the indicator values assigned to the connector pane of the HTTP method VI. You can configure the response format to return data as an XML-, HTML-, or plain text-formatted response. Configure the response format type on the HTTP Method VI Settings page of the **Web Service Properties** dialog box.



NOTE
Instead of returning data to a client via the connector pane, an HTTP method VI can stream data to a client in a custom format, such as a customized HTML page or image data.

HTML forms, web browsers, the HTTP Client VIs, and HTTP-capable third-party applications can exchange data with HTTP method VIs using HTTP methods.



Note Each time a client reconnects to an HTTP method VI, such as when it refreshes the browser, the client sends a new HTTP request.

HTTP Methods

Web services support four standard HTTP methods used in web communication:

- **POST**—Use to send data and upload files. You can use a common HTML form or other compatible client to submit information, known as POST data, via the POST method.
- **GET**—Use to retrieve data from a Web service.
- **PUT**—Serves as an alternative method to POST for uploading files. Under RESTful design, PUT and POST have different use cases.
- **DELETE**—Use to instruct the Web service to delete a specified resource.

Refer to the World Wide Web Consortium at www.w3.org for more information about HTTP method definitions, including the POST, GET, PUT, and DELETE methods.

Related Information

- [Understanding URL Mapping in Web Services](#)
- [Sending Data to a Deployed Application using URLs](#)
- [Using the POST HTTP Method](#)
- [Uploading Files using Web Services](#)
- [Understanding the Web Services Architecture and File Locations](#)



[Data Communication Methods Home](#)

Sending Data to a Deployed Application Using URLs (Real-Time, Windows)

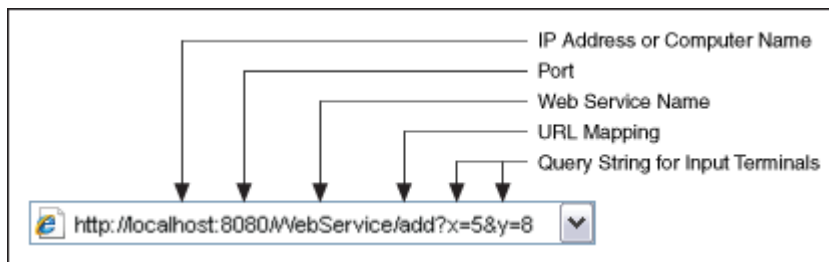
Parent Topic: [Transferring Data between Web Services and Clients](#)



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Clients use URLs and HTTP methods to transmit data directly to controls on the connector pane of HTTP method VIs, as well as to send values as POST data using the POST HTTP method. The URL that web clients use to exchange data with HTTP method VIs is built upon various values determined when you create a LabVIEW Web service. The following figure shows an example URL that a web client might use to exchange data with an HTTP method VI.



- **IP address or computer name** —If the web client accesses the Web service from the same system that runs the host web server, use the value `localhost`. If the web client accesses a Web service on a remote target, use the target's IP address or system name. Use NI Web Server Configuration to customize the hostname for the NI Web Server.
- **Port** —Matches the port of the host web server. NI Web Server uses 80/443 as the default port in accordance with HTTP/HTTPS protocol. However, the port number might change if another application occupies the port. You can customize the port number in NI Web Server Configuration. Application Web Server uses 8080 as the default port. You can customize the port number in NI Web-based Configuration & Monitoring.
- **Web service name** —Matches the **Service name** option on the Service Settings page of the **Web Service Properties** dialog box.
- **URL mapping** —Matches the [URL mapping](#) from the HTTP Method VI Settings page of the **Web Service Properties** dialog box. For each HTTP method VI, LabVIEW defines a default mapping that includes the web resource to which the VI belongs, if applicable, followed by the VI name.
- **Query string for input terminals** —Matches labels of controls assigned to the connector pane of the HTTP method VI. Append this query string to a URL mapping using the syntax in the previous illustration. For example, the query string `?x=5&y=8` passes data to controls labeled **x** and **y** on the connector pane of an HTTP method VI whose URL mapping is `add`. Note that you separate label-value pairs with an ampersand (&).

Related Information

[Understanding URL Mappings and Query Strings in Web Services](#)

[Hosting Web Services](#)

Understanding URL Mappings and Query Strings in Web Services (Real-Time, Windows)

Parent Topic: [Sending Data to a Published Application using URLs](#)



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Web clients use URLs to exchange data with published Web services. URL mappings and query strings are parts of the URLs:

- **URL mapping** —Serves as the identifier the host web server uses to direct an HTTP request to a specific HTTP method VI or static file. Each HTTP method VI and static file has one URL mapping.
- **Query string** —Identifies and assigns values to input parameters on the connector pane of the HTTP method VI. You can append input parameters to the end of a URL mapping by appending a question mark, a control label, an equals sign, and the value to assign the control.

For an HTTP method VI named `Add.vi`, LabVIEW automatically defines the URL mapping as `/Add`. If the VI contains the inputs **a** and **b**, the URL and query string `/Add?b=2&a=3` assigns values to the inputs. In this example, the URL a client uses is `http://127.0.0.1:8080/WebService/Add?b=2&a=3`.



Note Instead of using default URL mappings, you can [customize URL mappings](#), which affects how you assign values to input parameters of an HTTP method VI.

Omitting Input Parameter Values from Query Strings

A client can omit any input parameters from the query string. If a client omits an input parameter value, the Web service uses the default value for the input terminal stored in the VI. For example, given an HTTP method VI named `Add.vi` with controls labeled **a** and **b**, HTTP requests using the following URL mappings and query strings are valid:

- `/Add?b=2&a=3`
- `/Add?a=3` (uses default value of **b** control)
- `/Add?b=2` (uses default value of **a** control)
- `/Add` (uses default values of both controls)

Customizing URL Mappings

Typically, you can use the default URL mappings for a Web service. However, in advanced use cases, you might want to customize the URL mapping for an HTTP method VI. Complete the following steps to configure a custom URL mapping for an HTTP method VI:

1. Right-click the Web service project item and select **Properties**.
2. Browse to the HTTP Method VI Settings page of the **Web Service Properties** dialog box.
3. Select the HTTP method VI whose URL mapping you want to customize from the table of VIs.
4. On the **URL Mapping** tab, remove the checkmark from **Use standard URL mapping** to make the **URL** textbox editable. You now can edit the portion of the URL after the name of the Web service and the web resource to which the VI belongs, if

applicable. In other words, you cannot edit the IP address, port, Web service name, or web resource name (if any).

You can configure the URL mapping string as any value as long as the value begins with a forward slash and uses standard HTTP syntax. You can append input parameters to the end of a URL mapping by adding a forward slash, a colon, and the label of a connector pane input in the associated HTTP method VI. Note that this syntax is different from how you append query strings to default URL mappings to set parameter values.

An example of a custom URL mapping string is `/Add/ : a/ : b`. The URL mapping string `/Add` is the base of the URL mapping and `/ : a/ : b` configures input parameters for the HTTP method VI. If the URL mapping contains no input parameters to pass to the HTTP method VI, the VI must contain default values for any input terminals.

Passing Multiple Fragments using URL Mappings

You can create a custom browsing URL that assigns multiple fragments of the URL to a single string input terminal of the HTTP method VI. This technique is useful for passing file paths or other hierarchical values using a custom browsing URL.

For example, `/ : x/*y` creates an input parameter for a control labeled `x` and a control labeled `y` when appended to a URL mapping. You then can append a value, such as `/test/C/directory`, to the custom browsing URL that corresponds to the input parameter `/ : x/*y` that you added to the URL mapping. In this example, `x` is assigned `test` and `y` is assigned `C/directory`. The input parameter that contains more than one fragment must occur at the end of the custom browsing URL.

Using the POST HTTP Method (Real-Time, Windows)

Parent Topic: [Sending HTTP Requests to Web Services](#)

You can use the POST [HTTP method](#) to send data to an HTTP method VI. You can use a common HTML form, the HTTP Client VIs, or another compatible client to gather and submit POST data. Use the POST method instead of a URL mapping when sending a large quantity of data to an HTTP method VI.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Sending a POST Request using HTML Forms

The following example shows how to use an HTML form to submit POST data to a Web service. The Web service name is `MyWebService`. The [URL mapping](#) is `/add` and is configured to accept POST data on the HTTP Method VI Settings page of the **Web Service Properties** dialog box.

The following HTML code creates two value fields and a submit button:

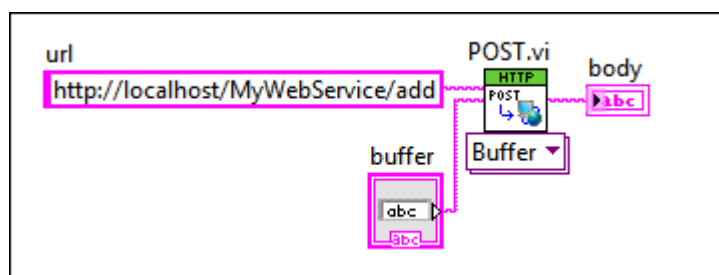
```
<form name="send" action="/MyWebService/add" method="POST">
<input type="text" name="x"><input type="text" name="y">
<input type="submit" value="Submit"></form>
```

If you assign the values 2 and 3 to the respective inputs labeled x and y and click the submit button, the HTML form submits the following POST data:

```
POST /MyWebService/add/ x=2&y=3
```

Sending a POST Request using the HTTP Client VIs

You can use the POST VI and the POSTMultipart VI to create a web client in LabVIEW that sends POST requests to a deployed Web service. The following figure shows code that uses the POST VI to perform a similar HTTP request as the HTML form in the previous example.



Uploading Files Using Web Services (Real-Time, Windows)

Parent Topic: [Sending HTTP Requests to Web Services](#)

Use the POST or PUT [HTTP methods](#) to upload files to a Web service. The Web service stores the files as .tmp files in a temporary directory.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Uploading a File using HTML Forms

The following HTML code creates an HTML form that uses the POST method to upload a file to a Web service. The Web service name is MyWebService. The URL mapping is /upload.

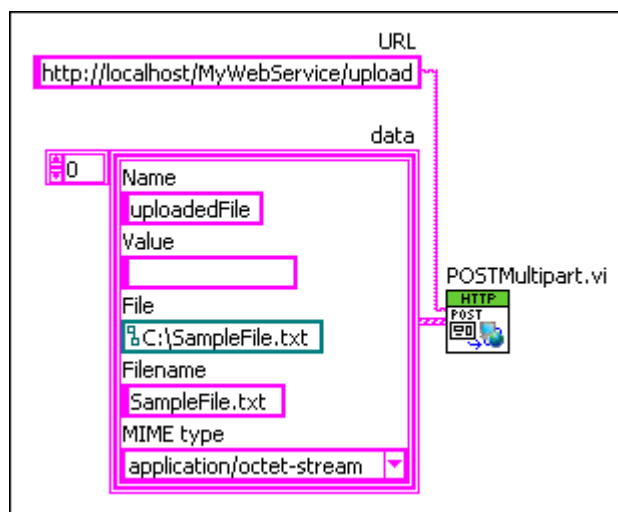
```
<form name="send" action="/MyWebService/upload"
enctype="multipart/form-data" method="POST"><input
```

```
type="file" name="uploadedFile"> <input type="submit"
value="Submit"> </form>
```

This HTML code creates a form with one value field and a submit button. If you assign a file on the client-side, `SampleFile.txt` for example, to the field labeled `uploadedFile` and click the submit button, the HTML form submits `SampleFile.txt` to the Web service.

Uploading a File using the HTTP Client VIs

Use the POST, POSTMultipart, and PUT VIs to create a web client in LabVIEW that uploads files to a Web service. The following figure shows code that uses the POSTMultipart VI to upload a file similar to the HTML form in the previous example.



[Data Communication Methods Home](#)

Importing Web Services

You can use Web services without managing all the complexities behind Web services. The Import Web Service wizard transforms any Web service into a library of VIs that you then can use to program the Web service as if it were independently available on the local computer. To import a Web service, you must provide a valid URL to a Web Service Description Language (WSDL) in the **Import Web Service** wizard, and the WSDL must validate correctly with ASP.NET. WSDL is an XML-formatted language used to describe a Web service and its functionality.



NOTE

- You must have the .NET Framework 4.0 installed to use the Import Web Service wizard.
- If you change the WSDL URL for the Web Service you want to import, you must regenerate the VIs to use the new URL for the Web Service.

Application Web Server (Real-Time, Windows)

Parent Topic: Overview: Web-based Communication with a LabVIEW Application (Real-Time, Windows)

You can use either NI Web Server or Application Web Server to host your Web services. Topics under this book are Application Web Server specific.



NOTE

For more information about concepts in this topic, refer to the Web services introduction and tutorial. LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.



RELATED INFORMATION

[Overview: Web-based Communication with a LabVIEW Application \(Real-Time, Windows\)](#) on page 700

[Overview: Web-based Communication with a LabVIEW Application \(Real-Time, Windows\)](#) on page 700

[Tutorial: Creating and Publishing a LabVIEW Web Service to the Application Web Server \(Real-Time, Windows\)](#) on page 1561

Storing Values During HTTP Sessions when Using the Application Web Server (Real-Time, Windows)



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

An HTTP session is a collection of the [HTTP requests](#) made between a web client and a Web service. Use HTTP sessions to save data such as authentication credentials or variables across multiple HTTP requests from a single web client. For example, a user can enter a username at the start of an HTTP session and the Web service can save that data for subsequent HTTP requests.



Note Web services store persistent data as session variables within HTTP cookies on the client system. You must enable cookies within the web client, such as a browser, to use HTTP sessions.

Use the following VIs to create and destroy an HTTP session; write, read, and delete session variables; and check if a current session exists for a particular web client:

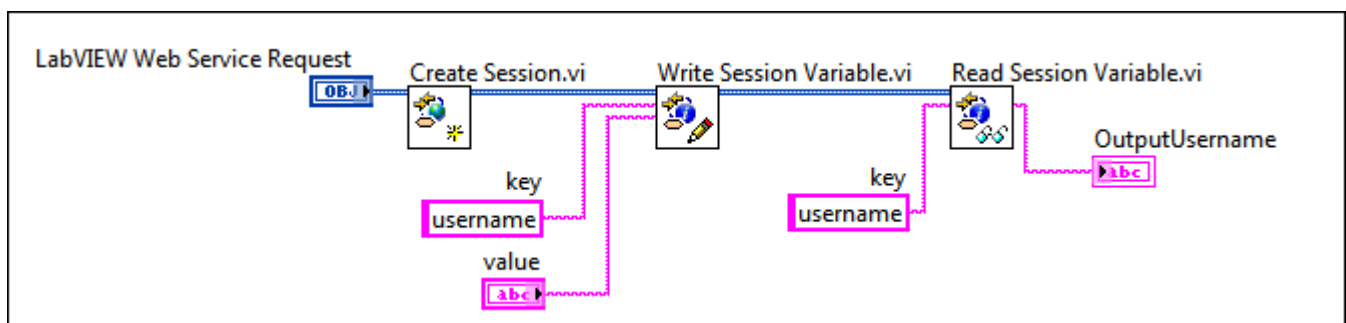
- Check If Session Exists

- Create Session
- Write Session Variable
- Get Session ID Cookie
- Read Session Variable
- Read All Session Variables
- Delete Session Variable
- Destroy Session

For example, you can create a session variable that represents a username. A client transmits a value for the username in the initial HTTP request. Using a session variable, the HTTP method VI saves a cookie on the client system. The client then can make HTTP requests without having to re-enter that value. Without an HTTP session and session variables, the username value resets each time the client makes an HTTP request.

Use cookies to enable the client to continue a specific HTTP session for multiple HTTP requests when using a web browser or other client that supports cookies. You can generate a session ID cookie when you use the Create Session VI to create a new HTTP session. Use the Get Session ID Cookie VI to use cookies to resume a specific HTTP session.

The following figure shows code from an HTTP method VI that establishes an HTTP session, writes a session variable `username`, and accepts a **value** from the web client. The Read Session Variable VI then reads the value of `username`, which you can use within the HTTP method VI across subsequent HTTP requests.



Related Information

Developing HTTP Method VIs

Viewing and Controlling Front Panels Remotely with the Web Server

You can view a VI front panel remotely, either from within LabVIEW or from within a Web browser, by connecting to the LabVIEW built-in Web Server. Additionally, you can also control a VI front panel remotely within LabVIEW.

When you open a front panel remotely from a client, the Web Server sends the front panel to the client, but the block diagram and all the subVIs remain on the server computer. When controlling a VI front panel remotely, you can interact with the front panel in the same way as if the VI were running on the client, except the block diagram executes on the server. Use this feature to publish entire front panels or to control your remote applications safely, easily, and quickly.



NOTE

Use the LabVIEW Web Server if you want to control entire VIs. Use [shared variables](#) to read and write data on a single front panel control in a VI.

Configuring the Server for Clients

The user at the server computer must first configure the server before a client can view or control a front panel remotely. Configure the Web Server by selecting **Tools»Options** and selecting the **Web Server** pages from the **Category** list. Use these pages to control browser access to the server and to specify which front panels are visible remotely. You also can use these pages to set a time limit on how long a remote client can control a VI when multiple clients are waiting to control the VI.

The Web Server allows multiple clients to connect simultaneously to the same front panel, but only one client at a time can control the front panel. The user at the server computer can regain control of any VI at any time. When the controller changes a value on the front panel, all client front panels reflect that change. However, client front panels do not reflect all changes. In general, client front panels do not reflect changes made to the display on front panel objects, but rather to the actual values in the front panel objects. For example, if the controller changes the mapping mode or marker spacing of a scale of a chart or if the controller shows and hides a scroll bar for a chart, only the controller front panel reflects these changes.

Viewing and Controlling Front Panels in LabVIEW or from a Web Browser

A client can only view a front panel remotely from a Web browser. However, a client can both view and control a front panel remotely using LabVIEW.

To remotely control a front panel from LabVIEW, the client and server computers must be running the same version of LabVIEW. If you are using a browser to view a remote front panel, you must use a version of the LabVIEW Run-Time Engine compatible with the version of LabVIEW on the server computer. Also, contact the server administrator to verify that the HTML document specifies the correct version of the LabVIEW Run-Time Engine.



Note Before you can view and control a front panel remotely, you must enable the Web Server on the server computer where the [VI](#) or [application](#) you want to view and control is located.

Viewing Front Panels from a Web Browser

If you want clients who do not have LabVIEW installed to be able to view a front panel remotely, they can use a Web Browser.

Follow the procedure to allow clients to remotely view the front panel using a Web Browser.

1. Using the Web Publishing Tool on the Web Server, publish the VI.
2. Provide the generated web address to the clients.

Viewing and Controlling Front Panels in LabVIEW

To view a remote front panel using LabVIEW as a client, select **Operate»Connect to Remote Panel** to display the Connect to Remote Panel dialog box. Use this dialog box to specify the server Internet address and the VI you want to view. By default, the remote VI front panel is initially in observer mode. You can request control by placing a checkmark in the **Request control** checkbox in the **Connect to Remote Panel** dialog box when you request a VI. When the VI appears on your computer, you also can right-click anywhere on the front panel and select **Request Control** from the shortcut menu. You also can access this menu by clicking the status bar at the bottom of the front panel window. If no other client is currently in control, you have control of the front panel. If another client is currently controlling the VI, the server queues your request until the other client relinquishes control or until the control time limit times out. Only the user at the server computer can monitor the client queue list by selecting **Tools»Remote Panel Connection Manager**. If you want to save the data generated by a VI running on a remote computer, use **TCP** instead of remote front panels.

All VIs you want clients to view and control must be in memory on the server computer. If the requested VI is in memory, the server sends the front panel data of the VI to the requesting client. If the VI is not in memory, the **Connection status** section of the Connect to Remote Panel dialog box displays an error message.



Note (Linux) NI does not support remote front panels on Linux.

Multiple Clients Simultaneously Controlling Remote Front Panels

Multiple clients can control a VI remotely at the same time. To allow simultaneous control of a VI, the VI must be **reentrant**. To make a VI reentrant, select **File»VI Properties**, select **Execution** from the **Category** list, and select either **Shared clone reentrant execution** or **Preallocated clone reentrant execution**. LabVIEW opens a **clone** of the reentrant VI for each client request for a remote front panel. You can use the Web Server:VI Access List property to programmatically limit access to clones already in memory for remote front panel connections.

Functionality Not Supported in Viewing and Controlling Remote Front Panels

The following list includes functionality not supported and recommendations to consider when viewing and controlling remote front panels.

- Avoid exporting VIs that have While Loops but no wait function. These VIs prevent background tasks from performing in a reasonable amount of time, making front panels unresponsive when viewed or controlled remotely.

- Some VIs might not work exactly the same way from a remote computer as they do when run locally. Embedded.NET and ActiveX controls do not display on a remote client because they draw and operate almost completely independently of LabVIEW. If a VI presents the standard file dialog box, the controller receives an error because you cannot browse a file system remotely. Also, the browse button of a path control is disabled in remote panels.
- Clients viewing a front panel remotely might see different behavior depending on whether the front panel they are connecting to is from a built application. Specifically, if the front panel is from a built application, any programmatic changes to the front panel made before the client connects to the front panel are not reflected on the client computer. For example, if a Property Node changes a caption on a control before a client connects to that front panel, the client will see the original caption of the control, not the changed caption.
- Only a controller can remotely view the front panel of a VI dynamically opened and run using the VI Server or the front panel of a subVI configured to display the front panel when called. Clients not controlling the VI cannot view the front panel.
- Block diagrams that achieve certain user interface effects by polling properties of a front panel control might experience decreases in performance when you control the VI from a remote computer. You can improve the performance of these VIs by using the Wait For Front Panel Activity function.
- LabVIEW cannot generate a Panel Close event for a VI being viewed or controlled remotely. If you are viewing or controlling a VI remotely, LabVIEW can generate events only in the Control class, that is not in the Application or VI class.
- If you are remotely viewing or controlling a VI which displays the front panel of a subVI when calling the subVI, avoid manually closing the front panel of the subVI. Otherwise, you might lose control of the VI on both the client computer and the server computer. If you must view the front panel of the subVI when called, configure LabVIEW to *close the subVI after it runs*.

Securing the Web Server with TLS/SSL (Windows)

You can enable Transport Layer Security (TLS), previously known as Secure Sockets Layer (SSL), on the *Web Server for remote front panels* and *the Application Web Server for Web services*. Use TLS/SSL encryption to create secure, encrypted connections when sending data between clients and the Web Server.



NOTE

TLS/SSL is available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Certificates in TLS/SSL

TLS/SSL uses X.509 certificates to establish secure connections between a client and a server.

Certificates are digital files that contain identifying information about the server, as well as a public key and a digital signature. Server name, expiration date, and locale are examples of identifying information. A public key is a string of characters used to generate the encryption, and the digital signature is a confirmation of the authenticity of the certificate.

The digital signature can be added by the certificate creator—a [self-signed certificate](#)—or added by a [certificate authority \(CA\)](#), a trusted third-party company that issues digital certifications.

When a client attempts to establish a secure connection to the server, the server provides this certificate. The client confirms the authenticity of certificate, often using a major Web browser. The Web browser automatically cross-checks the authenticity of the certificate against a root certificate list of known CAs and displays the appropriate prompts to accept or reject the certificate. If the client accepts the certificate, the client communicates back to the server with a unique encryption code based on the public key. The server then uses a private key that was created along with the certificate to decode the encryption. At this point, the client and server have established a secure, encrypted connection.



Caution The private key remains on the system running the Web Server and is not made public to either the client or CA. The private key is vital to maintaining the integrity of the encrypted connection and must remain on the system that generated the initial certificate. If the private key is compromised, discontinue use of the associated certificate.

Enabling TLS/SSL Security

[Enable TLS/SSL security for Web services](#) using the **Web Configuration** page for the [Application Web Server](#). You can [enable TLS/SSL security for remote front panels](#) in the Options dialog box. When Web clients connect to Web services or remote front panels with TLS/SSL enabled, you must use the `https://` protocol. For example, `https://localhost:443` connects to a Web Server with TLS/SSL set up on port 443.

Creating a Self-Signed TSL/SSL Certificate

Use self-signed certificates to begin using TLS/SSL immediately, without requiring a third-party CA to digitally sign your certificate. A self-signed certificate is signed by the creator of the certificate and is the quickest way to use TLS/SSL in LabVIEW. You also can use self-signed certificates to test a system before you obtain a certificate signed by a CA. Self-signed certificates might create additional security prompts when the client accesses the Web Server using a major Web browser.

You can use the [default LabVIEW self-signed certificate](#), which LabVIEW creates when you enable TLS/SSL.

LabVIEW Default Self-Signed Certificate

LabVIEW creates a default self-signed certificate when you enable TLS/SSL, without the need to create a new self-signed certificate or certificate signing request (CSR). If you do not select a custom certificate when you enable TLS/SSL, LabVIEW uses the default certificate.

The default self-signed certificate is valid for 10 years from the date of creation.

You can find the default self-signed certificate file in the (Windows 7) C : \ProgramData \National Instruments\certstore\server_certs\server_0.cer directory.

Using NI Web-based Configuration & Monitoring with TLS/SSL

In NI Web-based Configuration & Monitoring, use the **SSL Certificate Management** tab of the **Web Server Configuration** page to create and manage self-signed certificates and CSRs. Refer to the help included with NI Web-based Configuration & Monitoring for more information about the following procedures:

- Creating a Self-Signed Certificate
- Creating a Certificate Signing Request
- Creating a Certificate from a CSR

Creating a Certificate Signing Request

You can create a certificate signing request to obtain a certificate that is authorized with a digital signature from a CA, a trusted third-party company that digitally certifies the authenticity of certificates. A network administrator, such as a corporate IT department, might also issue digital signatures.

A CSR contains the identifying information and public key components of a certificate. You then send CSRs to a trusted CA, who adds a digital signature and returns a complete certificate.

Using a Certificate Authority

When you create a CSR, you must send the CSR to a trusted CA to digitally sign. A network administrator might have a preferred CA or might digitally sign CSRs itself. The CA creates a valid certificate for use with the web server.

Related Information

[Configuring Web Services Security](#)

Using LabVIEW with TCP, UDP, and Other Low-Level Communications Applications

LabVIEW supports several low-level protocols you can use to communicate between computers.

Each protocol is different, especially in the way it refers to the network location of a remote application. Each protocol generally is incompatible with other protocols. For example, if

you want to communicate between macOS and Windows, you must use a protocol that works on both platforms, such as TCP.

TCP and UDP

Transmission Control Protocol (TCP) and User Datagram Protocol [UDP](#) are available on all platforms LabVIEW supports. TCP is a reliable, connection-based protocol. It provides error detection and ensures that data arrive in order and without duplication. For these reasons, TCP is usually the best choice for network applications.

Although UDP can give higher performance than TCP and does not require a connection, it does not guarantee delivery. Typically, use UDP in applications in which guaranteed delivery is not critical. For example, an application might transmit data to a destination frequently enough that a few lost segments of data are not problematic.

Use the UDP Multicast Open VI instead of the UDP Open function to open connections capable of reading, writing, or reading and writing UDP data to or from a multicast IP address. A multicast IP address defines a multicast group. Multicast IP addresses are in the 224 . 0 . 0 . 0 to 239 . 255 . 255 . 255 range. When a client wants to join a multicast group, it subscribes to the multicast IP address of the group. After the client subscribes to a multicast group, the client receives data sent to the multicast IP address.

Running AppleScript Code (macOS)

AppleScript is a scripting language that enables you to control macOS applications as well as different parts of the OS. Use the Run AppleScript Code VI to request actions or return information from macOS applications external to LabVIEW using AppleScript code.

Pipe VIs (Linux)

Use the Pipes VIs to open, close, read, and write to Linux named pipes. Use named pipes to communicate between LabVIEW and unrelated processes.

Executing System-Level Commands (Windows and Linux)

Use the System Exec VI to execute or launch other Windows-based applications or Linux command-line applications from within VIs. With the System Exec VI, you execute a system-level command line that can include any parameters supported by the application you want to launch.



[Data Communication Methods Home](#)

Using LabVIEW with TCP/IP and UDP

Internet Protocol [IP](#), User Datagram Protocol [UDP](#), and Transmission Control Protocol [TCP](#) are basic tools for network communication. The name TCP/IP comes from two of the best-

known protocols of the Internet protocol suite, the Transmission Control Protocol and the Internet Protocol.

You can use TCP/IP to communicate over single networks or interconnected networks. The individual networks can be separated by large geographical distances. TCP/IP routes data from one network or Internet-connected computer to another. Because TCP/IP is available on most computers, it can transfer information among diverse systems.

LabVIEW and TCP/IP

You can use the TCP/IP protocols with LabVIEW on all platforms. LabVIEW includes TCP and UDP VIs and functions you can use to create client or server VIs.

IP

IP performs the low-level service of moving data between computers. IP packages data into components called datagrams. A datagram contains the data and a header that indicates the source and destination addresses. IP determines the correct path for the datagram to take across the network or Internet and sends the data to the specified destination.

IP cannot guarantee delivery. In fact, IP might deliver a single datagram more than once if the datagram is duplicated in transmission. Programs rarely use IP but use TCP or UDP instead.

UDP

UDP provides simple, low-level communication among processes on computers. Processes communicate by sending datagrams to a destination computer or port. A port is the location where you send data. IP handles the computer-to-computer delivery. After the datagram reaches the destination computer, UDP moves the datagram to its destination port. If the destination port is not open, UDP discards the datagram. UDP shares the same delivery problems of IP.

Use UDP in applications in which reliability is not critical. For example, an application might transmit informative data to a destination frequently enough that a few lost segments of data are not problematic.

Using UDP in LabVIEW

Because UDP is not a connection-based protocol such as TCP, you do not need to establish a connection with a destination before you send or receive data. Instead, you specify the destination for the data when you send each datagram. Operating systems do not report transmission errors.

Use the UDP Open function to open a UDP socket on a port. The number of simultaneously open UDP ports depends on the operating system. The UDP Open function returns a network connection refnum that uniquely identifies the UDP socket. Use this connection refnum to refer to this socket in subsequent VI calls.

Use the UDP Write function to send data to a destination, and use the UDP Read function to read that data. Each write operation requires a destination address and port. Each read operation contains the source address and port. UDP preserves the datagram bytes that you specified for each command you send.

The size of UDP datagrams varies depending on the operating environment but has a maximum of 65535 bytes. National Instruments recommends using UDP to send datagrams no greater than 1 KB to minimize the chance that the underlying IP packet will need to be split and reassembled in transmission, which greatly increases the chance of packet loss.

When you finish all communications on a port, use the UDP Close function to free system resources.

UDP Multicast

You can use the UDP functions to communicate to a single client (single-cast) or to all computers on the subnet through a broadcast. If you want to communicate to multiple specific computers, you must configure the UDP functions to iterate through a list of clients. Using this technique creates duplicate network traffic because LabVIEW sends a separate copy of the data to each client and maintains a list of clients interested in receiving the data.

Use multicasting to communicate between a single sender and multiple clients on a network without requiring the sender to maintain a list of clients or send multiple copies of the data to each client. To receive data broadcast by a multicast sender, all clients must join a multicast group. The sender does not have to join a group to send data. The sender specifies a multicast IP address, which defines a multicast group. Multicast IP addresses are in the 224 . 0 . 0 . 0 to 239 . 255 . 255 . 255 range. When a client wants to join a multicast group, it subscribes to the multicast IP address of the group. After subscribing to a multicast group, the client receives data sent to the multicast IP address.

To multicast in LabVIEW, use the UDP Multicast Open VI to open connections capable of reading, writing, or reading and writing UDP data. Specify the time-to-live (TTL) for writing data, the multicast address for reading data, and the multicast port number for reading and writing data. The default TTL is 1, which means LabVIEW sends the datagram only to the local subnet. When a router receives a multicast datagram, it decrements the datagram TTL. If the TTL is greater than 1, the router forwards the datagram to other routers. The following table lists what action occurs to a multicast datagram when you specify a value for the **time-to-live** parameter.

0	Datagram remains on the host computer.
1	Datagram sent to every client on the same local subnet that subscribes to that IP address. Hubs/repeaters and bridges/switches forward the datagram. Routers do not forward the datagram if the TTL is 1.
>1	Datagram is sent and routers forward it through TTL-1 layers.

Refer to the following VIs for examples of using UDP multicasting:

- UDP Multicast – Receiver VI: `labview\examples\Data Communication\Protocols\UDP\UDP Multicast\UDP Multicast.lvproj`
- UDP Multicast – Sender VI: `labview\examples\Data Communication\Protocols\UDP\UDP Multicast\UDP Multicast.lvproj`

TCP

TCP ensures reliable transmission across networks, delivering data in sequence without errors, loss, or duplication. TCP retransmits the datagram until it receives an acknowledgment.

Using TCP in LabVIEW

TCP is a connection-based protocol, which means that sites must establish a connection before transferring data. The data transmission occurs between a *client* and a *server*. TCP permits multiple, simultaneous connections.



NOTE

You can create up to 1024 TCP connections on a single port.

You initiate a connection by waiting for an incoming connection or by actively seeking a connection with a specified address. In establishing TCP connections, you have to specify the address and a port at that address. Different ports at a given address identify different services at that address.

Use the TCP Open Connection function to actively establish a connection with a specific address and port. If the connection is successful, the function returns a network connection refnum that uniquely identifies that connection. Use this connection refnum to refer to the connection in subsequent VI calls.

You can use the following techniques to wait for an incoming connection:

- Use the TCP Listen VI to create a listener and wait for an accepted TCP connection at a specified port. If the connection is successful, the VI returns a connection refnum, the address, and the port of the remote TCP client.
- Use the TCP Create Listener function to create a listener and use the TCP Wait on Listener function to listen for and accept new connections. The TCP Wait On Listener function returns the same listener ID you wired to the function. When you finish waiting for new connections, use the TCP Close Connection function to close a listener. You cannot read from or write to a listener.

The advantage of using the second technique is that you can use the TCP Close Connection function to cancel a listen operation, which is useful when you want to listen for a connection without using a timeout, but you want to cancel the listen when another condition becomes true. You can close the TCP Listen VI at any time.

When you establish a connection, use the TCP Read function and the TCP Write function to read and write data to the remote application.

Use the TCP Close Connection function to close the connection to the remote application. If unread data remains and the connection closes, you might lose data. Use a higher level protocol for your computer to determine when to close the connection.

Deciding between TCP and UDP

TCP is the best protocol to use if you want reliable data transmission. UDP is a connectionless protocol with higher performance, but it does not ensure reliable data transmission.

Timeouts and Errors

When you design a network application, consider carefully what should happen if something fails. For example, if the server crashes, determine how each client VI handles it.

One solution is to make sure that each client VI has a timeout. If something fails to produce results after a certain amount of time, the client continues execution. In continuing, the client can try to reestablish the connection or report the error. If necessary, the client VI can shut down the application.

TCP and UDP Examples

Refer to the following VIs for examples of using the TCP and UDP VIs and functions:

- `labview\examples\Data Communication\Protocols\TCP\Simple TCP\Simple TCP.lvproj`
- `labview\examples\Data Communication\Protocols\UDP\Simple UDP\Simple UDP.lvproj`



[Data Communication Methods Home](#)

OPC Systems With LabVIEW

Connecting to OPC Systems Using LabVIEW (Windows Only)

OLE for Process Control (OPC) is a set of standard interfaces developed to ensure interoperability among control devices from different manufacturers for the transfer of real-time plant data. The OPC Foundation—comprised of representatives from manufacturers of control systems, instrumentation systems, and process control systems—oversees the development and maintenance of the OPC specifications.

OPC Specifications

The OPC Foundation developed the OPC specifications so you can use devices from more than one manufacturer and interchange devices throughout a plant without having to change the code that communicates with those devices. These specifications also help you control processes and collect data in the plant and then transfer and use that data enterprise-wide.

The OPC Foundation currently maintains eight OPC specifications:

- OPC Data Access (DA)
- OPC Alarms and Events
- OPC Batch
- OPC Data eXchange
- OPC Historical Data Access
- OPC Security
- OPC XML-DA
- OPC Unified Architecture (UA)



NOTE

LabVIEW supports the OPC Data Access specification. You can [connect to the LabVIEW OPC server from third-party OPC clients](#). You also can [develop OPC clients in LabVIEW](#) to connect to other OPC servers.



Note The LabVIEW OPC UA Toolkit supports the OPC UA specification. You can use the OPC UA VIs to create OPC UA server or OPC UA client applications.

The OPC Foundation has released three major versions of the OPC specification. LabVIEW currently supports the following versions of the OPC DA Specification:

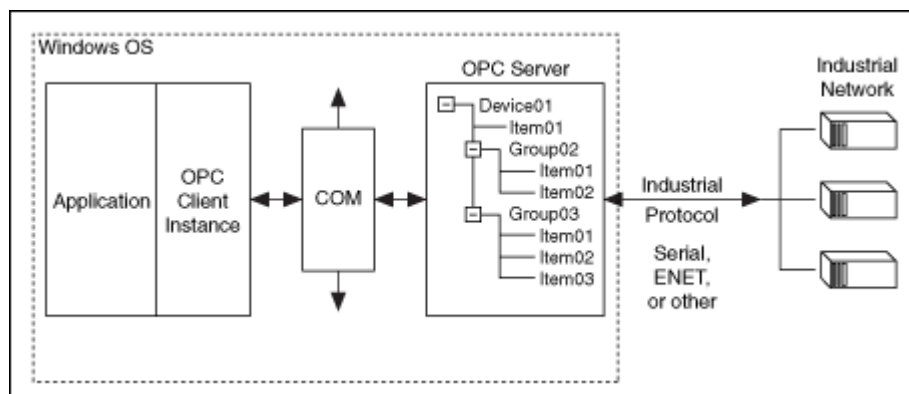
- Version 1.0—DataSocket Client.
- Version 2. x —DataSocket Client, Variable Engine OPC Server, (DSC Module) DSC OPC Client.
- Version 3.0—Variable Engine OPC Server, (DSC Module) DSC OPC Client.

Refer to the OPC Foundation website for more information about the OPC specifications.

Local OPC Systems

For a local OPC system, you install both the OPC server and the OPC client on the same computer running Windows.

As shown in the following illustration, a plant contains an industrial network of control devices that communicates with the OPC server using an industrial protocol such as serial or ENET.



When the OPC server receives data from the industrial network, the OPC server translates the data into a standard OPC format and makes the data accessible to OPC clients. The OPC clients can read the data or write new data to the devices through the OPC server. The OPC server organizes data into groups and items and exposes the data through a standard interface. Groups allow the OPC client to organize data and make group-level requests for update notifications. Items provide information about the data that the OPC server publishes.

The OPC server implements several interfaces that depend on COM, a Windows-based technology, to exchange data with OPC servers. This is the reason OPC is available only on Windows platforms. When an OPC client connects to an OPC server, the client specifies which data items to subscribe to and how often to receive updates.

Remote OPC Systems

For a remote OPC system, you install the OPC server and the OPC client on different computers. Both computers must run Windows, and you must connect the computers using Ethernet.

You must properly configure DCOM settings on the OPC server to make an OPC server/client system operate properly. DCOM restricts the access rights to specific computers on a domain, and security preferences can prevent OPC clients from communicating with remote OPC servers. Therefore, local OPC is often easier to configure than remote OPC.

You also must configure many Windows settings, such as security, firewalls, and the OPC server and OPC client software. Refer to the Windows Help and Support Center for information about configuring security, DCOM, and firewalls on Windows. Refer to the documentation for the OPC server and OPC client software for information about configuring the OPC server and client.

Configure the settings in the following order for best results:

- Windows security, including users, groups, and respective access rights
- DCOM

- firewalls
- OPC server
- OPC client

Connecting to the LabVIEW OPC Server from Third-Party OPC Clients (Windows Only)

LabVIEW 8.0 or later contains an OPC server called the *Shared Variable Engine* (SVE). The SVE supports *OPC Data Access 2. x* and *OPC Data Access 3.0*. You can publish data from the SVE using *shared variables*.

To connect to the SVE from a third-party OPC client, use the ProgID **National Instruments Variable Engine**. If the OPC client allows you to browse for OPC servers, you can locate the **National Instruments Variable Engine** under OPC version 2. x or 3.0, depending on which versions the client supports.

In the third-party OPC client, you can browse all data items published by the SVE. The SVE allows connections to scalars and arrays of scalars but does not recognize data items of complex data types, such as clusters. You can browse FieldPoint and DAQ data items under their device categories.

Developing OPC Clients in LabVIEW (Windows Only)

You have two options for developing OPC clients in LabVIEW that allow you to connect to *OPC systems*: LabVIEW DataSocket, which is available in the LabVIEW Development System, or the LabVIEW Datalogging and Supervisory Control (DSC) Module.

LabVIEW DataSocket

You can use DataSocket to *open a connection to a data item* published on an OPC server using the opc protocol. This option works well for applications with the following conditions:

- You do not need to log data automatically or add alarming, scaling, and security.
- You have a small application with references to only one or two OPC data items.
- You do not need to configure settings such as the update rate or deadband.
- You need to connect to an OPC server that supports OPC version 1.0 or 2. x. If you need to use OPC version 3.0, you must use the DSC Module.



NOTE

LabVIEW 64-bit does not support using DataSocket connections to an OPC server.

LabVIEW DSC Module

The DSC Module extends the LabVIEW graphical development environment with additional functionality for the rapid development of distributed measurement, control, and high-channel count monitoring applications. The DSC Module also extends the data services portion of the LabVIEW shared variable, which you can bind to data items published on an OPC server.

The DSC Module is ideal for applications with the following conditions:

- You need to log data automatically or enable alarming, scaling, and security.
- You have many OPC data items. The DSC Module can locate all OPC servers installed on the computer or network and read available information about the server capabilities and data items directly from the OPC server. You then can browse data items and configure settings such as the update rate and deadband.
- You need to access OPC items from shared variables.
- You need to connect to an OPC server that supports OPC version 3.0.

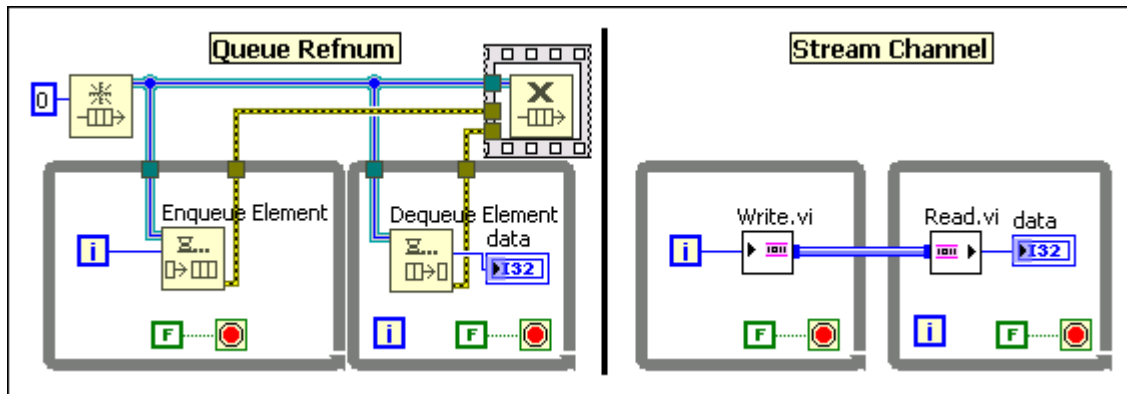
You can use the DSC Module as an OPC client to access OPC servers, including the NI OPC Servers that the DSC Module provides. Use the NI OPC Servers application, available by selecting **Start»All Programs»National Instruments»NI OPC Servers»NI OPC Servers**, to transfer data from industrial devices and systems to client applications on a host machine. The NI OPC Servers application enables data sharing between a variety of applications, such as human-machine interface software, manufacturing execution systems, and enterprise resource planning applications.

Refer to ni.com/labviewdsc for information about or to purchase the DSC Module.

Communicating Data between Parallel Sections of Code Using Channel Wires

A channel wire is a wire in LabVIEW that expresses an asynchronous communication between two parallel sections of code without forcing an execution order. Using a channel wire is similar to using a refnum or a variable that one loop writes into and the other loop reads from. Instead of using a refnum or a variable, you represent the data transfer graphically as a wire.

The following figure illustrates that you can use channel wires to build your block diagram to achieve the same functionality as using queue refnums.



Every channel wire must have at least two endpoints: a writer and a reader. Endpoints are places in the code that exchange data. Writer endpoints write data to the channel. Reader endpoints read data from the channel. By connecting the writer endpoint to the reader endpoint using a channel wire, you enable asynchronous data communication between parallel sections of code.

LabVIEW provides the following channel templates that you can use to build applications. Each template expresses a different communications protocol. You can choose from the following templates based on your communications and application needs:

LabVIEW	Real-Time Module	FPGA Module*
Stream Tag Messenger Accumulator Tag Event Messenger High Speed Stream Lossy Stream One Element Stream	Stream Tag Messenger Accumulator Tag Event Messenger High Speed Stream Lossy Stream One Element Stream Real Time Stream	Stream Tag Accumulator Tag Lossy Stream One Element Stream
*For each supported channel template, the FPGA Module supports only a subset of the endpoints.		

You can create a channel writer endpoint or a channel reader endpoint by right-clicking a terminal or a wire and selecting **Create»Channel Writer** or **Create»Channel Reader**. In the Select Channel Endpoint dialog box that appears, you can switch between writer or reader endpoints by clicking **Switch Read/Write**. When you create an endpoint from a terminal, you instantiate the channel template with that data type as its transmission type.

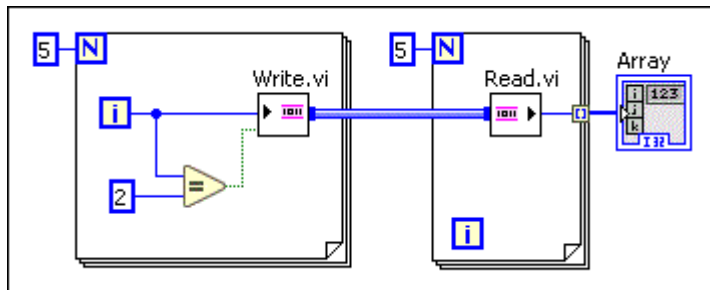
All Stream channels are not forkable, which means they connect a single writer endpoint to a single reader endpoint. All the other channels are forkable to allow multiple writers or multiple readers on the same channel.

Understanding the Lifetime of Channels

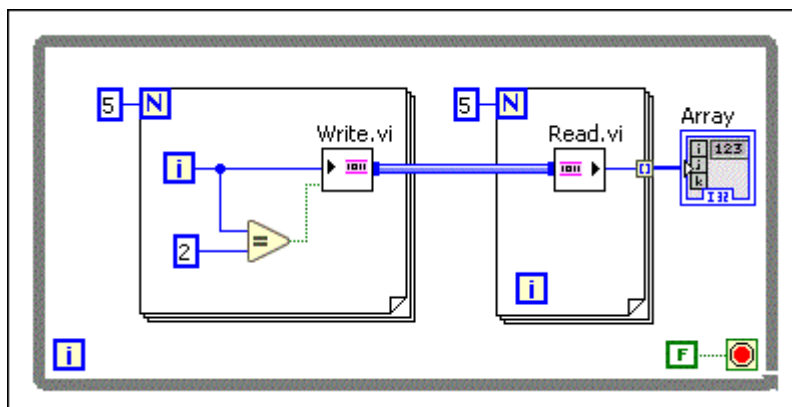
The calling block diagram of a channel governs the channel execution. A calling block diagram is the lowest level block diagram that contains all endpoints of a channel or all the nodes that call those endpoints.

A channel initializes when its calling block diagram begins running. A channel closes when the **last element?** or **abort** input on one of its endpoints is TRUE. If a channel closes, it ignores further writes to the channel. The channel reinitializes when its calling block diagram begins another run. A calling block diagram can be a structure node, the top-level block diagram of a VI, or a higher level block diagram that calls the current top-level VI.

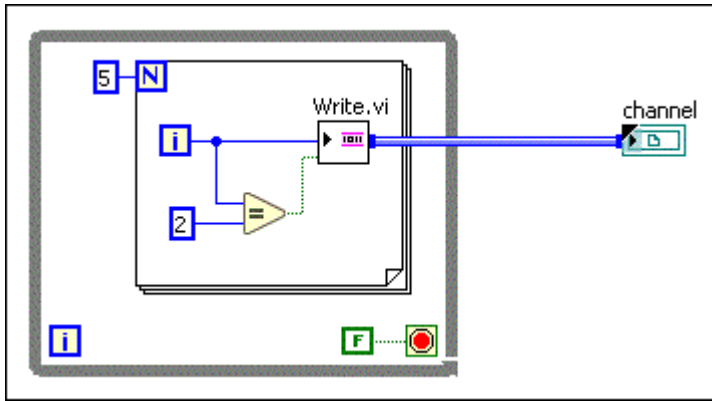
In the following figure, the calling block diagram of the channel is the top-level diagram of this VI. The channel initializes when the calling block diagram begins running, which means the channel initializes each time the top-level VI begins running or each time the top-level VI is called as a subVI.



In the following figure, the calling block diagram of the channel is the While Loop. The channel initializes when the While Loop begins running, which means the channel initializes on each iteration of the While Loop.



In the following figure, the calling block diagram of the channel is not the top-level VI but is higher than the current top-level block diagram. The channel initializes when the higher level block diagram begins running, which means the channel initializes each time the higher level block diagram begins running or each time the higher level block diagram is called as a subVI. The channel does not act upon the iterations of the While Loop to initialize because the While Loop does not contain all the endpoints of the channel.



In all the channel instances above, the Write endpoint writes the first three values to the channel but ignores the final two values because the **last element?** input is TRUE when the number of elements left unwritten in the array equals to 2.

Stopping a Replicate Endpoint

When you create a channel reader or writer, select **With Abort** to create an endpoint with a boolean input **abort**. When TRUE, **abort** alerts all endpoints to produce an **aborted?** output. Any endpoint can send the abort signal. If you have loops that include channel wires, wire all loops to the **aborted?** output of the endpoints to stop the loops when an endpoint sends the abort signal. No local is necessary for this mechanism. If you use a Replicate endpoint, the original channel must supply the abort value. The Replicate endpoint stops its internal loop and propagate the abort value to the replicated channels.

Advantages of Channel Wires Compared to Refnums and Variables

Using channel wires provides the following benefits:

- Channel wires do not require you to create and destroy refnums like queues.
- Channel wires require fewer nodes on the block diagram than queues.
- Channel wires show where the data flows to and from, which makes them more visible and easier to debug than refnums and variables.
- Channel wires can incorporate stop and abort signals with your data, which makes your application less error prone than creating such signals using refnums and variables.
- Channel wires allow you to create co-routines that fill the role of closures in other programming languages. Refer to the `labview\examples\Channels\Replacing The Function At The Heart Of An Algorithm At Run Time\Replacing The Function At The Heart Of An Algorithm At Run Time.lvproj` for an example of using channel wires to create co-routines.

Using LabVIEW with Wireless Devices

With IrDA technology, devices can communicate through a wireless infrared interface. In LabVIEW two VIs running on two separate computers can communicate using built-in IrDA capabilities. For example, you can build a VI that runs a test and collects the results on one desktop computer. You can then transmit that data via infrared beams to another computer.

Bluetooth technology provides flexible, low-power communication among Bluetooth devices over a radio frequency. In LabVIEW, VIs running on separate computers or on Mobile devices can use Bluetooth capabilities to communicate.

Overview of IrDA Client/Server Networks

An IrDA network is similar to an isolated TCP/IP network, where you can assign IP addresses at random as long as each address on the network is unique. Because an IrDA network is dynamic and devices can enter and leave the network frequently, there are no fixed IrDA addresses that a client can look up to establish communication with a server. When the network detects a computer, the network identifies each device by its name (usually specified by the user) and a dynamically generated unique 32-bit ID.

To establish communication between devices on a wireless network, the IrDA device acting as a server monitors the network for devices attempting to establish communication on the network. The server creates a listener, so it can listen for any device that enters the network, as opposed to opening a connection by specifying an address to determine if that device is connected to the network. The listener establishes a service, which is similar to opening a port in TCP, by accessing a free entry in a database on the server called the information access service (IAS) database. The database can include up to 128 entries. Each established service in the IAS database is assigned a Logical Service Access Point Selector (LSAP-SEL), a number ranging from 0 to 127, and a corresponding service ID, which is a string that identifies the service.

The client queries the database with the service ID to find the LSAP-SEL number. After the LSAP-SEL number is established, communication between the devices can begin.

For example, you could identify a service with the service ID temperature so when the server establishes a connection with a client, it sends a collection of temperatures to the client. The server then listens for a client requesting the service ID temperature. When the client connects to the network, it sends the service temperature to the server, which in turn establishes an LSAP-SEL number for the service. The client then queries the server for the LSAP-SEL number corresponding to the temperature service. After the LSAP-SEL number is established, the server sends the temperature data to the client.

Refer to the Infrared Data Association Web site for more information about IrDA technology.

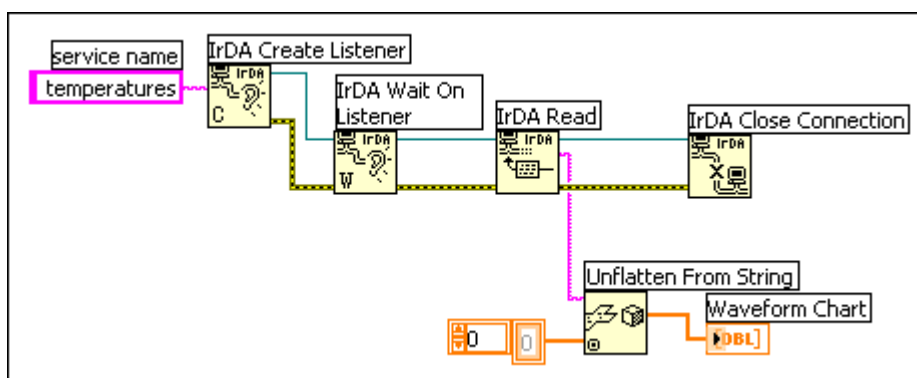
IrDA Communication Model in LabVIEW

After you install on your computer an infrared device that conforms to the IrDA standards and ensure that it is working properly, you can start building IrDA applications.

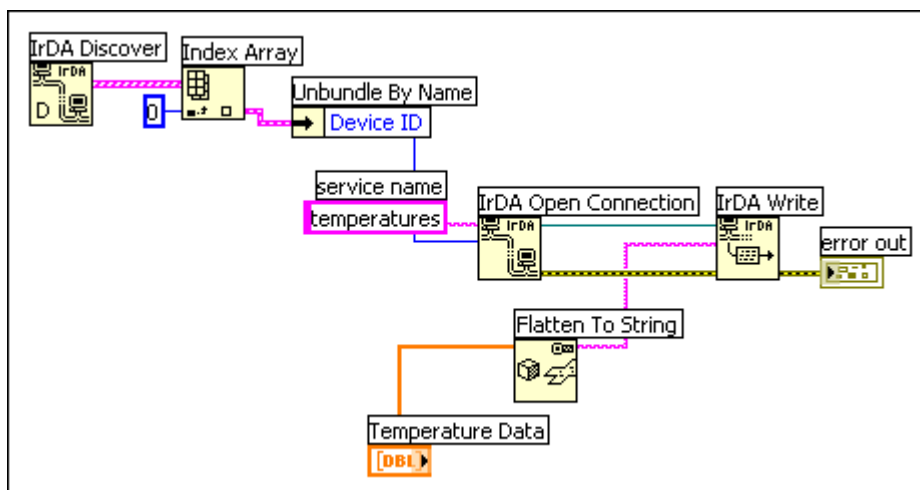
To establish a wireless communication link between VIs running on separate computers, create an IrDA server using the IrDA functions. The server establishes the line of communication between itself and the remote computer. The server listens for and discovers transmissions from the remote computer. The server transmits data to and receives data from the remote computer and closes the connection.

- Use the IrDA Create Listener function to establish a service. Use this function also to specify the service name that your VI and a VI on a remote computer use to establish a LSAP-SEL number.
- Use the IrDA Wait On Listener function to wait for the detection of a remote computer. When the infrared sensor attached to your computer detects another computer passing the service ID, the server establishes communication between itself and the client. You can communicate with only one computer at a time.
- Use the IrDA Write function to write data to a VI on a remote computer. The data you write must be a string. Use the Flatten to String function to convert any data that is not a string to the string format. You must then unflatten the string on the remote computer. You also can use the Flatten to XML function to convert data to the XML format.
- Use the IrDA Read function to read data from a VI on a remote computer. The remote computer sends the data as a string, even if the data is a different data type. Use the Unflatten from String function to convert the data to the correct data type. You also can use the Unflatten from XML function to [convert data from the XML format](#). You must know what kind of data you are receiving.
- Use the IrDA Close Connection function to close the service.

The VI in the following illustration creates a service called `temperatures`, listens for a remote computer requesting that service, reads the temperature data the device collects, unflattens the string to an array of numbers, plots that data to a chart, and closes the connection.



The VI in the following illustration discovers the device ID of the remote server, establishes a connection to the service `temperatures`, flattens an array of numbers to a string, and writes temperature data to the server.



Overview of Bluetooth Client/Server Networks

Bluetooth is a wireless technology that uses a radio frequency of 2.4 GHz to allow devices to communicate. The range of a Bluetooth connection is between 30 and 40 feet, depending on the device and environmental conditions.

The LabVIEW Bluetooth VIs and functions use RFCOMM, which is a connection protocol the Winsock interface exposes. RFCOMM is simple transfer protocol that emulates serial communication. The RFCOMM interface defines Bluetooth servers and clients.

Creating Bluetooth server and client applications in LabVIEW is similar to creating server and client applications for TCP communication. A Bluetooth server uses the Service Discovery Protocol (SDP) to broadcast the availability of the services the server contains and listens for inbound connections. A client creates an outbound RFCOMM connection to a server. Once the client and server connect to each other, they exchange data until the client or server terminates the connection or until the connection is lost.

LabVIEW supports Bluetooth devices that use the Microsoft Bluetooth driver. Refer to the Microsoft website for Bluetooth devices that support the Microsoft Bluetooth driver. Most Bluetooth devices use a proprietary Bluetooth driver by default. To use the device with LabVIEW, you must switch to the Microsoft Bluetooth driver.

Refer to the Bluetooth and Microsoft websites for more information about Bluetooth technology.

Connecting Dynamically with Call By Reference Nodes

Channel wires not only express static connections between parallel sections of code, they also express dynamic connections when working with Call By Reference nodes. The following illustration demonstrates dynamically connecting channel wires with Call By Reference nodes.



The channel wire that connects the writer endpoint and the Call By Reference node and the channel wire that connects the Call By Reference node and the reader endpoint have different colors. This is because channel wires display the color of the transmission data type, not the color of the data pattern.

Cycles occur if a data flow originates from an output of a node and terminates as an input of the same node. If you use wires other than channel wires, such cycles are illegal and break the VI unless you add a Feedback Node that provides the cycle with a starting point. With channel wires, you can create cycles without adding a Feedback Node because channel wires do not create dataflow dependencies among endpoints. Every endpoint acts as a starting point.

The following list describes important details about the previous diagram:



NOTE

You also can remove the dataflow dependency by introducing a timeout on one of the endpoints, or using a template channel that does not wait, such as a Tag channel.

NI Service Locator

The NI Service Locator stores registered services and returns data to a client who requests a service. The client can use a service name to query the NI Service Locator and obtain the port number associated with that service name. Thus, the client does not need to know the port number of the server application. The NI Service Locator runs as a background service and requires no user interaction.

Registering Services with the NI Service Locator

You can use the **service name** input of the TCP Create Listener function and the UDP Open function to register a service name and a port number with the NI Service Locator. You also can use the VI Server page of the Options dialog box to set the port number or service name. Use the Server:Port and Server:Service Name properties to set the port number or service name programmatically.

Refer to the following VIs for examples of using functions to register a service name:

- `labview\examples\Data Communication\Protocols\TCP\Simple TCP\Simple TCP.lvproj`

Querying the NI Service Locator to Find Registered Services

After you register a service with the NI Service Locator, you can use the Open Application Reference function, the TCP Open Connection function, and the UDP Write function to query the NI Service Locator for the port number associated with a service name you specify.

Refer to the KnowledgeBase for more information about the NI Service Locator.

Multitasking, Multithreading, and Multiprocessing

Multitasking refers to the ability of the operating system to quickly switch between tasks, giving the appearance of simultaneous execution of those tasks. For example, on Windows 3.1, a task is generally an entire application, such as Microsoft Word, Microsoft Excel, or LabVIEW. Each application runs for a small time slice before yielding to the next application. Windows 3.1 uses a technique known as cooperative multitasking, where the operating system relies on running applications to yield control of the processor to the operating system at regular intervals. Windows relies on preemptive multitasking, where the operating system can take control of the processor at any instant, regardless of the

state of the application currently running. Preemptive multitasking guarantees better response to the user and higher data throughput.

Multithreading extends the idea of multitasking into applications, so that specific operations within a single application can be subdivided into individual threads, each of which can theoretically run in parallel. Then, the operating system can divide processing time not only among different applications, but also among each thread within an application. For example, in a LabVIEW multithreaded program, the application might be divided into three threads — a user interface thread, a data acquisition thread, and an instrument control thread — each of which can be assigned a priority and operate independently. Thus, multithreaded applications can have multiple tasks progressing in parallel along with other applications.

Multiprocessing, or multi-core programming, refers to two or more processors in one computer, each of which can simultaneously run a separate thread. Thus, a single-threaded application can run on only one processor at a time. A multithreaded application can have multiple separate threads executing simultaneously on multiple processors. In the LabVIEW multithreaded example, the data acquisition thread could run on one processor while the user interface thread runs on another processor. Single-threaded applications can have a serious effect on system performance.



NOTE

You can use the LabVIEW Desktop Execution Trace Toolkit to monitor thread usage, memory leaks, and other aspects of LabVIEW programming.

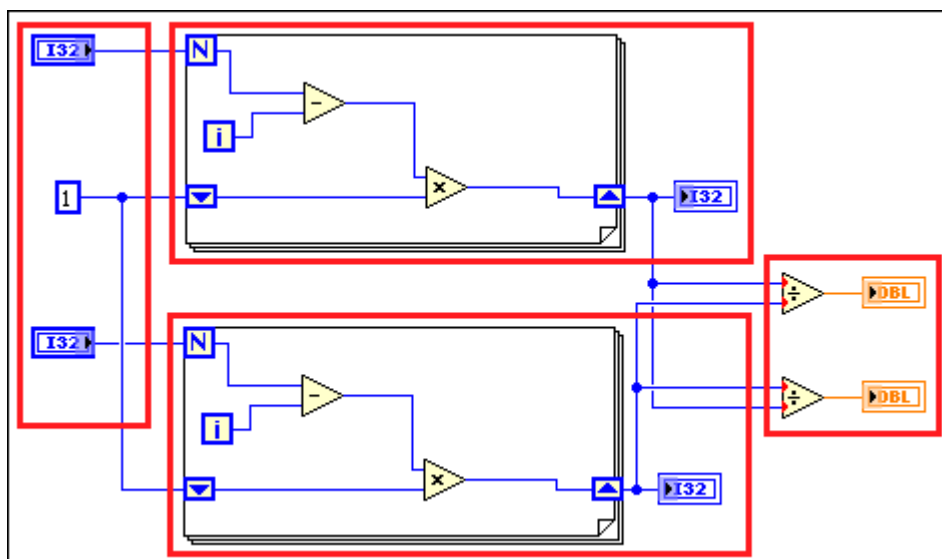
LabVIEW Threading Model

All the complex tasks of thread management are transparently built into the LabVIEW execution system. Text-based programmers must learn new, complex programming practices to create a multithreaded application. However, all LabVIEW applications are automatically multithreaded without any code modifications.

In addition to the preemptive multitasking of the operating system, LabVIEW employs a cooperative multitasking system. During compilation, LabVIEW analyzes VIs to find groups of nodes that can execute together in what are called clumps. Each priority and execution system combination has a run queue data structure that retains which clumps can run together. When the execution system activates a thread, the execution system retrieves and executes a clump from the run queue. When the execution system finishes executing, it stores additional clumps that meet the input conditions on the run queue, which allows the block diagram to execute in any of the available execution threads. If the block diagram includes enough parallelism, it can simultaneously execute in all threads.

LabVIEW does not permanently assign clumps of code to a particular thread. LabVIEW can execute a clump using a different thread the next time you run the VI.

The red rectangles in the following block diagram show an example of how LabVIEW groups nodes in clumps.



In this example, the block diagram has three different clumps because the clump of controls is the same as the clump of indicators. As a result, this clump executes at two different times. A clump sleeps, or delays, before and after it executes. After the first clump on the left executes, it sleeps while the two clumps with the For Loops execute. The right and left clump stops sleeping when the VI is ready to execute the operations in the clump of indicators.

Multitasking in LabVIEW

LabVIEW uses preemptive and cooperative multithreading. Operating systems and processors with preemptive multithreading use a limited number of threads, so in certain cases, these systems return to using cooperative multithreading.

The execution system preemptively multitasks VIs using threads. However, a limited number of threads are available. For highly parallel applications, the execution system uses cooperative multitasking when available threads are busy. Also, the operating system handles preemptive multitasking between the application and other tasks.

Basic Execution System

The basic execution system maintains a queue of active tasks. For example, if you have three loops running in parallel, at any given time one task is running and the other two are waiting in the queue. Assuming all tasks have the same priority, one of the tasks runs for a certain amount of time. That task moves to the end of the queue, and the next task runs. When a task completes, the execution system removes it from the queue.

The execution system runs the first element of the queue by calling the generated code of the VI. At some point, the generated code of that VI checks with the execution system to see if the execution system assigns another task to run. If not, the code for the VI continues to run.

Synchronous/Blocking Nodes

A few nodes or items on the block diagram are synchronous, meaning they do not multitask with other nodes. In a multithreaded application, they run to completion, and the thread in which they run is monopolized by that task until the task completes.

Property Nodes and Invoke Nodes used on control or application references, even for the Value property on a control reference, synchronize with the UI thread. Therefore, if the UI thread is busy, such as when displaying a large amount of data in a graph, the Property Node or Invoke Node will not execute until the UI thread completes its current work.

Shared library calls and computation functions run synchronously. Most analysis VIs and data acquisition VIs contain shared library calls and therefore run synchronously. Configure a single, non-reentrant VI to call the shared library to ensure that only one thread calls a non-reentrant shared library at a time. If the shared library is called from more than one non-reentrant VI, LabVIEW may return unexpected results.

Almost all other nodes are asynchronous. For example, structures, I/O functions, timing functions, and subVIs run asynchronously.

The Wait on Occurrence, Timing, GPIB, VISA, and Serial VIs and functions wait for the task to complete but can do so without holding up the thread. The execution system takes these tasks off the queue until their task is complete. When the task completes, the execution system puts it at the end of the queue. For example, when the user clicks a button on a dialog box the Three Button Dialog VI displays, the execution system puts the task at the end of the queue.

Deadlock

Deadlock occurs when two or more tasks are unable to complete because they are competing for the same system resource(s). An example of deadlock can be two applications that need to print a file. The first application, running on thread 1, acquires the file and locks it from other applications. The second application, in thread 2, does the same with the printer. In a non-preemptive environment, where the operating system does not intervene and free a resource, both applications wait for the other to release a resource, but neither releases the resource it already holds. One way to avoid deadlock is to configure VIs to access common resources in the same order.

Managing the User Interface in the Execution System

In addition to running VIs, the execution system must coordinate interaction with the user interface. When you click a button, move a window, or change the value of a slide control, the execution system manages that activity and makes sure that the VI continues to run in the background.

The single-threaded execution system multitasks by switching back and forth between responding to user interaction and running VIs. The execution system checks to see if any user interface events require handling. If not, the execution system returns to the VI or accepts the next task off the queue.

When you click buttons or pull-down menus, the action you perform might take a while to complete because LabVIEW runs VIs in the background. LabVIEW switches back and forth between responding to your interaction with the control or menu and running VIs.

Execution Systems in LabVIEW

LabVIEW has six multiple execution systems that you can assign by selecting **File»VI Properties** and selecting **Execution** in the VI Properties dialog box. You can select from the following execution systems:

- **user interface** —Handles the user interface. Behaves exactly the same in multithreaded applications as in single-threaded applications. VIs can run in the user interface thread, but the execution system alternates between cooperatively multitasking and responding to user interface events.
- **standard** —Runs in separate threads from the user interface.
- **instrument I/O** —Prevents VISA, GPIB, and serial I/O from interfering with other VIs.
- **data acquisition** —Prevents data acquisition from interfering with other VIs.
- **other 1** and **other 2** —Available if tasks in the application require their own thread.
- **same as caller** —For subVIs, run in the same execution system as the VI that called the subVI. For top-level VIs, **same as caller** has the same effect as setting the preferred execution system to **standard**.

These execution systems provide some rough partitions for VIs that must run independently from other VIs. By default, VIs run in the Standard execution system.

The names Instrument I/O and Data Acquisition are suggestions for the type of tasks to place within these execution systems. I/O and data acquisition work in other systems, but you can use these labels to partition the application and understand the organization.

Execution systems are not responsible for managing the user interface. If a VI in one of these execution systems needs to update a control, the execution system passes responsibility to the user interface execution system. Assign the user interface execution system to VIs that contain a large amount of Property Nodes.

LabVIEW allocates a number of threads proportional to the number of processors on a machine. Each thread handles a task. For example, if a VI calls a shared library, the separate thread continues to run other VIs within that execution system. Because each execution system has a limited number of threads, tasks remain pending if the threads are busy.

Although VIs you write run correctly in the standard execution system, consider using another execution system. For example, if you are developing instrument drivers, you might want to use the instrument I/O execution system.

Even if you use the standard execution system, the user interface is still separated into its own thread. Any activities conducted in the user interface, such as drawing on the front panel, responding to mouse clicks, and so on, take place without interfering with the execution time of the block diagram code. Likewise, executing a long computational routine does not prevent the user interface from responding to mouse clicks or keyboard data entry.

Computers with multiple processors benefit even more from multithreading. On a single-processor computer, the operating system preempts the threads and distributes time to each thread on the processor. On a multiprocessor computer, threads can run simultaneously on the multiple processors so more than one activity can occur at the same time.

Benefits of Multithreaded Applications

Applications that take advantage of multithreading have numerous benefits, including the following:

- More efficient CPU use
- Better system reliability
- Improved performance on multiprocessor computers

More Efficient CPU Use

In many LabVIEW applications, you make synchronous acquisition calls to an instrument or data acquisition device. Such calls often take some time to complete. In a single-threaded application, a synchronous call such as this effectively blocks, or prevents, any other task within the LabVIEW application from executing until the acquisition completes. In LabVIEW, multithreading prevents this blocking. While the synchronous acquisition call runs on one thread, other parts of the program that do not depend on the acquisition, such as data analysis and file I/O, run on different threads. Thus, execution of the application progresses during the acquisition instead of stalling until the acquisition completes.

In this way, a multithreaded application maximizes the efficiency of the processor because the processor does not sit idle if any thread of the application is ready to run. Any program that reads and writes from a file, performs I/O, or polls the user interface for activity can benefit from multithreading simply because you can use the CPU more efficiently during these synchronous activities.

Better System Reliability

By separating the program onto different execution threads, you can prevent other operations in your program from adversely affecting important operations. The most common example is the effect the user interface can have on more time-critical operations. Many times, screen updates or responses to user events can decrease the execution speed of the program. For example, when someone moves a window, resizes it, or opens another window, the program execution effectively stops while the processor responds to that user interface event.

In a LabVIEW multithreaded application, user interface operations are separated onto a dedicated user interface thread, and the data acquisition, analysis, and file I/O portions of the program can run on different threads. By giving the user interface thread a lower priority than other more time-critical operations, you can ensure that the user interface operations do not prevent the CPU from executing more important operations, such as collecting data from a computer-based instrument. Giving the user interface lower priority improves the overall system reliability, including data acquisition and processing, the

performance of LabVIEW, and the performance of the computer as a whole, by ensuring that data is not lost because an operator moves a window, for example.

Another example where multithreading provides better system reliability is when you perform high-speed data acquisition and display the results. Screen updates are often slow relative to other operations, such as continuous high-speed data acquisition. If you attempt to acquire large amounts of data at high speed in a single-threaded application and display all that data in a graph, the data buffer may overflow because the processor is forced to spend too much time on the screen update. When the data buffer overflows, you lose data.

However, in a LabVIEW multithreaded application with the user interface separated on its own thread, the data acquisition task can reside on a different, higher priority thread. In this scenario, the data acquisition and display run independently so the acquisition can run continuously and send data into memory without interruption. The display runs as fast as it can, drawing whatever data it finds in memory at execution time. The acquisition thread preempts the display thread so you do not lose data when the screen updates.

Improved Performance on Multiprocessor Computers

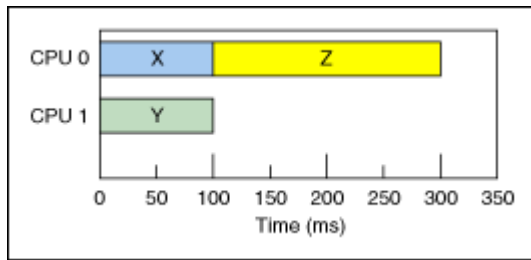
One of the most promising benefits of multithreading is that it can harness the power of multiprocessor computers. Many high-end computers today offer two or more processors for additional computation power. Multithreaded applications are poised to take maximum advantage of those computers. In a multithreaded application where several threads are ready to run simultaneously, each processor can run a different thread. In a multiprocessor computer, the application can attain true parallel task execution, thus increasing overall system performance.

In contrast, single-threaded applications can run on only a single processor, thus preventing them from taking advantage of the multiple processors to improve performance. Therefore, to achieve maximum performance from multithreaded operating systems and/or multiprocessor machines, an application must be multithreaded.

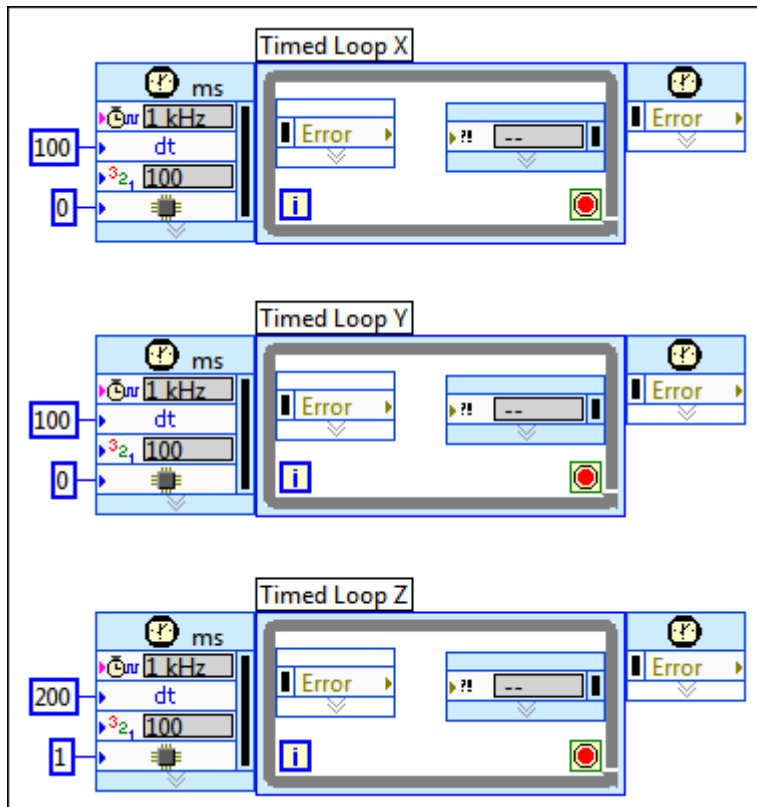
Manually Assigning CPUs

LabVIEW makes it easy to take advantage of multiple CPU systems, also known as multi-core, multiprocessor, or SMP systems, using parallelism or [pipelining](#). In certain cases, you might be able to achieve an even more efficient CPU utilization by [manually assigning](#) particular threads to particular processors.

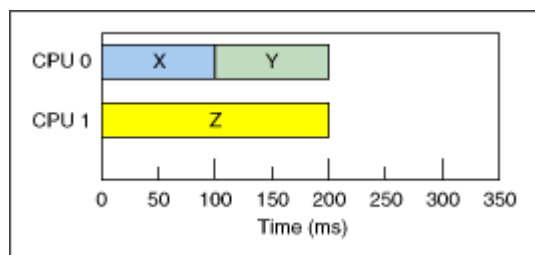
(Real-Time, Windows) You can use the Timed Loop to manually control CPU allocation. For example, consider an application with parallel Timed Loop X, Y, and Z running on a system with two CPUs. The Timed Loops take 100 ms, 100 ms, and 200 ms, respectively, to execute. If the Timed Loops are all set to the default priority, the CPU scheduler might assign Timed Loop X and Z to run on one CPU and Timed Loop Y to run on the other CPU, resulting in a total execution time of 300 ms, as shown in the following illustration.



You could optimize the VI by manually assigning Timed Loop X and Y to one CPU and Timed Loop Z to the other CPU, as shown in the following block diagram.



In this case, manual CPU assignment reduces the total execution time to 200 ms, as shown in the following illustration.

**NOTE**

The Real-Time Module executes system threads on the default processor (CPU 0). You can reduce jitter by assigning deterministic Timed Loops to a dedicated CPU other than the default CPU. You also can use automatic load balancing on one set of CPUs while reserving another set for manual CPU assignment. The Real-Time Module CPU scheduler handles timed structure threads separately from system threads, and by default performs automatic load balancing over all CPUs for both types of threads. Refer to the National Instruments Web site for more information about specifying the set of CPUs available for automatic load balancing for each type of thread.

Pipelining for Systems with Multiple CPUs

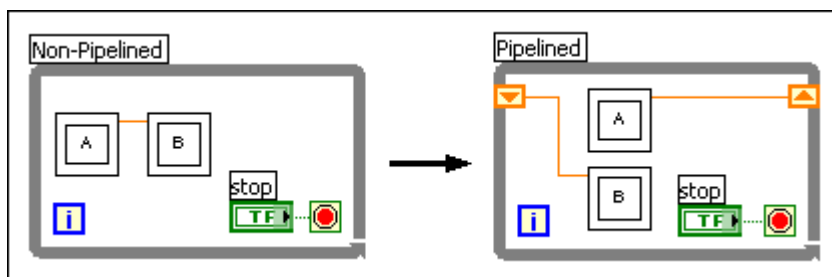
The primary advantage of a multiple-CPU system, also known as multi-core, multiprocessor, or SMP system, is that multiple threads can execute in parallel. Therefore, it is difficult to take advantage of a multiple processor system when an application consists mainly of a single sequential process. However, you can take advantage of multiple CPUs to improve the throughput of a sequential process by implementing a pipelined architecture. A pipeline takes advantage of parallel execution on multiple CPUs while preserving sequential dataflow.

**NOTE**

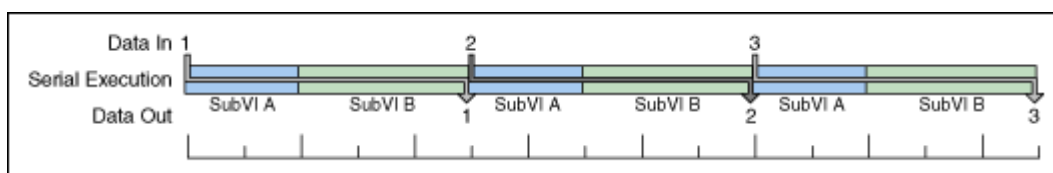
Pipelining an application requires data transfer between CPUs, which takes more time than simply passing data to the next operation on a single CPU. Therefore, implementing a pipeline increases the speed of the VI only if the time saved through parallel processing exceeds the time spent transferring data between CPUs.

Pipeline Implementation

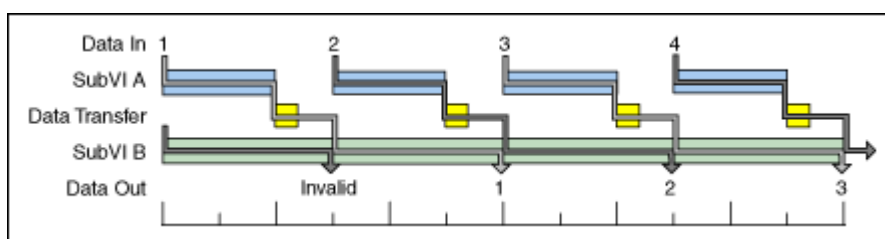
To implement a pipeline in an LabVIEW VI, divide the code into discrete steps or subVIs. For example, you can divide a complex sequential process into two discrete steps by wrapping the first half of the process in one subVI and the second half in another subVI. The following block diagram shows a standard sequential architecture on the left and a pipelined architecture on the right.



The architecture on the left does not take advantage of multiple CPUs because LabVIEW runs both subVIs in a single execution thread that must execute sequentially on a single CPU, as shown in the following illustration.



However, when you wire the subVIs together through shift registers, LabVIEW pipelines the subVIs. Now, when the VI runs on a system with more than one CPU, the subVIs execute in parallel, as shown in the following illustration.



Note When you implement a pipeline, ensure that the stages of the pipeline do not use shared resources. Simultaneous requests for a shared resource impede parallel execution and diminish the performance benefit of a multiple-CPU system.

If the CPUs executing the stages of the pipeline do not need to execute other tasks and you want to maximize CPU utilization, you can attempt to balance the stages of the pipeline so that each stage takes roughly the same time to execute. When one pipeline stage takes longer to execute than another, the CPU running the shorter stage must wait while the longer stage finishes executing.

Pipeline Dataflow

Pipelining takes advantage of parallel processing while preserving sequential dataflow dependencies. In the previous example, subVI A processes input 1 during the first iteration of the loop, while subVI B processes the default value of the shift register, yielding an invalid output. During the second loop iteration, subVI A processes input 2 while subVI B processes the output of subVI A from the first loop iteration. Notice that the output from subVI B does not become valid until the pipeline fills. Once the pipeline is full, all subsequent loop iterations yield valid output, with a constant lag of one loop iteration.



Note You must use caution to prevent undesired behavior due to the invalid outputs that occur at the beginning of pipelined execution. For example, you can use a Case structure to enable actuators only after N Timed Loop iterations elapse.

In general, the output of the final pipeline stage lags behind the input by the number of stages in the pipeline, and the output is invalid for each loop iteration until the pipeline fills. The number of stages in a pipeline is called the pipeline depth, and the latency of a pipeline, measured in loop iterations, corresponds to its depth. For a pipeline of depth N, the result is invalid until the Nth loop iteration, and the output of each valid loop iteration lags behind the input by N – 1 loop iterations.



Note The number of pipeline stages that can execute in parallel is limited to the number of available CPUs.

Pipelining with Timed Loops

(Real-Time, Windows) If you want to target each pipeline stage to a particular CPU, you can implement a pipeline using Timed Loops. You cannot use shift registers in a Timed Loop to implement a pipeline because LabVIEW maps each Timed Loop to a single thread. To implement a pipeline with Timed Loops, you must place each pipeline stage in a parallel Timed Loop and pass data between the Timed Loops using a queue, local variable, or global variable.



Note Implementing a pipeline using local or global variables can be difficult. Local and global variables do not wait for new data to become available, which can result in skipped inputs and repeated outputs from the same input.



Note Refer to the LabVIEW Real-Time Module Help for information about using pipelines in RT applications.

Prioritizing Parallel Tasks

You can prioritize parallel tasks by strategically using Wait (ms) functions or by changing the priority setting in the **ExecutionCategory** of the VI Properties dialog box.

In most cases, you should not change the priority of a VI from the default. Using priorities to control execution order might not produce the results you expect. If used incorrectly, the lower priority tasks might be pushed aside completely. If the higher priority tasks are designed to run for long periods, lower priority tasks do not run unless the higher priority task periodically waits.

Using Wait Functions

You can use the Wait function to make less important tasks run less frequently. For example, if several loops are in parallel and you want some of them to run more frequently,

use the Wait functions for the lower priority tasks. This relinquishes more time to other tasks. In many cases, using Wait functions is sufficient. You probably do not need to change the priorities by selecting **File»VI Properties** and selecting **Execution** in the VI Properties dialog box.

As described in the [Synchronous/Blocking Nodes](#) section, when a block diagram waits, the computer removes it from the queue so other tasks can run.

Wait functions are most useful in loops polling the user interface. A wait of 100 to 200 ms is barely noticeable, but it frees up the application to handle other tasks more effectively. In addition, the wait frees the operating system so it has more time to devote to other threads or applications. Consider adding waits to the less time-critical sections of block diagrams to make more time available for other tasks.

When you use the Wait (ms) function, LabVIEW waits for the amount of time you specify. When you use the Wait Until Next ms Multiple function, LabVIEW monitors a millisecond counter and waits until the counter reaches a multiple of the amount you specify.

Changing the Priorities

You also can change the priority of a VI by selecting **File»VI Properties** and selecting **Execution** in the VI Properties dialog box. You can select from the following levels of priority, listed in order from lowest to highest:

- Background priority (lowest)
- Normal priority
- Above Normal priority
- High priority
- Time-Critical priority (highest)
- Subroutine priority

The first five priorities are similar in behavior (lowest to highest), but the Subroutine priority has additional characteristics. The following two sections apply to all of these priorities, except the subroutine level.

Priorities in the User Interface Execution System

In the user interface execution system of multithreaded applications, the execution system queue has multiple entry points. The execution system places higher priority VIs on the queue in front of lower priority VIs. If a high-priority task is running and the queue contains only lower priority tasks, the high-priority VI continues to run. For example, if the execution queue contains two VIs of each priority level, the time-critical VIs share execution time exclusively until both finish. Then, the high priority VIs share execution time exclusively until both finish, and so on. However, if the higher priority VIs call a [function that waits](#), the execution system removes higher priority VIs from the queue until the wait or I/O completes, assigning other tasks (possibly with lower priority) to run. When the wait or I/O completes, the execution system reinserts the pending task on the queue in front of lower priority tasks.

Also, if a high priority VI calls a lower priority subVI, that subVI inherits the highest priority of any calling VI, even if the calling VI is not running. Consequently, you do not need to modify the priority levels of the subVIs that a VI calls to raise the priority level of the subVI.

Priorities in Other Execution Systems and Multithreaded Applications

Each of the execution systems has a separate execution system for each priority level, not including the Subroutine priority level nor the user interface execution system. Each of these prioritized execution systems has its own queue and two threads devoted to handling block diagrams on that queue.

Rather than having six execution systems, there is one for the User Interface system regardless of the priority and 25 for the other systems — five execution systems multiplied by five for each of the five priority levels.

The operating system assigns operating system priority levels to the threads for each of these execution systems based on the classification. Therefore, in typical execution, higher priority tasks get more time than lower priority tasks. Just as with priorities in the user interface execution system, lower priority tasks do not run unless the higher priority task periodically waits.

Some operating systems try to avoid this problem by periodically raising the priority level of lower priority tasks. On these operating systems, even if a high priority task wants to run continuously, lower priority tasks periodically get a chance to run. However, this behavior varies from operating system to operating system. On some operating systems, you can adjust this behavior and the priorities of tasks.

The user interface execution system is single-threaded. The user interface thread uses the Normal priority of the other execution systems. So if you set a VI to run in the Standard execution system with Above Normal priority, the user interface execution system might not run, which might result in a slow or non responsive user interface. Likewise, if you assign a VI to run at Background priority, it runs with lower priority than the user interface execution system.

As described earlier, if a VI calls a lower priority subVI, that subVI is raised to the same priority level as the caller for the duration of the call.

Subroutine Priority Level

The Subroutine priority level permits a VI to run as efficiently as possible. VIs that you set for Subroutine priority do not share execution time with other VIs.

When a VI runs at the Subroutine priority level, it effectively takes control of the thread in which it is running, and it runs in the same thread as its caller. No other VI can run in that thread until the subroutine VI finishes running, even if the other VI is at the Subroutine priority level. In single-threaded applications, no other VI runs. In execution systems, the thread running the subroutine does not handle other VIs, but the second thread of the execution system, along with other execution systems, can continue to run VIs.

In addition to not sharing time with other VIs, subroutine VI execution is streamlined so that front panel controls and indicators are not updated when the subroutine is called. A subroutine VI front panel reveals nothing about its execution.

A subroutine VI can call other subroutine VIs, but it cannot call a VI with any other priority. Use the Subroutine priority level in situations in which you want to minimize the overhead in a subVI that performs simple computations.

Also, because subroutines are not designed to interact with the execution queue, they cannot call any function that causes LabVIEW to take them off of the queue. That is, they cannot call any of the Wait, GPIB, VISA, or Dialog Box functions.

Subroutines have an additional feature that can help in time-critical applications. If you right-click on a subVI and select **Skip Subroutine Call if Busy** from the shortcut menu, the execution system skips the call if the subroutine is currently running in another thread. This can help in time-critical loops where the execution system safely skips the operations the subroutine performs, and where you want to avoid the delay of waiting for the subVI to complete. If you skip the execution of a subVI, all outputs of the subVI become the default value for the indicator on the subVI front panel.

Suggestions for Using Execution Systems and Priorities

In most applications, it is not necessary to use priority levels or an execution system other than the Standard execution system, which automatically handles the multitasking of the VIs. By default, all VIs run in the Standard execution system at Normal priority. In a multithreaded application, a separate thread handles user interface activity, so the VIs are insulated from user interface interaction. In a single-threaded application, the execution system alternates between user interface interaction and execution of the VIs, giving similar results.

In general, the best way to prioritize execution is to use Wait functions to slow down lower priority loops in the application. This is particularly useful in loops for user interface VIs because delays of 100 to 200 ms are barely noticeable to users.

If you use priorities, use them cautiously. If you design higher priority VIs that operate for a while, consider adding waits to those VIs in less time-critical sections of code so they share time with lower priority tasks.

Be careful when manipulating global variables, local variables or other external resources that other tasks change. Use a synchronization technique, such as a *functional global variable* or a *semaphore*, to protect access to these resources.

Synchronizing Access to Global and Local Variables and External Resources

Because the execution system can run several tasks in parallel, you must make sure global and local variables and resources are accessed in the proper order.

Preventing Race Conditions

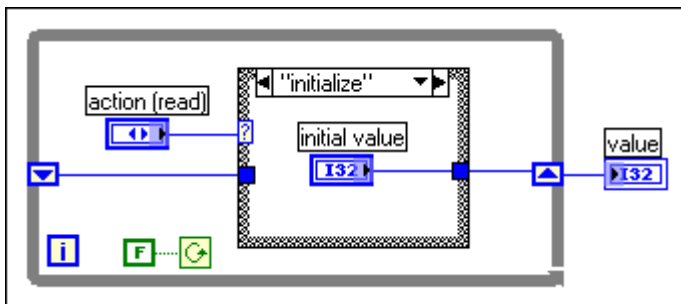
You can prevent race conditions in one of several ways. The simplest way is to have only one place in the entire application through which a global variable is changed.

In a single-threaded application, you can use a Subroutine priority VI to read from or write to a global variable without causing a race condition because a Subroutine priority VI does not share the execution thread with any other VIs. In a multithreaded application, the Subroutine priority level does not guarantee exclusive access to a global variable because another VI running in another thread can access the global variable at the same time.

Functional Global Variables

Functional global variables are non-reentrant VIs that use loops with uninitialized shift registers to hold global data. A functional global variable usually has an **action** input parameter that specifies which task the VI performs. You can use functional global variables to protect critical sections of code that access the action on a variable, which can help avoid the race conditions associated with global variables. With global variables, when you read the value of a variable, modify that value, and then write the value back to the variable, *race conditions* can occur because two pieces of parallel code that read the same variable at the same time can overwrite the changes that the other introduces. Use functional global variables to protect actions that modify data. For example, functional global variables can prevent race conditions for critical sections containing actions that read, increment, and write data to memory or actions that update a database or modify a file.

The following illustration shows a functional global variable that implements a simple count global variable. The VI uses an uninitialized shift register in a While Loop to hold the result of the operation. The actions in this example are **initialize**, **read**, **increment**, and **decrement**.



Every time you call the VI, the block diagram in the loop runs exactly once. Depending on the **action** parameter, the case inside the loop initializes, does not change, incrementally increases, or incrementally decreases the value of the shift register.

Although you can use functional global variables to implement simple global variables, as shown in the previous example, they are especially useful when implementing more complex data structures, such as a stack or a queue buffer. You also can use functional global variables to protect access to global resources, such as files, instruments, and data acquisition devices, that you cannot represent with a global variable, since calls to the functional global variable execute sequentially unless you mark the VI as reentrant.

You can solve most synchronization problems with functional global variables, because the functional global VI ensures that only one caller at a time changes the data it contains. One disadvantage of functional global variables is that when you want to change the way you modify the resource they hold, you must change the functional global VI block diagram and add a new action. In some applications, where the use of global resources changes frequently, these changes might be inconvenient. In such cases, design the application to use a semaphore to protect access to the global resource.



Semaphores

A semaphore, also known as a mutex, is an object you can use to protect access to shared resources, such as global variables. The code that accesses the shared resource is called a critical section. A semaphore protects access to a critical section by allowing only a certain number of tasks to acquire access to the semaphore at the same time. In general, you want only one task at a time to have access to a critical section protected by a common semaphore. However, it is possible to configure a semaphore to permit more than one task (up to a predefined limit) to access a critical section.

Most use cases for semaphores require the following steps:

1. **Identify the critical sections of code** —You typically use semaphores to control access to sections of code that access a shared resource. For example, if multiple subVIs interact with the same global variable, you might want to guarantee that those sections of code do not execute simultaneously so you can avoid a race condition. In this example, each section of code that reads from or writes to the global variable is a critical section of code. File and device I/O are other common examples of code that constitutes a critical section.
2. **Create a semaphore for the critical sections to share** —Use the Obtain Semaphore Reference VI to create a new semaphore or to retrieve a pre-existing semaphore by name. By default, a new semaphore allows only one task at a time to acquire the semaphore. To allow more tasks to have access to the semaphore at the same time, specify the desired number of tasks for the **size** input of the Obtain Semaphore Reference VI.
3. **Acquire access to the semaphore before each critical section** —Use the Acquire Semaphore VI to request access to the semaphore. If the semaphore is in use by another task, data flow halts at the Acquire Semaphore VI until the other task releases the semaphore. Once a section of code acquires a semaphore, no other task can acquire it until the code releases the semaphore. Therefore, the code guarded between the Acquire Semaphore VI and the Release Semaphore VI is guaranteed to complete execution before another guarded section of code begins executing.
4. **Release access to the semaphore after each critical section** —Use the Release Semaphore VI to release access to the semaphore so that another task that is waiting for the semaphore can now acquire it and execute its own critical section of code.
5. **Release the reference to the semaphore** —Use the Release Semaphore Reference VI to release a reference to a semaphore. To ensure efficient use of system

resources, always release a semaphore reference when you can guarantee that all tasks have finished using the semaphore.

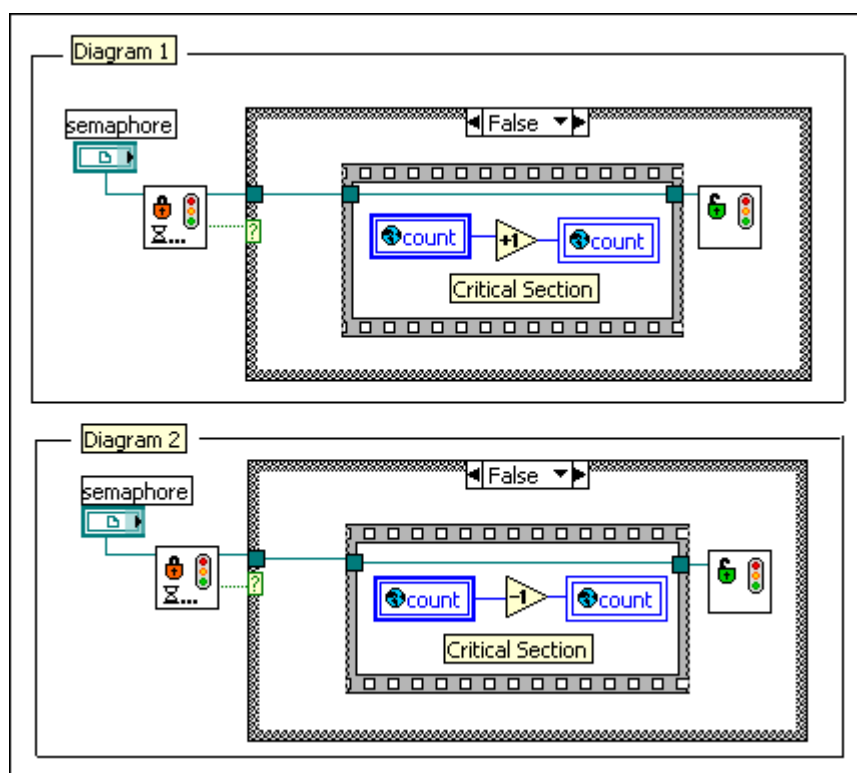


NOTE

Some advanced uses of semaphores do not acquire and release the semaphore in the same section of code.

Refer to the Simple Semaphore VI in the `labview\examples\Synchronization\Semaphore` directory for an example of using the Obtain Semaphore Reference VI and the Release Semaphore Reference VI.

The following illustration shows how you can use a semaphore to protect the critical sections. Another VI created the semaphore and passed the reference to these subVIs. The semaphore has a **size** of 1, so only one task can access the semaphore at a time.



Each of the previous block diagrams contains a critical section that accesses the same global variable, **count**. Therefore, the block diagrams share the same semaphore. Before either block diagram executes its critical section, it calls the Acquire Semaphore VI to detect if the other diagram already acquired the semaphore. If the semaphore is busy, the Acquire Semaphore VI waits until the semaphore becomes available. When the semaphore becomes available, the Acquire Semaphore VI returns FALSE for **timed out**, indicating that the block diagram has permission to execute its critical section. When the block diagram finishes executing its critical section, the Release Semaphore VI releases the semaphore, permitting another waiting block diagram to resume execution.

Lifespan of a Semaphore

A semaphore remains in memory as long as a top-level VI that is not idle owns a reference to the semaphore. If the top-level VI becomes idle, LabVIEW releases all semaphore references the VI owns, including references that exist in subVIs of the top-level VI. If LabVIEW releases the last reference to a named semaphore, LabVIEW destroys the semaphore. Because you can obtain only one reference to an unnamed semaphore, LabVIEW destroys an unnamed semaphore when the top-level VI becomes idle. To use a semaphore across multiple top-level VIs, name the semaphore and call the Obtain Semaphore Reference VI from each top-level VI so that each VI obtains its own unique reference to the semaphore.

Multiprocessing and Hyperthreading in LabVIEW

A hyperthreaded computer has a single processor but acts as a computer with a multiprocessor. (Windows) Hyperthreading is a feature of some versions of the Intel Pentium 4 and later. When you launch the Windows Task Manager on a hyperthreaded computer and click the **Performance** tab, the Windows Task Manager displays the usage history for two CPUs. (macOS) Launch the Activity Monitor to check performance.

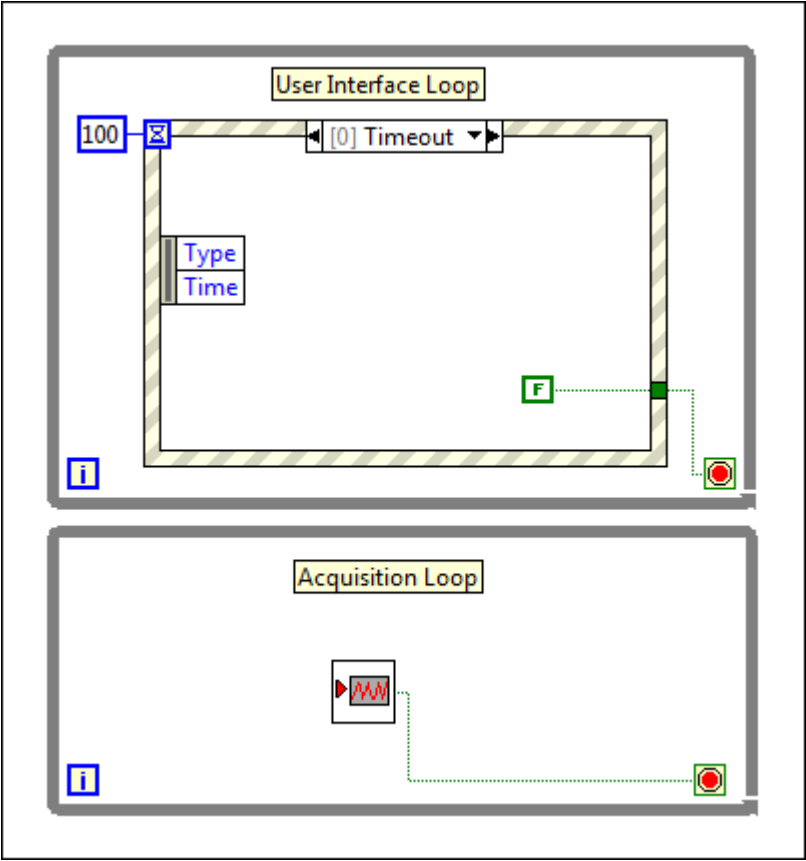
A hyperthreaded processor acts like multiple processors embedded on the same microchip. Some of the resources on the chip are duplicated, such as the register set. Other resources are shared, such as the execution units and the cache. Some resources, such as the buffers that store micro operations, are partitioned, with each logical processor receiving a portion.

Optimizing an application to take advantage of hyperthreading is similar to optimizing an application for a multiprocessor system, also known as a multi-core, multiple CPU, or SMP system, but there are some differences. For example, a hyperthreaded computer shares the execution units, and a dual-processor computer contains two complete sets of execution units. Therefore, any application that is limited by floating-point execution units performs better on the multiprocessor computer because you do not have to share the execution units. The same principle applies with cache contention. If two threads try to access the cache, the performance is better on a multiprocessor computer, where each processor has its own full-size cache.

LabVIEW Execution Overview

The LabVIEW execution system is already built for multiprocessing. In text-based programming languages, to make an application multithreaded, you have to create multiple threads and write code to communicate among those threads. LabVIEW, however, can recognize opportunities for multithreading in VIs, and the execution system handles multithreading communications for you.

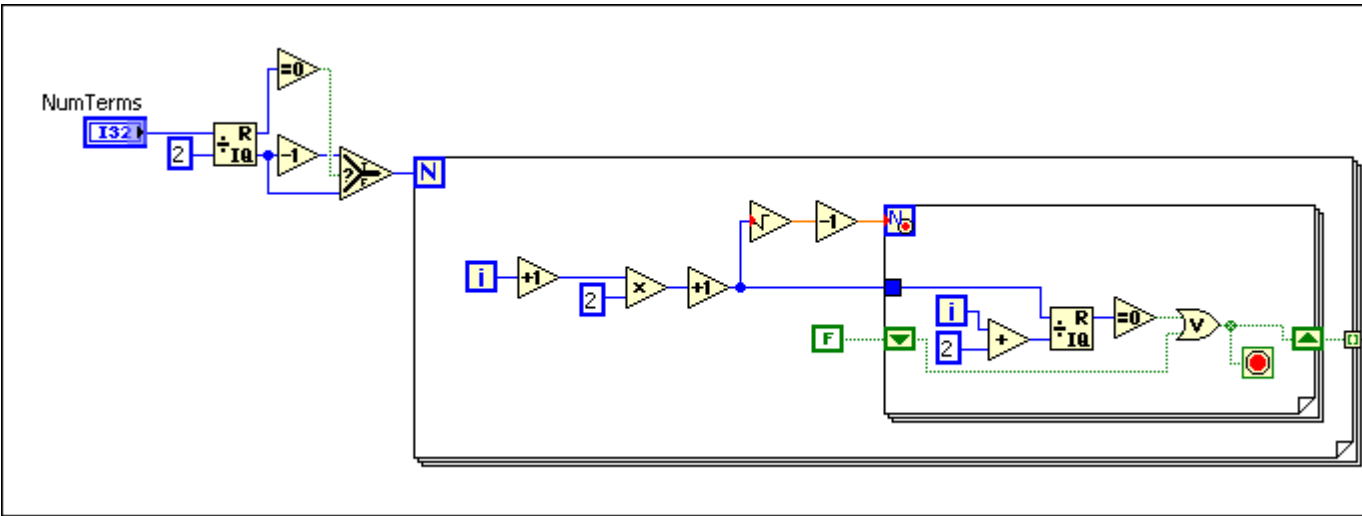
The following example takes advantage of the LabVIEW multithreaded execution system.



In this VI, LabVIEW recognizes that it can execute the two loops independently, and in a multiprocessing or hyperthreaded environment, often simultaneously.

Primes Parallelism Example

The following example calculates prime numbers greater than two.

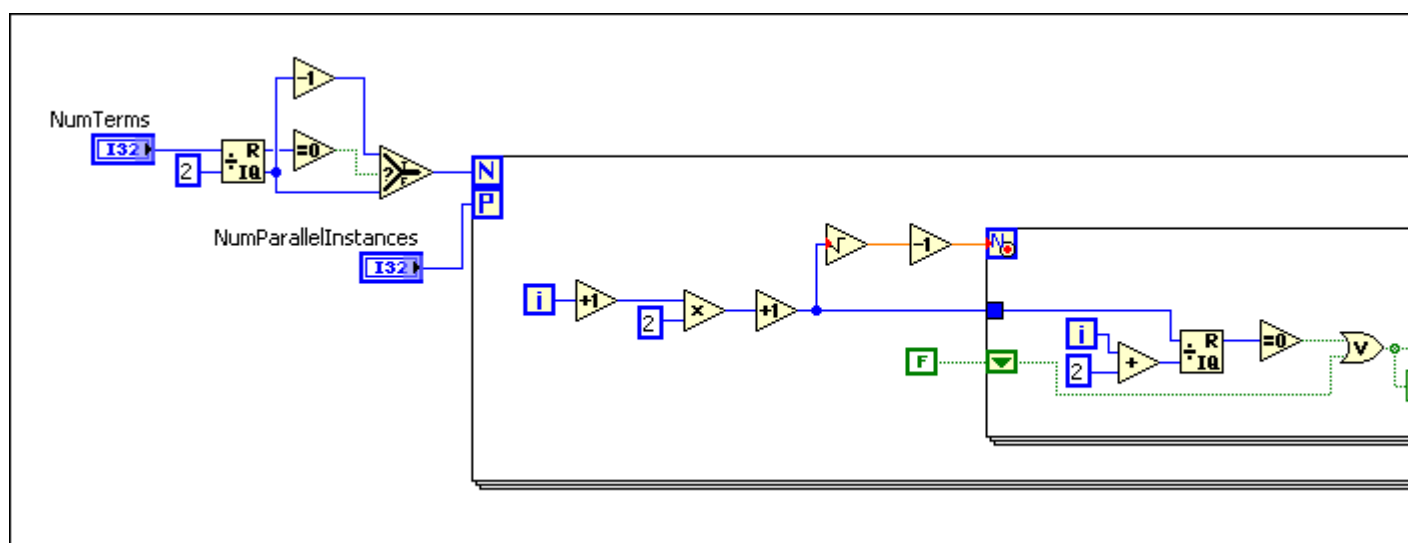


The block diagram evaluates all the odd numbers between three and **Num Terms** and determines if they are prime. The inner For Loop returns TRUE if any number divides the term with a zero remainder.

The iterations of the outer For Loop do not depend on each other. If the iterations of a For Loop can execute in any order, you can [enable loop iteration parallelism](#) on the For Loop to improve performance. In this example, the outer For Loop is a good candidate for iteration parallelism because it performs a significant amount of computation per iteration.

To enable parallelism on this VI, right-click the For Loop and select **Configure Iteration Parallelism** from the shortcut menu. This action displays the For Loop Iteration Parallelism dialog box.

On a multiprocessor computer, parallel loop iterations allow LabVIEW to take advantage of multiple processors to execute the For Loop faster. The parallel version of this example is more efficient because LabVIEW can simultaneously execute different codes.



Notice that these two example VIs do not include code for explicit thread management. The LabVIEW dataflow programming paradigm allows the LabVIEW execution system to run the parallel loop instances in different threads.

Programming for Hyperthreaded or Multiprocessor Systems

Optimizing the performance of an application for a hyperthreaded computer is nearly identical to doing so for a multiprocessor computer. However, differences exist because a hyperthreaded computer shares some resources between the two logical processors, such as the cache and execution units. If you think a shared resource on a hyperthreaded computer limits an application, test the application with an advanced sampling performance analyzer, such as the Intel VTune.

Memory Management in LabVIEW

Managing Performance and Memory

This section describes factors affecting VI performance and offers techniques to help you obtain the best performance possible. One tool this section describes is the Profile Performance and Memory window, which displays data about the execution time and

memory usage of VIs. This section also describes memory management in the dataflow model that LabVIEW uses and outlines tips for creating VIs that use memory efficiently.



NOTE

You can use the LabVIEW VI Analyzer Toolkit to check VIs interactively or programmatically for performance, efficiency, and other aspects of LabVIEW programming.

Refer to the `labview\examples\Performance` directory for examples of managing performance and memory.

Find related examples



RELATED INFORMATION

[Using the Profile Performance and Memory Window](#) on page 770

[VI Execution Speed](#) on page 265

[VI Memory Usage](#) on page 273

[VI Code Complexity](#) on page 308

[Reentrancy: Allowing Simultaneous Calls to the Same SubVI](#) on page 309

[Profiling VI Execution Time and Memory Usage](#) on page 955

[Optimizing Execution Speed for Built Applications](#) on page 956

[Choosing between Editor Responsiveness and VI Execution Speed](#) on page 958

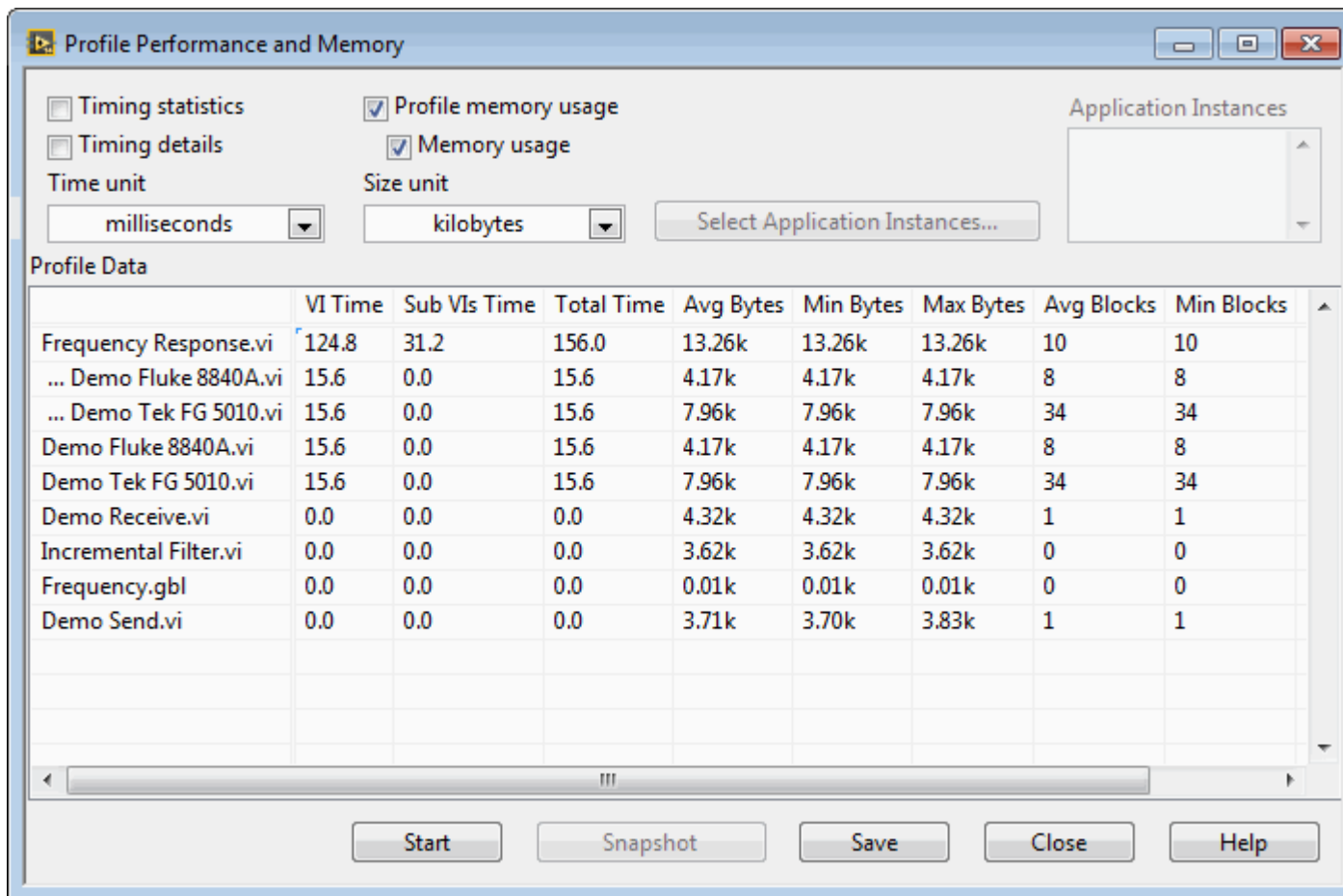
[Resolving Out of Memory Errors When Compiling a VI](#) on page 960

Using the Profile Performance and Memory Window

The Profile Performance and Memory window is a powerful tool for determining where your application is spending its time and how it is using memory. The **Profile Performance and Memory** window has an interactive tabular display of time and memory usage for each VI in your system. Each row of the table contains information for a specific VI. The time spent by each VI is divided into several categories and summarized. The **Profile Performance and Memory** window calculates the minimum, maximum, and average time spent per run of a VI.

You can use the interactive tabular display to view all or part of this information, sort it by different categories, and look at the performance data of subVIs when called from a specific VI.

Select **Tools»Profile»Performance and Memory** to display the Profile Performance and Memory window. The following illustration shows an example of the window already in use.



The collection of memory usage information is optional because the collection process can add a significant amount of overhead to the running time of your VIs. You must choose whether to collect this data before starting the Profile Performance and Memory window by checking the **Profile memory usage** checkbox appropriately. This checkbox cannot be changed once a profiling session is in progress.

Viewing the Results

You can choose to display only parts of the information in the table. Some basic data is always visible, but you can choose to display the statistics, details, and (if enabled) memory usage by placing or removing checkmarks in the appropriate checkboxes in the Profile Performance and Memory window.

Performance information also is displayed for global VIs. However, this information sometimes requires a slightly different interpretation, as described in the following category-specific sections.

You can view performance data for subVIs by double-clicking the name of the subVI in the tabular display. When you do this, new rows appear directly below the name of the VI and contain performance data for each of its subVIs. When you double-click the name of a global VI, new rows appear for each of the individual controls on its front panel.

You can sort the rows of data in the tabular display by clicking in the desired column header. The current sort column is indicated by a bold header title.

Timings of VIs do not necessarily correspond to the amount of elapsed time that it takes for a VI to complete. This is because a multithreaded execution system can interleave the execution of two or more VIs. Also, there is a certain amount of overhead not attributed to any VI, such as the amount of time taken by a user to respond to a dialog box, or time spent in a Wait (ms) function on a block diagram, or time spent to check for mouse clicks.

Timing Information

When the **Timing statistics** checkbox contains a checkmark, you can view additional details about the timing of the VI.

When the **Timing details** checkbox contains a checkmark, you can view a breakdown of several timing categories that sum up the time spent by the VI. For VIs that have a great deal of user interface, these categories can help you determine what operations take the most time.

Memory Information

If you place a checkmark in the **Memory usage** checkbox, which is only available if you place a checkmark in the **Profile memory usage** checkbox before you began the profiling session, you can view information about how your VIs are using memory. These values are a measure of the memory used by the data space for the VI and do not include the support data structures necessary for all VIs. The data space for the VI contains not just the data explicitly being used by front panel controls, but also temporary buffers the compiler implicitly created.

The memory sizes are measured at the conclusion of the run of a VI and might not reflect its exact, total usage. For example, if a VI creates large arrays during its run but reduces their size before the VI finishes, the sizes displayed do not reflect the intermediate larger sizes.

This section displays two sets of data—data related to the number of bytes used, and data related to the number of blocks used. A block is a contiguous segment of memory used to store a single piece of data. For example, an array of integers might be multiple bytes in length, but it occupies only one block. The execution system uses independent blocks of memory for arrays, strings, paths, and pictures. Large numbers of blocks in the memory heap of your application can cause an overall degradation of performance (not just execution).

Effects of Code Complexity on Compile Time

Each time LabVIEW *compiles a VI*, the compiler attempts to apply certain optimizations to the block diagram in order to maximize the eventual execution speed of the VI. To perform these optimizations, the compiler must analyze the existing code to determine where optimizations are possible and helpful. This analysis requires both memory and time in proportion to the *code complexity* of the VI. Therefore, as the amount of code in a VI increases, the memory and time that the compiler requires to perform the optimizations also increases.

Because of the memory demands to optimize exceedingly complex VIs, the compiler can potentially run out of memory or cause significant delays when it tries to compile these VIs. To avoid this problem for the vast majority of VIs, the compiler compares the complexity of each VI to a compiler optimizations threshold that causes the compiler to limit these memory-intensive optimizations when the complexity of a VI exceeds the threshold. However, limiting optimizations inherently prevents the VI from achieving maximum execution speed. Therefore, for complex VIs, the compiler produces a tradeoff between editor responsiveness and VI execution speed.

You can avoid this tradeoff entirely by dividing your VIs into subVIs. This approach yields a responsive editor, optimized execution speed, and maintainable, readable code. However, for VIs that you cannot reasonably divide further into subVIs, LabVIEW allows you to *adjust the threshold* to address the following situations:

- LabVIEW runs out of memory while compiling a VI or building an application.
- LabVIEW exhibits significant delays while compiling a VI or building an application.
- You want to compile the VI or built application for maximum execution speed after you finish editing it.

Identifying When LabVIEW Compiles a VI

In order to run a VI, LabVIEW must translate, or compile, the code of the VI into a form that the LabVIEW Runtime Engine can execute. After you make certain changes to the VI, LabVIEW must eventually recompile the code. However, because compilation requires time and memory, LabVIEW does not immediately recompile the code while you edit the VI. Instead, LabVIEW recompiles the VI the next time you perform one of the following actions:

- Saving the VI
- Running the VI
- Building an application that requires the VI
- Deploying the VI to a real-time operating system
- Loading a VI last saved in a previous version of LabVIEW or on a different platform.



NOTE

When you install an old LabVIEW toolkit for use with the current version of LabVIEW, LabVIEW automatically recompiles the toolkit VIs even before you load them.

Changes That Require LabVIEW to Recompile a VI

LabVIEW marks a VI as needing recompilation when you perform one of the following actions:

- Modifying the block diagram
- Adding a control or indicator
- Modifying the appearance of a control or indicator. However, changing the value of a control does not require LabVIEW to recompile the VI.

- Changing certain properties of the VI in the VI Properties dialog box
- Changing a subVI in a way that requires LabVIEW to recompile the subVI. If you separate compiled code from the VI or the subVI, these changes do not require LabVIEW to recompile the VI.
- Programmatically performing any of the above actions

Related Information

[Effects of Code Complexity on Compile Time](#)

[Choosing between Editor Responsiveness and VI Execution Speed](#)

[Separating Compiled Code from a File](#)

Memory Management for Large Data Sets

LabVIEW automatically handles [memory allocation](#). Since this process is automatic, LabVIEW must take precautions to handle the data safely. This means that LabVIEW copies data frequently. If the program deals with large sets of data, large and frequent data copies may lead to an out of memory error. Use the following guidelines to avoid memory problems and optimize the program to handle large data sets.



NOTE

You also can use the Profile Performance and Memory window to acquire and display data about the execution time and memory usage of VIs. Use this window to determine which VIs produce memory problems.

Reducing Copies of Large Data Sets

Because LabVIEW is a dataflow language, LabVIEW creates a copy when a VI needs more data, such as when one wire becomes two wires. Although LabVIEW usually is able to detect when to make a new copy, if LabVIEW is unsure, it makes a copy just in case.



Note When you use the LabVIEW Real-Time Module, memory copies affect the determinism of the real-time applications.

Use the Show Buffer Allocations window to determine where LabVIEW allocates memory. This window identifies data buffers, which LabVIEW uses to store copies of data.

Use the following guidelines to reduce the number of copies LabVIEW needs to make:



Note Some of the following tips are contrary to [good LabVIEW design practice](#), and you should consider these suggestions only to decrease memory consumption while working with abnormally large sets of data.

- Use the smallest possible data type. For example, use a 16-bit integer instead of a double-precision floating-point number.
- Use simple arrays. LabVIEW makes an extra copy of the data when extracting a data array for processing from waveform or dynamic data.
- Create large block diagrams or inline subVIs to **reduce overhead**. Avoid calling subVIs, because LabVIEW creates data copies when calling subVIs.
- When routing data through subVIs, make sure all block diagram terminals are outside Case structures or loops. LabVIEW generates more data copies if the terminals exist in Case structures or loops.
- Avoid routing data through loops unless necessary. If you need to route data through a loop, use shift registers instead of tunnels. If you use tunnels, LabVIEW makes a data copy every time the loop iterates.
- If possible, **use required inputs**. LabVIEW creates copies of data when generating a default value.
- Use the In Place Element structure instead of the Flat Sequence structure.



Tip Newer LabVIEW versions make fewer automatic copies of data than previous versions. So the newer the version, the better LabVIEW handles large sets of data.

Transferring Large Data Sets

When you cannot avoid creating data copies, the next possible solution is to make each copy smaller. To do so, break large data sets into smaller sets when transporting data from one place to another - a strategy known as chunking. When chunking, the copies LabVIEW makes do not adversely affect memory usage. The copies do adversely affect throughput, so minimizing them is still a good idea. The following example demonstrates this concept.

You have 512 MB of data that you need to copy to disk. You could fetch all the data and save to disk with a single call. However, even if you minimize the number of copies down to one, you are still transferring the original data and one extra copy. This means you are requesting 1 GB of memory from LabVIEW. A better approach is to loop on getting 500 KB of data a time and stream that to disk. The memory hit is now down to 1 MB, or 500 KB for the original data and 500 KB for the copy, which is well within the limits of most computers.

A side benefit is that you save the enormous amount of time it would take LabVIEW to allocate the large block of memory. Streaming 250 MB of data to disk should take 15 seconds or less on most modern computers. It could take LabVIEW that long just to allocate the 1 GB of RAM it would take to do this the other way.

If you stream large sets of data or store data in a Redundant Array of Independent Disks (RAID), you also can transfer the data without **buffering** to reduce data copies and speed up data transfers. To disable buffering, wire a TRUE value to the **disable buffering** input of the Open/Create/Replace File function.

Displaying Large Data Sets

In many interactive applications, the only thing you want to do with the data is display it for someone to see. There may be a real reason to display 5 million data points, but this

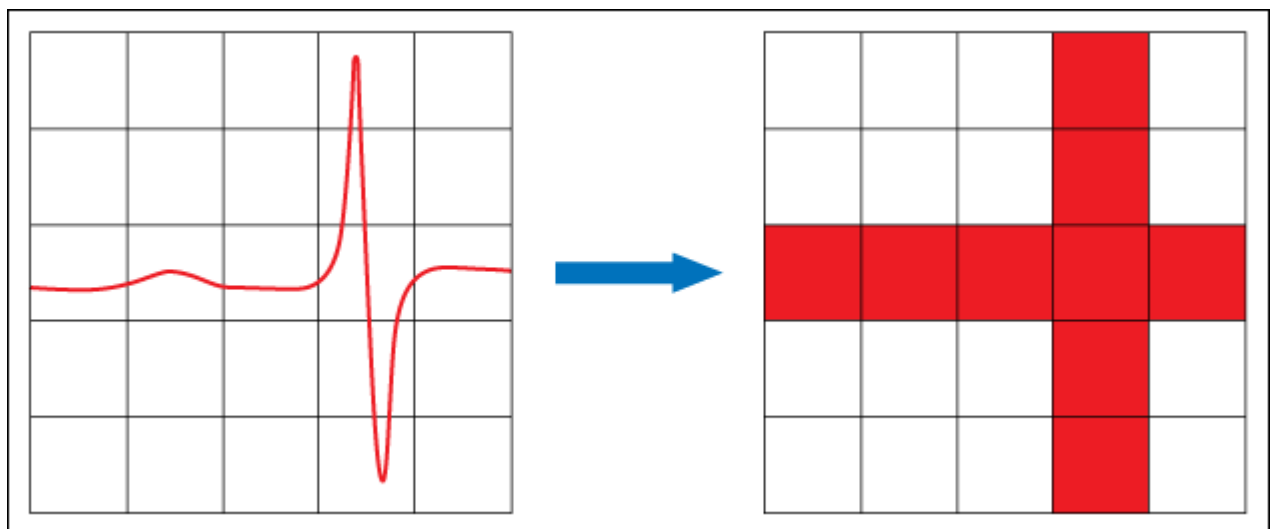
amount of data is far beyond the capabilities of most displays. The average LabVIEW graph is on the order of 300 to 1000 pixels wide. Five million points is three orders of magnitude more than you can actually see on a waveform graph. Data decimation is the answer to this problem.

For instance, you would like to see a pixel-accurate version of the huge data set on a graph. If there is a single glitch in a 5 million point buffer, the plot should be a horizontal line with a single spike one pixel wide. If the data is a sine wave with more cycles than the pixel width of the screen, the graph should be a solid band across the screen - no aliasing. A max-min decimation algorithm solves both of these use cases.

Max-min decimation is decimation in which you use the maximum and minimum data points of each decimation interval to provide the decimation. Simple decimation uses the first point of each decimation interval for the data point of the decimation interval. Simple decimation leads to aliasing artifacts, so it should not be used unless time is of the utmost importance and accuracy is not that important.

To implement max-min decimation, first determine the pixel width of the graph. Use the Plot Area:Size:Width property of the graph to find this. To reduce artifacts, you need at least two decimation intervals per pixel width, so multiply the graph pixel width by two to get the nominal number of intervals to divide the data into. Divide the data length by this number and round up to the nearest integer. This gives you the chunk size for decimation. For each chunk, find the maximum and minimum points and order them in the same order they occurred in the data set. Do not worry that the last chunk has less points than the rest. The problem is less than a pixel wide and cannot be seen on the computer screen. String all the max and min data together and plot. You will have four points per pixel width on the screen. This allows the single, pixel-wide spike to occur without bleeding over into adjacent pixels. The max/min decimation assures you always see the peaks of the data, giving you the solid band the high frequency sine wave should produce. This all occurs with much less data plotted to the graph, resulting in much higher speeds.

In the following illustration, if you process the data on the left using max-min decimation, LabVIEW produces the graph on the right.



Storing Large Data Sets

Use either a queue or a [data value reference](#) to store large data sets in memory without consuming a large amount of memory. Create a queue with a single element that contains the data. Any time you want to access the data, dequeue the element. This blocks other parts of the program from simultaneous access. After you operate on the data, enqueue the element again. The only object you need to transfer is the queue reference. LabVIEW does not create extra data copies when transferring queues. If you name the queue, you can get a reference to it at any point if you specify the name in the Obtain Queue function. Creating multiple data objects is as easy as creating multiple queues.

You also can use a data value reference to store data without creating extra data copies. Data value references are slightly more efficient than queues but have no timeout options. Instead of transferring data to a queue, you can create a reference to data and pass around the reference. To operate on the data, use the In Place Element structure. The Data Value Reference Read / Write Element border node accepts a data value reference as an input, allows you to operate on the data within the In Place Element structure, and replaces that data in the original memory space.



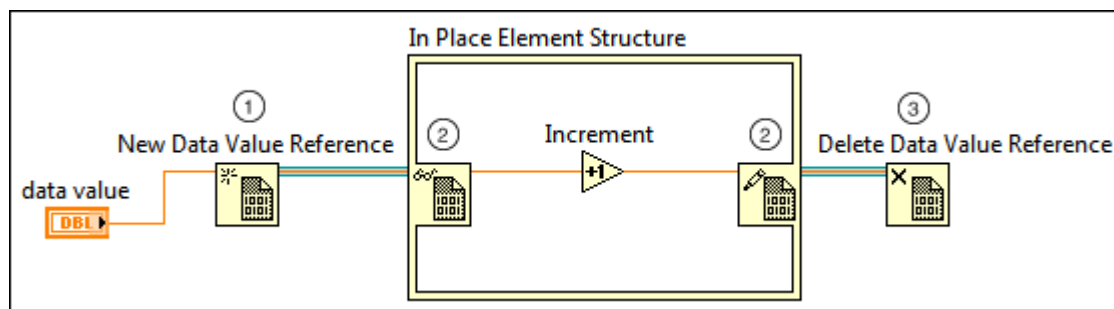
Note You can allow only member VIs of a LabVIEW class to create data value references to the class in the Inheritance page of the Class Properties dialog box.

You also can use a [functional global variable](#) to store large data sets in memory. Using a functional global variable, LabVIEW saves and accesses the data in chunks, which allows for passing data without consuming a large amount of memory. Use an uninitialized shift register as the means to hold the data. Use the Array functions to read, write, and resize the data. The Array functions operate in-place and do not create data copies. This approach is typically slower than the queue approach.

Storing Data and Reducing Data Copies with Data Value References

Because LabVIEW copies data frequently, applications with large data sets might experience slower performance and out of memory errors. You can help [manage memory](#) and avoid frequent data copies by using data value references to [store large data sets](#).

The following block diagram shows how to access and operate on the data stored at the location the data value reference points to.



NOTE

You can enable read-only access to the data value reference by right-clicking the border node on the right of the structure and selecting **AllowParallel Read-Only Access**. When the border node on the right is unwired, LabVIEW allows multiple, concurrent read-only operations and does not modify the data value reference.

Storing and Reducing Copies of External Data

External data value references are similar to data value references except they work directly with data owned by components external to LabVIEW, like the direct memory access (DMA) buffer of a device driver that controls a data-streaming device. You can work with this external data without incurring a copy to bring the data into LabVIEW. If a device driver supports an external data value reference, you can interact with the external data similarly to the way you interact with data value references. You also can use the Advanced TDMS VIs and functions to *interact* with external data value references.



[Data Communication Methods Home](#)

How LabVIEW Stores Data in Memory

LabVIEW stores data in memory for controls, indicators, wires, and other objects depending on the data type of the object.

Boolean Data

LabVIEW stores Boolean data as 8-bit values. If the value is zero, the Boolean value is FALSE. Any nonzero value represents TRUE.

Byte Integer

Byte integer numbers have an 8-bit format, signed and unsigned.

Word Integer

Word integer numbers have a 16-bit format, signed and unsigned.

Long Integer

Long integer numbers have a 32-bit format, signed and unsigned.

Quad Integer

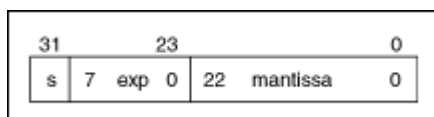
Quad integer numbers have a 64-bit format, signed and unsigned.

Fixed-Point

Fixed-point numbers have a 64-bit format, signed and unsigned. LabVIEW signs extend fixed-point numbers to 64 bits even when the extended bits are not significant.

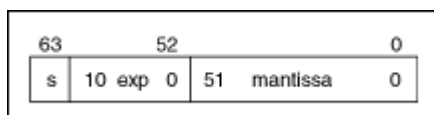
Single

Single-precision floating-point numbers have a 32-bit IEEE single-precision format.



Double

Double-precision floating-point numbers have a 64-bit IEEE double-precision format.



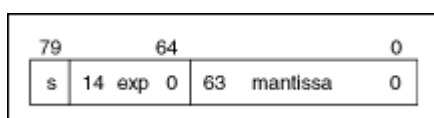
Extended

Extended-precision floating-point numbers have an 80-bit IEEE extended-precision format.



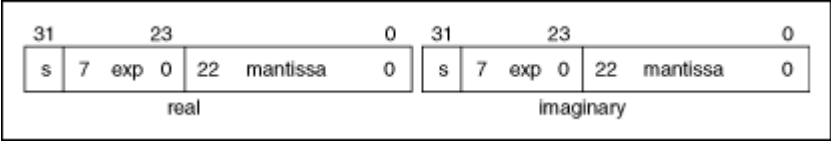
NOTE

In some cases, extended-precision floating-point numbers can have a 64-, 96-, and 128-bit IEEE extended-precision format depending on the computer processor. Most Intel and AMD processors use an 80-bit format.



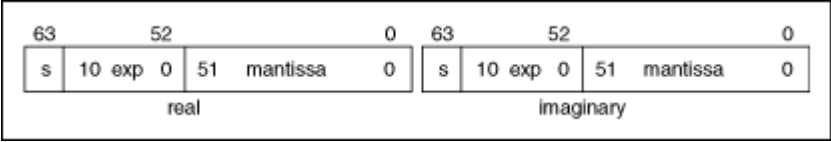
Complex Single

Complex single-precision floating-point numbers consist of real and imaginary values in 32-bit IEEE single-precision format.



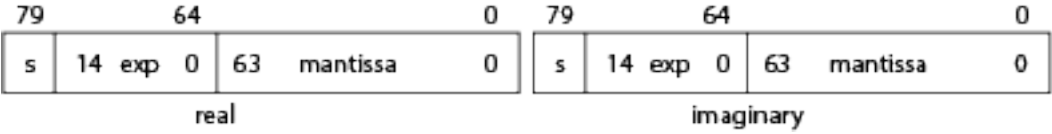
Complex Double

Complex double-precision floating-point numbers consist of real and imaginary values in 64-bit IEEE double-precision format.



Complex Extended

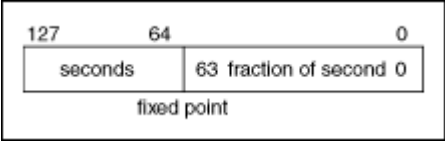
Complex extended-precision floating-point numbers consist of real and imaginary values in IEEE extended-precision format.



For floating point and complex numbers, s is the sign bit (0 for positive, 1 for negative), exp is the biased exponent (base 2), and mantissa is a number in the [0,1] range.

Time Stamp

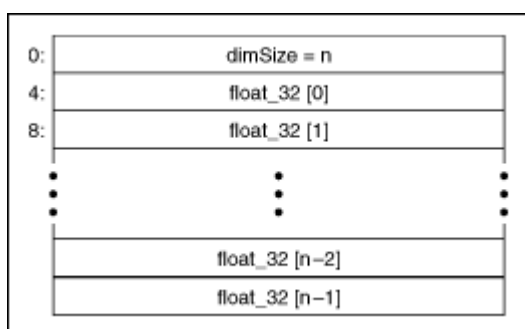
LabVIEW stores a time stamp as a cluster of two doubles. The first double represents the time-zone-independent number of complete seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time [01-01-1904 00:00:00]. The second double represents the fractions of seconds.



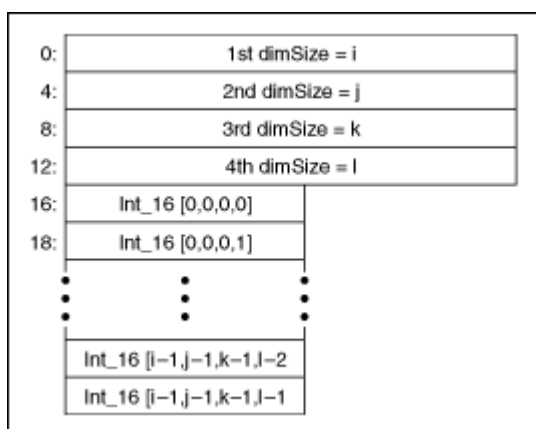
Arrays

LabVIEW stores arrays as handles, or pointers to pointers, that contain the size of each dimension of the array in 32-bit integers followed by the data. If the handle is 0, the array is empty. Because of alignment constraints of certain platforms, a few bytes of padding might follow the dimension size so that the first element of the data is correctly aligned. If you write a shared library that uses a LabVIEW array, you might need to [adjust the size of the handle](#).

The following illustration shows a 1D array of single-precision floating-point numbers. The decimal numbers to the left represent the byte offsets of locations in memory where the array begins.



The following illustration shows a 4D array of 16-bit integers.

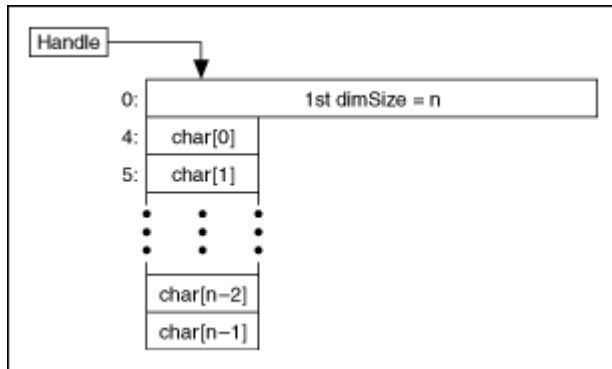


In LabVIEW, 1D and 2D arrays are aligned in memory. This aids in performance for linear algebra operations and operations involving the matrix data type.

Strings

LabVIEW stores strings as pointers to a structure that contains a 4-byte length value followed by a 1D array of byte integers (8-bit characters), as shown in the following illustration. If the handle, or the pointer to the structure, is NULL, LabVIEW treats the string as an empty string, which is the same as a string with length value of zero. Because LabVIEW uses the length value and not a terminating character to determine the end of the string, you can embed all characters that include the NULL character, or ASCII character 0, in the string at any point. When passing LabVIEW strings to external code that expects a C

string, the embedded NULL characters cause the code to interpret the string as terminating at the first NULL character.



Paths

A LabVIEW path is a pointer to an opaque data structure that contains a path type and path components. The path type is 0 for an absolute path, 1 for a relative path, and 3 for a Universal Naming Convention (UNC) path. A UNC path occurs on Windows only and has \<machine name>\<share name>\ rather than a drive letter as its first component. Any other value of path type indicates an invalid path.

Use the following functions to access information about a path:

- FDepth()
- FDirName()
- FIsAPath()
- FIsAPathOfType()
- FIsEmptyPath()
- FNamePtr()
- FVolName()

Use the following functions to add elements to a path:

- FAddPath()
- FAppendName()

Use the following functions to create a new path:

- FNotAPath()
- FPathCpy()
- FEmptyPath()
- FMakePath()

- `FRelPath()`

Use the `FDestroyPath()` function to dispose of a path.

Use the following functions to convert a path to or from a textual form:

- `FFileSystemStringToPath()`
- `FPathToFileSystemDSString()`
- `FPathToText()`
- `FTextToPath()`
- `ConvertPathToPlatformIndependentText()`
- `ConvertPlatformIndependentTextToPath()`

Use the following functions to compare two paths:

- `FPathCmp()`
- `FPathCmpLexical()`

Related Information

Using LabVIEW Manager Functions in Shared Libraries

Clusters

LabVIEW stores cluster elements of varying data types according to the cluster order. You can view and *modify the cluster order* by right-clicking the cluster border and selecting **Reorder Controls In Cluster** from the shortcut menu. LabVIEW stores scalar data directly in the cluster, and it stores arrays, strings, and paths indirectly in the cluster. Because of alignment constraints of certain platforms, LabVIEW might add some bytes of padding to the end of a cluster in an array so that the beginning of the next cluster is aligned correctly. If you write a shared library that uses an array of clusters, you must account for this padding if you *adjust the size of the handle* to the array.

The memory layout of a cluster depends on the platform you are running. LabVIEW may add padding between cluster elements so that those elements are aligned to certain address boundaries. Address boundaries relate to the concept of natural alignment. A piece of data is aligned on its natural boundary if the address at which it begins is a multiple of the size of the data. For example, a 4-byte integer is aligned naturally if it begins at an address which is a multiple of four. Most compilers have an upper limit on this boundary at which the restriction is relaxed. For instance, an 8-byte integer may be aligned on a 4-byte boundary. For the platforms on which LabVIEW runs, the alignment constraints are as follows:

- (Phar Lap ETS) Data is aligned only to 1-byte boundaries.
- (macOS 64-bit and Linux 64-bit) Data is aligned naturally up to 8-byte boundaries.

- (Windows 64-bit and VxWorks) Data is aligned naturally up to 8-byte boundaries.

On all platforms, clusters adopt the alignment of their most restrictively aligned element.

The following tables represent a cluster that contains a 16-bit integer, a double-precision floating-point number, an extended-precision floating-point number, an array, and an 8-bit unsigned integer. Each row contains the memory address and its corresponding element and element size.

- (Phar Lap ETS) Because data is aligned only to 1-byte boundaries, the memory address of the next cluster is 17.

Memory Address	Element	Size (bytes)
I16 integer	2	
EXT float	10	
Handle to Array	4	
U8 integer	1	

- (Windows 64-bit, macOS 64-bit, and Linux 64-bit) Because data is aligned naturally up to 8-byte boundaries, LabVIEW adds 7 bytes of padding after the U8 integer so that the memory address of the next cluster is 32.

Memory Address	Element	Size (bytes)
I16 integer	2	
EXT float	10	
Padding	4	
Handle to Array	8	
U8 integer	1	
Padding	7	

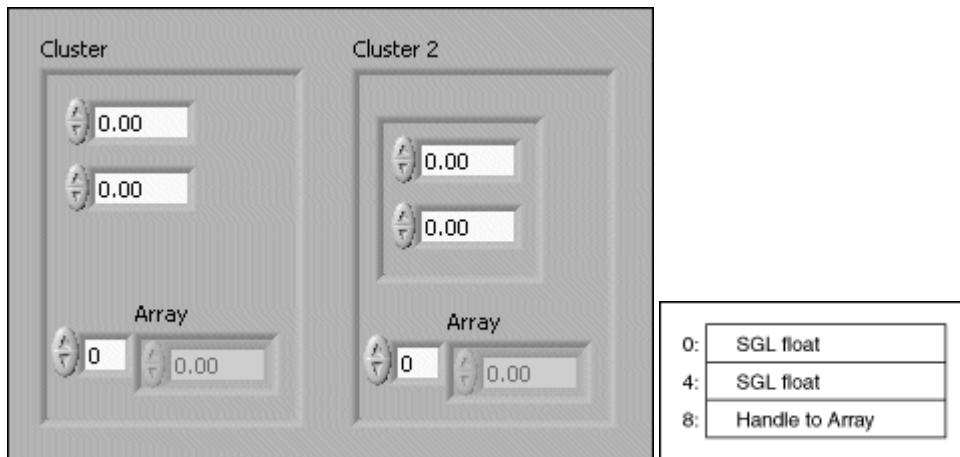
- (VxWorks) Because data is aligned naturally up to 8-byte boundaries, LabVIEW adds 7 bytes of padding after the U8 integer so that the memory address of the next cluster is 40. Note that VxWorks implements EXT float as long double.

Memory Address	Element	Size (bytes)
I16 integer	2	
Padding	6	
EXT float	16	
Handle to Array	8	
U8 integer	1	

Memory Address	Element	Size (bytes)
Padding	7	

LabVIEW stores clusters embedded within other clusters directly in line and with no indirection. LabVIEW stores only arrays, strings, and paths indirectly.

The following illustration shows two different clusters that store data the same way.



Waveform

LabVIEW stores waveforms like clusters.

Refnum

LabVIEW stores refnums as signed 32-bit integers.

Variant

LabVIEW stores variants as handles to a LabVIEW internal data structure. Variant data is made up of 4 bytes.

Related Concepts

[Flattened Data](#)

[Type Descriptors](#)

Flattened Data

LabVIEW converts data from the format in memory to a form more suitable for writing to or reading from a file. This more suitable format is called flattened data.

Because LabVIEW stores strings, arrays, and paths in handles (pointers to pointers in separate regions of memory), clusters that contain these strings and arrays are noncontiguous. In general, LabVIEW stores data in tree form. For example, LabVIEW stores a cluster as a double-precision, floating-point number and a string as an 8-byte floating-point number followed by a 4-byte handle to the string. LabVIEW does not store the string data adjacent to the extended-precision, floating-point number in memory. Therefore, to write the cluster data to disk, LabVIEW must get the data from two different places. With a cluster that contains many strings, arrays, and/or paths, LabVIEW stores the data in many different places.

When you save data to a file, LabVIEW flattens the data into a single string before saving it. Flattening data makes the data from an arbitrarily complex cluster contiguous rather than stored in several places. When LabVIEW loads data from a file, LabVIEW must perform the reverse operation, reading a single string and unflattening it into its internal, possibly noncontiguous, form.

LabVIEW normalizes the flattened data to a standard form so VIs that run on any platform can use the data. LabVIEW stores flattened numeric data in big-endian form (most significant byte first), and stores flattened extended-precision, floating-point numbers as 16-byte quantities.

When writing data to a file for use by an application that was not created using LabVIEW or when reading data from a file produced by an application that was not created using LabVIEW, you can transform the data into little-endian (least significant byte first) or big-endian form after flattening or before unflattening. Windows-based applications typically expect numeric data to be in little-endian form.

Use the Flatten To String and Unflatten From String functions to flatten and unflatten data just as LabVIEW does internally when LabVIEW saves and loads data.

The flattened form of a piece of data does not encode the type of the data. LabVIEW stores this information in a [type descriptor](#). The Unflatten From String function requires you to wire a data type as an input so the function can decode the string properly.

Use the [variant data type](#) to work with data independently of data type instead of flattening the data when you write to memory and unflattening the data when you read from memory. Use the Variant functions to create and manipulate variant data.



NOTE

If you use the Flatten To String function on variant data, LabVIEW will flatten the variant and all its contents, including attributes. However, if you use the Variant To Flattened String function, LabVIEW flattens only the variant and discards any attributes.

Booleans and Numeric Objects

The flattened form of any numeric type and Boolean type stores the data only in big-endian format. For example, a 32-bit integer with a value of -19 is flattened to FFFF FFED. A double-precision floating-point number with a value equal to $1/4$ is flattened to 3FD0 0000 0000 0000. A Boolean value of TRUE is any nonzero value. A Boolean value of FALSE is 00.

The flattened form for extended-precision numbers is the 128-bit extended-precision, floating-point format. When you save extended-precision numbers to disk, LabVIEW stores them in this format.

Strings and Paths

Because strings and paths have variable sizes, a flattened 32-bit integer that records the length in bytes precedes the flattened form. For example, a string type with value ABC is flattened to 0000 0003 4142 43. For strings, the flattened format is similar to the format the string takes in memory.

However, paths do not have a length value preceding them when LabVIEW stores them in memory, so this value comes from the actual size of the data in memory and prefixes the value when LabVIEW flattens the data. This length is preceded by four characters: PTH0.

For example, a path with value C:\File is flattened to 5054 4830 0000 000B 0000 0002 0143 0466 696C 65.

5054 4830 indicates PTH0. 0000 000B indicates 11 bytes total. 0000 is the type. 0002 is the number of components. 0143 indicates the letter C as a Pascal string. 0466 696C 65 indicates the word File as a Pascal string.

Arrays

Flattened 32-bit integers that record the size, in elements, of each of the dimensions of an array precede the data for a flattened array. The slowest varying dimension is first, followed successively by the faster varying dimensions, just as the dimension sizes are stored in memory. The flattened data follows immediately after these dimension sizes in the same order in which LabVIEW stores them in memory. The following example shows a 2D array of six 8-bit integers.

{ {1, 2, 3}, {4, 5, 6} } is flattened to 0000 0002 0000 0003 0102 0304 0506.

The following example shows a flattened 1D array of Boolean variables.

{T, F, T, T} is flattened to 0000 0004 0100 0101. The preferred value for TRUE is 01.

Clusters

A flattened cluster is the concatenation, in cluster order, of the flattened data of its elements. For example, a flattened cluster of a 16-bit integer of value 4 (decimal) and a 32-bit integer of value 12 is 0004 0000 000C.

A flattened cluster of a string ABC and a 16-bit integer of value 4 is 0000 0003 4142 4300 04.

A flattened cluster of a 16-bit integer of value 7, a cluster of a 16-bit integer of value 8, and a 16-bit integer of value 9 is 0007 0008 0009.

Waveforms

LabVIEW flattens waveforms like clusters.

Refnums

LabVIEW stores the majority of flattened refnums as *flattened 32-bit integers*, which represent an internal LabVIEW data structure.

You can classify the remaining refnums by their refnum *type code*.

Type codes 0xE, 0xF and 0x15 are refnums that store their data as a *flattened string*. This string contains the value of the refnum tag, and can be empty (4 bytes of zero).

Type codes 0x1A, 0x1C, and 0x1D concatenate, in the following order:

- 1. A flattened string for the name in the refnum tag. This string is empty (4 bytes of zero) if the refnum does not have a tag.
- 2. A flattened string that contains information specific to the refnum. This string can be empty (4 bytes of zero).
- 3. A flattened string that contains information specific to the refnum. This string can be empty (4 bytes of zero).
- 4. A flattened 32-bit signed integer that contains information specific to the refnum.
- 5. A flattened string that contains information specific to the refnum. This string can be empty (4 bytes of zero).

Classes

LabVIEW flattens a LabVIEW class according to the following general format:

level in hierarchy	class name	version list	private data

- level in hierarchy —4-byte unsigned integer representing the number of levels into the class hierarchy the class occurs. For example, if this value is 2, the class has one ancestor class between itself and LabVIEW Object.

Note If level in hierarchy is 0, the object is an instance of LabVIEW Object, and LabVIEW excludes the other three fields from the string.

- class name —A sequence of Pascal strings preceded by a single byte that indicates the length of the qualified name of the class. Each Pascal string, or sequence of bytes where the first byte indicates the length of the string and the remaining bytes are the text of the string, represents one segment of the qualified name. The last Pascal string is a length byte of zero. This section of the flattened string includes enough zero pad bytes to increase the class name section to a multiple of 4 bytes.
- version list —A series of clusters in which each cluster is four 2-byte unsigned integers that represent a version number. The first cluster in this list represents the version of class name, the second is the version of its parent, and so on. This list contains one version number for each level in hierarchy.

Note If level in hierarchy is 1 and the version is 0, the flattened data represents the default data of the class, and LabVIEW excludes the remaining field from the string.

- private data —A series of flattened clusters representing the private data of each level of the hierarchy. In reverse order of the version list, the first cluster in this series corresponds to the oldest ancestor class. Each flattened cluster begins with a 4-byte signed integer that represents the number of bytes in the data that follows. If this initial number is 0, the flattened cluster represents the default data for the corresponding class in the hierarchy. Otherwise, the following data uses the standard flattened cluster representation explained earlier in this topic. Each flattened cluster ends with enough pad bytes to increase the cluster to a multiple of 4 bytes.

.NET in LabVIEW

Learn about .NET support in LabVIEW.

Using .NET with LabVIEW

.NET allows you to access Windows services such as the performance monitor, event log, and file system, as well as advanced Windows APIs such as the Speech Recognition and Generation service. .NET also provides access to Web protocols such as SOAP, WSDL, and UDDI.

You can use LabVIEW as a .NET client to access the objects, properties, and methods associated with .NET servers. With .NET Framework, you can also access events.

With .NET Framework, you also can host .NET Framework user interface controls on the front panel of a VI. Although LabVIEW is not a .NET Framework server, you can

communicate to LabVIEW remotely with the ActiveX server interface through .NET Framework support for COM objects.



NOTE

Creating and communicating with .NET Framework objects in LabVIEW requires the .NET CLR 4.0 that installs with LabVIEW. You must use a .NET 2.0 configuration file if you want to load .NET 2.0 mixed-mode assemblies. Refer to the *Requirements for Using .NET with LabVIEW* for more information about .NET Framework restrictions in LabVIEW. NI strongly recommends that you use .NET Framework objects only in LabVIEW projects.

Be sure to save .NET Framework assemblies in appropriate locations to ensure that they load and behave as expected.



RELATED INFORMATION

[Using ActiveX with LabVIEW](#) on page 398

[Requirements for Using .NET Framework Assemblies in LabVIEW](#) on page 791

[Loading .NET Framework 2.0, 3.0, and 3.5 Assemblies in LabVIEW](#) on page 1484

[Creating .NET Objects](#) on page 792

[Using Libraries in LabVIEW Projects](#) on page 370

[Loading .NET Framework Assemblies](#) on page 797

[LabVIEW New Features and Changes](#) on page 47

Learn about updates, including new features and behavior changes, introduced in each version of LabVIEW.

.NET Framework Environment



NOTE

The following content applies to .NET Framework only. LabVIEW has been gradually adding support for .NET Core since LabVIEW 2024 Q3. Refer to *LabVIEW New Features and Changes* for specific .NET Core support added in different releases.

Understanding the components of the .NET environment is helpful for understanding how the .NET works as well as for ensuring that your computer is prepared to use .NET Framework with LabVIEW. After learning about the basic components of the .NET environment, verify that the .NET environment on your computer meets the [minimum requirements for using .NET with LabVIEW](#).

The .NET Framework environment consists of the following items:

- **.NET Framework** —The collective term for the Common Language Runtime and the class libraries that compose a specific version of .NET Framework.
- **Common Language Runtime (CLR)** —An execution system that is responsible for loading and running .NET Framework assemblies. The CLR provides run-time services such as language integration, security enforcement, memory, garbage

collection, process management, and thread management. To help .NET Framework communicate with various programs, the CLR provides a data type system that supports multiple programming languages and operating systems. Developers then can view the system as a collection of data types and objects rather than as a collection of memory and threads. The CLR requires compilers and linkers to generate information in the CLR intermediate language (IL) metadata format. All .NET Framework programming language compilers generate CLR IL code rather than assembly code.

- **Class Libraries** —A set of classes that provides standard functionality such as input/output, string manipulation, security management, network communications, thread management, text management, and user interface design. These classes provide the same functionality as the Win32/COM system. In the .NET Framework, you can use classes created in one .NET Framework language in another .NET Framework language.
- **Assemblies** —A unit of deployment similar to a DLL, OCX, or executable for a component in COM. Assemblies are DLLs and executables you build using a .NET Framework compiler. Assemblies can consist of a single file or multiple files. An assembly includes a manifest that contains information about the assembly name, version, publisher security, files that make up the assembly, dependent assemblies, resources, and exported data types.
- **Global Assembly Cache (GAC)** —A listing of the public assemblies available on the system. The GAC is analogous to the registry that COM uses.

Requirements for Using .NET Framework Assemblies in LabVIEW

Whether you want to create and manipulate .NET Framework objects in LabVIEW or use LabVIEW to build .NET Framework assemblies for other applications to use, you must adhere to the following default requirements and settings.

Compatible Common Language Runtime (CLR)

To create and manipulate .NET Framework objects in LabVIEW, you must use the .NET CLR 4.0. This version of the CLR is part of the .NET Framework 4.0 that installs with LabVIEW.

LabVIEW uses the CLR 4.0 to:

- **Load assemblies** —LabVIEW can load .NET Framework assemblies that target the .NET Framework CLR 4.0 or earlier. However, LabVIEW loads all assemblies in the .NET Framework CLR 4.0.
- **Build stand-alone applications, shared libraries, packed project libraries, source distributions, and ZIP files that contain VIs that use .NET Framework functions or build .NET Framework interop assemblies** —LabVIEW targets these applications to the .NET Framework CLR 4.0.

Compatible Class Libraries

LabVIEW has access to the class library assemblies associated with the .NET CLR 4.0, which installs with LabVIEW. The class library assemblies of the .NET Framework determine which .NET Framework features are available to LabVIEW applications. If you **build an application** from a LabVIEW project or VI that relies on assemblies, the target computer for the application must have the .NET Framework 4.0 installed.

Unsupported .NET Framework Features in LabVIEW

LabVIEW supports most language features of the .NET Framework 4.0. However, .NET Framework 4.0 introduces a few new features to .NET Framework CLR that LabVIEW does not support. Refer to the KnowledgeBase at ni.com for a list of .NET Framework features that LabVIEW does not support.

Locations for Saving .NET Framework Assemblies on Disk

You must save .NET Framework assemblies that you want to call from LabVIEW in a location where LabVIEW can find them. Use the following guidelines to decide where to save assemblies.



NOTE

NI strongly recommends that you always put VIs that use .NET Framework objects in a project.

Paths Options *Loading .NET Assemblies*



TIP

If you encounter problems loading a particular assembly or accessing objects in an assembly, **debug the assembly call** or use the .NET Assemblies in Memory dialog box to verify that the correct version of the assembly is in memory.

Creating .NET Objects

You can create .NET objects on the block diagram. Use a Constructor Node to create a .NET object on the block diagram and access properties and methods for the object. With .NET Framework, use a .NET container to create a .NET Framework control on the front panel. You also can add .NET Framework controls to the Controls palette for later use. You can handle events for .NET Framework objects.



NOTE

You cannot switch between design mode and run mode for a .NET container as you can for an ActiveX container. The behavior of a .NET container is similar to that of an ActiveX container in design mode.

.NET Framework objects can be visible or invisible to the user. For example, buttons, windows, pictures, documents, and dialog boxes are visible to users. Application objects are invisible to users.

.NET Framework controls automatically use the user interface execution system.

Use the .NET functions to access the .NET objects and their associated properties and methods.



RELATED INFORMATION

[Creating .NET Objects on the Block Diagram](#) on page 1482

[Creating .NET Controls in a .NET Container](#) on page 1481

[Creating .NET Subpalettes](#) on page 839

[Using ActiveX with LabVIEW](#) on page 398

[Multitasking in LabVIEW](#) on page 752

[.NET Properties and Methods](#) on page 793

.NET Properties and Methods

.NET objects expose properties and methods that other applications can access. You access an application by accessing an object associated with that application and setting a property or invoking a method of that object.

On the front panel or block diagram, right-click a .NET object and select **Create»Property Node** or **Create»Invoke Node**, and select a property or method from the shortcut menu to set a property or invoke a method for the object. You also can select .NET-specific properties or invoke .NET-specific methods for a .NET object. On the block diagram, right-click a .NET object and select **Create»Property Node** or **Create»Invoke Node**, and select a .NET-specific property or method from the shortcut menu.



NOTE

You cannot view and set .NET properties in a Property Browser as you can for [ActiveX](#). You must set .NET properties using Property Nodes.

.NET Events

.NET events are the actions taken on a .NET object, such as clicking a mouse, pressing a key, or receiving notifications about things such as running out of memory or tasks completing. Whenever these actions occur to the object, the object sends both an event to alert the .NET container and the event-specific data. The .NET object defines the events available for an object.

To use .NET events in an application, you must [register for the event and handle the event when it occurs](#). .NET event registration is similar to [dynamic event registration](#). However, the architecture of a .NET event VI is different from the architecture of an event-handling VI. The following components make up a typical .NET event VI:

- .NET object for which you want to generate an event.

- Register Event Callback function to specify and register for the type of event you want to generate. The Register Event Callback function is a growable node capable of handling multiple events, similar to the Register For Events function.
- **Callback VI** that contains the code you write to handle the event you specify.

When you wire a .NET object to the Register Event Callback function and specify the event you want to generate for that object, you are registering the .NET object for that event. After you register for the event, **create a callback VI** that contains the code you write to handle the event. Different events might have different event data formats so changing the event after you create a callback VI might break wires on the block diagram. Select the event before you create the callback VI.

You can handle events on .NET Framework controls in a container. For example, you can place a calendar control in a .NET Framework container and specify that you want to handle a DoubleClick event for the items displayed in the calendar.



NOTE
LabVIEW does not support .NET Core controls.

.NET Data Types

LabVIEW converts the data types of .NET property, method, and constructor parameters into LabVIEW data types so LabVIEW can read and interpret the data. LabVIEW displays data types it cannot convert as .NET refnums.

The following table lists the supported .NET Framework data types and the corresponding LabVIEW data types.

Table 2 : Supported .NET Framework Data Types and Corresponding LabVIEW Data Types

Supported .NET Framework Types	LabVIEW Types
System.Int64, System.UInt64, System.Int32, System.UInt32, System.Int16, System.UInt16	
System.String	
System.Boolean	
System.Byte, System.Char, System.UByte	
System.Single, System.Double, System.Decimal	
System.Array	Displayed as an array of the corresponding type
Enumeration	Displayed as a ring control, constant, or indicator with an integer value of a corresponding type.

Table 2 : Supported .NET Framework Data Types and Corresponding LabVIEW Data Types (Continued)

Supported .NET Framework Types	LabVIEW Types
DateTime	
Any other .NET Framework object	

The following table lists the supported .NET 8.0 (Core) data types and the corresponding LabVIEW data types.

Table 3 : Supported .NET 8.0 (Core) Data Types and Corresponding LabVIEW Data Types

Supported .NET 8.0 (Core) Types	LabVIEW Types
System.Int64, System.UInt64, System.Int32, System.UInt32, System.Int16, System.UInt16	 ,  ,  ,  ,  , 
System.String	
System.Boolean	
System.Byte, System.Char	
System.Single, System.Double	
System.Array	Displayed as an array of the corresponding type
Any other .NET 8.0 (Core) object	

Configuring a .NET Framework Client Application

In LabVIEW, .NET Framework client applications are LabVIEW projects and VIs that create and manipulate objects defined in .NET Framework assemblies. When an application uses a .NET Framework object, LabVIEW employs a set of *default behaviors* to determine which version of a particular .NET Framework assembly to reference and which version of the Common Language Runtime (CLR) loads that assembly. However, you can create .NET configuration files to override the default ways that LabVIEW loads and interacts with .NET objects for a specific project or for all projects. You must use a configuration file for each of the following tasks:

- *Debugging some assembly calls*
- *Loading .NET Framework 2.0 assemblies*
- *Loading a specific version of an assembly*
- Setting up remote .NET Framework access
- Setting logging options
- Setting security preferences

Requirements for .NET Framework Configuration Files

A .NET Framework configuration file must meet the following requirements:

- Corresponds to a specific LabVIEW project, shared library, or stand-alone application
- Has a name that matches the file it configures followed by the `.config` extension. For example, the configuration file for `MyApp.lvproj` must be named `MyApp.lvproj.config`, the configuration file for `MyApp.exe` must be named `MyApp.exe.config`, and so on.
- Resides in the same directory as the application it configures



NOTE

National Instruments strongly recommends that you use .NET Framework objects only in VIs that are members of a [LabVIEW project](#). VIs that are not part of a project can reference only the `LabVIEW.exe.config` file located in the same directory as `LabVIEW.exe`.

- Contains XML content that is specific to its configuration purpose. Refer to the Microsoft Developer Network (MSDN) for information about creating configuration files for setting up remote .NET Framework access, setting logging options, and setting security.

Building Applications from Configured Projects

When you build a stand-alone application from a LabVIEW project that uses a .NET Framework configuration file, the resulting application also requires a copy of the configuration file. To meet the [configuration file requirements](#), the configuration file for the stand-alone application must reflect the name of the application, and the two files must be located in the same directory.



NOTE

Refer to [Distributing .NET Assemblies with .NET Client Applications](#) for more information about distributing configuration files.

Using .NET Assemblies with LabVIEW

An assembly is a unit of deployment similar to a DLL, OCX, or executable for a component in COM. Assemblies are DLLs and executables you build using a .NET compiler. Assemblies can consist of a single file or multiple files. An assembly includes a manifest that contains information about the assembly name, version, publisher security, files that make up the assembly, dependent assemblies, resources, and exported data types. Single-file assemblies contain all the data in a single file, including the manifest and any resources it needs. Multi-file assemblies might have external resources, such as bitmaps, icons, and sound files, or have one file for the core code and another for helper libraries.

Assemblies can be public or private. Private assemblies are usually located in the same directory as the application. With .NET Framework, public assemblies are located in the Global Assembly Cache (GAC). The developer of an application typically writes private assemblies for use by that application. The developer of the assembly also decides the

version of the assembly. The assembly name is the filename, minus any file extension, of the file that contains the manifest.

For .NET Framework, the Common Language Runtime (CLR) uses [configuration settings](#) to determine the assembly version to load for an application. These configuration settings might override your request for a specific assembly version. For example, the system administrator can configure your computer to load version 1.0.0.1 instead of 1.0.0.0. If you then try to load version 1.0.0.0, the CLR promotes the assembly version to 1.0.0.1. LabVIEW also notifies you of the promotion.

Loading .NET Framework Assemblies



NOTE

The following content applies to .NET Framework only. LabVIEW has been gradually adding support for .NET Core since LabVIEW 2024 Q3. Refer to *LabVIEW New Features and Changes* for specific .NET Core support added in different releases.

If you reference a .NET object from the front panel or block diagram of a VI, ensure that LabVIEW can load the .NET assembly for that object. The Common Language Runtime (CLR) is responsible for locating .NET Framework assemblies that you call. Refer to the Microsoft Developer Network (MSDN) Web site for more information about how the CLR locates assemblies. If the CLR cannot find the assembly, LabVIEW then searches for the assembly in the same manner it searches for missing VIs. LabVIEW searches for missing VIs in the directories you specify on the Paths page of the Options dialog box. If LabVIEW cannot find the .NET assembly for a .NET object referenced directly on the front panel or block diagram, LabVIEW generates a load-time error. If LabVIEW cannot load a dependent assembly needed during run time, LabVIEW generates a run-time error.

The CLR uses the directory of the running executable as the default search path when it loads private .NET assemblies. If you reference a .NET object from a VI that does not belong to a LabVIEW project, the CLR considers LabVIEW .exe to be the running executable. The CLR therefore searches for private assemblies in the directory in which the LabVIEW .exe file is located. If you reference a .NET object from a VI that does belong to a LabVIEW project, the CLR considers the project to be the running executable. The CLR therefore searches for private assemblies in the project directory. If you reference a .NET assembly from a VI and the assembly does not belong to the .NET, NI strongly recommends that you store the VI in a project to avoid having to place files in the directory in which the LabVIEW .exe file is located.

If you call a .NET Framework assembly from a VI that does not belong to a project, you technically can save the assembly in the same directory as its calling VI. LabVIEW searches certain VI directories, including the calling VI directory, for assemblies that the CLR cannot load by default. However, calling assemblies stored in this location can result in name conflicts and other unexpected .NET Framework behavior. Therefore, NI does not recommend that you save assemblies in this location.

Detecting Changes to Assemblies in Memory

After LabVIEW loads an assembly into memory, the assembly stays in memory until you close the [application instance](#) that loaded the assembly. While an assembly is in memory,

LabVIEW does not detect changes that you make to the assembly on disk. Therefore, before LabVIEW can access any changes to the assembly, you must *update the assembly version in memory*.

Loading VIs after Changing an Assembly

Microsoft Visual Studio .NET and other development tools provided in the .NET SDK can assign strong names to an assembly. Assemblies with the same strong name are expected to be identical.

When you load a VI with a change in the path of a .NET assembly or with a change in the version number or culture string of a strong-named assembly, LabVIEW launches a warning dialog box informing you of the change. Once loaded, the VI includes an asterisk in its title bar and in the list of open VIs displayed in the **Window** menu. When you save the VI, the asterisk disappears until you make a new change.

When you load a VI with a change in the time stamp of a .NET Framework assembly, LabVIEW does not launch a warning dialog box but does display an asterisk in the title bar of the VI.



TIP

Verify that LabVIEW is loading the correct version of your assembly by referring to the .NET Assemblies in Memory dialog box.

Refer to the KnowledgeBase at ni.com for more information about how to select and load specific versions of .NET Assemblies.

Distributing .NET Assemblies with .NET Client Applications

After you create a .NET client application in LabVIEW, you can *build that project or VI* into a stand-alone application, shared library, packed project library, source distribution, or ZIP file. You then can distribute the application to another computer as long as you also supply the relevant assembly and configuration files that define the characteristics and behaviors of the .NET objects in the application.



NOTE

.NET 8.0 (Core) Considerations

LabVIEW only identifies assemblies as dependencies that are referenced directly from VIs. If you need additional assemblies, you have to add them manually.

**NOTE****.NET Framework Considerations**

NI recommends that the target computer for the distributed application have the same or later version of .NET Framework installed as the version that LabVIEW used to build the application. Refer to [Compatible Class Libraries](#) for more information about the .NET Framework requirements of the target computer.

You must distribute the following files along with any application that you build from a .NET Framework client application:

**NOTE**

You can [build an installer](#) for your distributed application that includes the relevant files.

- data directory of the application—When you create an application, LabVIEW automatically copies any necessary .NET Framework assemblies that were not located in the Global Assembly Cache (GAC) into this directory.
- Private .NET Framework assemblies that are located in the GAC—LabVIEW does not copy these assemblies into the data directory, so you must distribute them yourself.
- [.NET configuration file](#)—If a project or VI relies on a .NET Framework configuration file, any application that you build from that project or VI requires a similar configuration file. You do not need to distribute a configuration file in the following situations:
 - The original project or VI does not require a configuration file.
 - You are building a .NET Framework interop assembly.

Where to Save Distributed Files on a Target Computer

For a distributed .NET Framework application to run on a target computer, the relevant files must be saved in the following locations on the target computer:

- data directory—Store this directory in the same directory as the application.
- Private .NET Framework assemblies from the GAC—Store the assemblies in the GAC or in another location that the CLR searches.
- .NET Framework configuration file—The location for the configuration file depends on what type of application you are distributing.
 - Stand-alone application—Store the configuration file in the same directory as the application.
 - Shared library, packed project library, source distribution, or ZIP file—Store the configuration file in the directory of the application that is using the shared library, packed project library, source distribution, or ZIP file.

Touch Panel in LabVIEW

Creating a Project Using the Touch Panel Project Template

Complete the following steps to use the Touch Panel Project template to create a new project.

1. Select **File»Create Project** to display the Create Project dialog box.
2. Search for the **Touch Panel Project** template and click **Next**.
3. On the **Configure the new project** page, complete the following steps to configure details of the project.
 1. Specify the name of the **.lvproj** file and directory to save the project in **Project Name** and **Project Root**, respectively.
 2. (Optional) In the **Modify VI Icons** section, edit the appearance of VI icons.
 3. Click **Browse** to open the Add Targets and Devices dialog box. Select the touch panel target to add to the project and click **OK**.
 4. Select a *touch panel VI template* from the **VI template** pull-down menu.
4. Click **Finish** to save the project and open the Project Explorer window.

The project appears in the Project Explorer window. Double-click the VI under the target to open the front panel and block diagram windows. Refer to the comments on the block diagrams of VIs in the project for more information about how to modify the project. Also refer to the **Project Documentation** folder in the **Project Explorer** window for more information about how to modify the project.

Navigation Controls

Use the navigation controls located on the System and Modern palettes to navigate through multiple front panels in an application. Navigation controls include the **Previous Button**, **Next Button**, and **Jump Button** controls.

Previous Button and Next Button Navigation Controls

The **Previous Button** and **Next Button** controls navigate through and display the front panels of VIs in a project library. The controls navigate through the VIs in a project library

based on the order the VIs appear in the project library. If you want to change the order of the VIs, you must [change the order in the project library](#).



NOTE

You must [reference a VI in a LabVIEW project library](#). `lvlib` to use the **Previous Button** and **Next Button** controls. When you add a **Previous Button** or **Next Button** control to the front panel of a VI that is not referenced by a LabVIEW project library file, the control displays an error glyph.

When a user selects the **Next Button** control for the last VI in a project library, the **Next Button** control displays the front panel for the first VI listed in the project library. You only can access VIs in the same project library with these controls.



Tip Select **File»New** to display the New dialog box, and select **Other Files»Multi-panel Application** to display a template that uses the **Previous Button** and **Next Button** controls.

You can set the string text on the **Previous Button** and **Next Button** programmatically by using the **Button Text** property. Right-click a navigation control from the front panel, select **Create»Property Node**, and select the **Button Text** property from the shortcut menu. LabVIEW creates the Property Node on the block diagram. The Property Node links implicitly to the navigation control.

Jump Button Navigation Control

You can use the **Jump Button** control to navigate to a specific VI. When you add a **Jump Button** control to the front panel, the Configure Jump Button dialog box appears. Use this dialog box to select the target VI and the text to display on the button. You can specify any VI on disk as the target VI, even if the VI is not referenced by a project library. You may want to use a **Jump Button** control if you have a VI that functions as the main panel in an application or references multiple project libraries.



NOTE

Use the **Button Text** property and **Target VI** property to set the string text and target VI, respectively, on the **Jump Button** programmatically.

Protecting Data Using Write Filters

Write filters protect data from unwanted modifications by redirecting write operations to another location or overlay. Use the Enhanced Write Filter (EWF) VIs to protect volumes and redirect writes to a disk location on another volume or to the RAM. Use the File-Based Write Filter (FBWF) VIs to protect files and folders in a volume and redirect writes to the memory cache. The Write Filters VIs are available for VIs under a [Touch Panel target](#). Refer to the MSDN Web site at `msdn.microsoft.com` to learn more about write filters.

LabVIEW supports the Write Filters VIs, which you use to enable, disable, and configure write filters, for touch panel devices. You must target a touch panel device to use the Write Filters VIs.

Supported Touch Panel Targets

LabVIEW supports the following touch panel targets:



NOTE
The LabVIEW Run-Time Engine is required to run touch panel applications.

Target	Processor	Use
NI TPC-2215	x86	Use with NI TPC-2215 hardware
NI TPC-2212	x86	Use with NI TPC-2212 hardware
NI TPC-2206	x86	Use with NI TPC-2206 hardware
Windows Embedded Standard Device	x86	Use with third-party touch panel computers running Windows Embedded Standard



- NOTE**
- LabVIEW supports Windows Embedded Standard 7 only.
 - Windows Embedded Standard is a componentized version of Windows, which allows you to customize the operating system image. National Instruments tests and certifies LabVIEW to work with the Windows Embedded Standard OS image that ships with National Instruments touch panel computers. If you use third-party touch panel hardware with Windows Embedded Standard and you do not install the correct components, the LabVIEW application may not function properly. Please contact National Instruments technical support at ni.com/support if you experience unexpected behavior executing a LabVIEW touch panel application with a third-party Windows Embedded Standard touch panel computer. However, National Instruments does not officially support compatibility issues with third-party Windows Embedded Standard touch panel computers.
 - National Instruments touch panel computers come with an English version of the Windows Embedded Standard operating system. The English version supports VI front panels that use Western European languages, including English, French, and German. For support in other languages, contact National Instruments technical support at ni.com/support.

Using Touch Panel VI Templates

Use a touch panel VI template as a starting point for developing touch panel VIs. LabVIEW includes portrait and landscape templates whose user interfaces are preset to the touch panel device. These templates include controls and indicators that you commonly use in touch panel applications.

You can add a touch panel VI template by right-clicking a touch panel target in the Project Explorer window and selecting **New VI Template**. The **Select a template** dialog box appears. Select the VI template to use with the target.

LabVIEW How To

This section provides instructions for performing various tasks with LabVIEW.

Setting up LabVIEW

About LabVIEW Evaluation and Activation (Windows)

Use the LabVIEW Platform media to install LabVIEW and many LabVIEW modules and toolkits and evaluate the products for free for a set period of time. This topic explains what you can expect when you evaluate LabVIEW, modules, or toolkits and how you can activate those products after you evaluate them.

Evaluating LabVIEW, Modules, or Toolkits

You can access all the tools, VIs and functions, help, and examples available to the LabVIEW Professional Development System and most modules or toolkits in a free evaluation mode. When the evaluation period expires, you must activate the software to continue using LabVIEW.

Evaluating LabVIEW

When you run LabVIEW in evaluation mode, LabVIEW includes the following limitations:

- **A time limit until the evaluation version expires** —While not activated, LabVIEW prompts you to activate the product each time you launch LabVIEW. You can continue to run LabVIEW in evaluation mode.
- **An evaluation version watermark during the time limit** —All user VIs and controls have an evaluation watermark and any VIs you create while in non-activated mode, except for subVIs in a subpanel of a VI, include a watermark on all displayed front panel windows.

After the evaluation period for LabVIEW expires, you are no longer able to launch LabVIEW.

Evaluating Modules and Toolkits

After the evaluation period for a module or toolkit expires, you are no longer able to perform the following actions:

- Run VIs from the module or toolkit.

- Run VIs that use VIs from the module or toolkit.
- Build VIs that use VIs from the module or toolkit into an executable or shared library.
- Access menus, tools, or palettes installed by the module or toolkit.

When you install a module or toolkit for evaluation, you install the help for the module or toolkit. The help for the module or toolkit remains installed after the evaluation period. Also, the NI Example Finder displays all module or toolkit examples after the evaluation period. You must uninstall the module or toolkit to remove the help and examples.

If you run LabVIEW in evaluation mode, you receive functionality of the Professional Development System. If you then purchase and activate a LabVIEW license, you receive full functionality of the development system you purchase. For example, if you create a VI using features that are only available in the Professional Development System and then you purchase and activate the Full Development System, the VI you created using the Professional Development System will be broken.

Activating LabVIEW, Modules, or Toolkits after Evaluation

To activate the LabVIEW license, use the serial number you received on the Certificate of Ownership included in the software kit and follow the activation instructions.

If you purchase add-on software, such as modules or toolkits, use NI License Manager to activate the software. If you do not see the software listed in NI License Manager, you can install it from the LabVIEW Platform media.

LabVIEW installs the Application Builder when you install LabVIEW. If you activate the LabVIEW Professional Development System, you automatically activate the Application Builder. If you activate the LabVIEW Base Development System or Full Development System, you must purchase and activate the Application Builder separately.

Upgrading from an Evaluation Version to a Paid Version

If you are upgrading from an evaluation version to a paid version, you do not need to reinstall your software. Follow the prompts that appear when you launch your application or use NI License Manager to activate the software.

Moving between Paid Versions

If at a later time you decide to move to a different product package, such as upgrading from LabVIEW Base Development System to LabVIEW Professional Development System, you can activate the new package without installing additional software. Use NI License Manager to activate your software.

Configuring Your LabVIEW User Name

Complete the following steps to set or change your LabVIEW user name.

1. Select **Tools»User Name** to display the User Login dialog box.
2. By default, the operating system user name is selected. To change your LabVIEW user name, select a name from the pull-down menu. You also can select **other** from the pull-down menu and type a new user name in the **Username** text box.
3. Click the **OK** button.

Logging In and Logging Out of LabVIEW (Windows)

Complete the following steps to log in to LabVIEW.

1. Select **Tools»Security»Login** to display the NI Security Login dialog box.
2. Enter the user account name in the **User name** text box.
3. Enter the account password in the **Password** text box.
4. Select the security domain in which *the user account is defined* from the **NI domain** listbox.



NOTE

If there are no security domains to choose from, you can define one using the [Domain Account Manager](#).

5. Click **OK** to log in.

To log out of LabVIEW, select **Tools»Security»Logout**.



Note When no one is logged in, LabVIEW uses a default nobody account to define the access rights for the user. There is no password assigned to the nobody account.

Creating your project

Creating a LabVIEW Project

Use [projects](#) to group together LabVIEW files and files not specific to LabVIEW, create build specifications, and deploy or download files to targets.

You must use a project to build applications and shared libraries. You also must use a project to work with a Windows Embedded Standard, RT, or FPGA target. The LabVIEW Real-Time and FPGA Modules add functionality to the LabVIEW development system and must be purchased separately.

Complete the following steps to create a project:

1. Select **File»Create Project** to display the Create Project dialog box. The **Create Project** dialog box includes a list of templates and sample projects you can use to ensure that the project you create uses reliable designs and programming practices.

2. Search for the template or sample project that supports the goal you want the project to accomplish. Use the following features to search for templates and sample projects:
 - **Filters** —Displays only results of a certain type, such as templates or sample projects for a specific target.
 - **Additional Search** —Searches the keywords, titles, and descriptions of the filtered results.
 - **More Information** —Opens the help file for the item. Review the help file to ensure that the template or sample project supports the goal you want the project to accomplish.
3. Click the **Next** or **Finish** button to configure details of the project, including how the project is saved. After you finish configuring the project, LabVIEW saves the project and opens the Project Explorer window.
4. Use the **Project Explorer** window to modify the project. Refer to the comments on the block diagrams of VIs in the project for more information about how to modify the project. Also refer to the **Project Documentation** folder in the **Project Explorer** window for more information about how to modify the project.

You also can use the Project:New method to create a blank new project programmatically.



NOTE

You can build custom templates and sample projects and use them to develop LabVIEW projects. Refer to the KnowledgeBase for more information on how to make custom templates and sample projects accessible from the Create Project dialog box.

Selecting a Control

On the Controls palette, click **Select a Control**, or in the icons view, click the **Select a**



Control icon to add a control to the front panel that does not appear on the **Controls** palette. Select the appropriate .ctl file to add the control to the front panel.

Adding Folders to a Project

Use the **Project Explorer** window to add folders to create an **organizational structure** for items in a **LabVIEW project**.

Adding Auto-Populated Folders to a Project

Adding **auto-populated folders** adds a directory on disk to the project. LabVIEW continuously monitors and updates the folder according to changes made in the project and on disk. A blue folder icon with a yellow cylinder identifies this type of folder.

Complete the following steps to add an auto-populated folder to the project.

1. From the **Items** page, right-click a target or a folder under a target and select **Add»Folder (Auto-populating)** from the shortcut menu to display a file dialog box. You also can select a target and then select **Project»Add To Project»Folder (Auto-populating)**.
2. Navigate to and select a directory on disk to add to the project.
3. Click the **Select Folder** button to add the current folder to the project. An auto-populated folder appears in the project. LabVIEW automatically adds all contents of the folder to the project. If items in the folder on disk are already in the project, the items move within the auto-populated folder.
4. From the **Files** page, you can move, rename, reorganize, and remove files from the auto-populating folder. Project operations from the **Files** page both update and reflect the contents on disk. Likewise, LabVIEW updates the folder in the project if you make changes to the directory on disk. One advantage of making changes to files from the project instead of from disk is if you update files from the LabVIEW project, LabVIEW can detect filename changes and update caller VIs automatically.



NOTE

(macOS and Linux) LabVIEW does not support the use of symbolic links for files in auto-populating folders on macOS or Linux operating systems.

You also can use the Convert to Auto-populating Folder method to add an auto-populating folder to the project programmatically.



Tip You can arrange items on the **Items** page. Right-click a folder and select **Arrange by»Name** from the shortcut menu to arrange items in alphabetical order. Right-click a folder and select **Arrange by»Type** from the shortcut menu to arrange items by file type. Right-click a folder and select **Arrange by»Path** from the shortcut menu to arrange items by file path.

To disconnect an auto-populated folder from disk, right-click the auto-populated folder on the **Items** page and select **Stop Auto-populating** from the shortcut menu. LabVIEW disconnects the folder from the corresponding folder on disk. This option is available only to top-level folders and applies recursively to subfolders of auto-populated folders.

You also can use the Stop Auto-populating method to disconnect an auto-populating folder from disk programmatically.



Note The project displays *project library* contents by library hierarchy and not by disk organization. For example, if a VI exists in a project library file `.lvlib` the VI appears in the project under the library file. The VI does not appear in the project in its location on disk. Similarly, an auto-populated folder cannot contain a virtual folder unless the virtual folder is within a library hierarchy.

Adding Virtual Folders to a Project

A virtual folder is a folder in the project that organizes project items and does not represent files on disk. A silver folder icon identifies this type of folder.

Complete the following steps to add a virtual folder to the project.

1. From the **Items** page, right-click a target or a folder under a target and select **Add»Folder (Snapshot)** from the shortcut menu to display a file dialog box. You also can select a target and then select **Project»Add To Project»Folder (Snapshot)**.
2. Navigate to and double-click a directory on disk to add to the project.
3. LabVIEW creates a new virtual folder in the project with the same name as the directory on disk. LabVIEW also creates project items that represent the contents of the entire directory, including files and contents of subdirectories.



Note A virtual folder does not update in the project if you make changes to the directory on disk.

You can convert a virtual folder to an auto-populated folder. Right-click the virtual folder and select **Convert to Auto-populating Folder** to display a file dialog box. Select a folder on disk to auto-populate with. An auto-populating folder appears in the project. LabVIEW automatically renames the virtual folder to match the disk folder and adds all contents of the disk folder to the project. If items in the directory already exist in the project, the items move within the auto-populating folder. Items in the virtual folder that do not exist in the directory on disk move to the target.

You also can convert virtual folders into [LabVIEW project libraries](#). Right-click a virtual folder under a target and select **Convert to Library** from the shortcut menu.

Create new virtual folders to organize project items by right-clicking a target and selecting **New»Virtual Folder** from the shortcut menu.

You also can use the Add Item method to add a folder to the project programmatically.

Adding Target Folders to a Project

A target folder organizes one or more targets and its items in your LabVIEW project. A yellow folder icon with a silver controller identifies this type of folder.

To add a target folder to a LabVIEW project, you must have a module or driver that supports targets installed.

To add a target folder to the project, right-click the project root in the **Project Explorer** window and select **New»Target Folder** from the shortcut menu. A target folder appears in the **Project Explorer** window.

Adding Hyperlinks to a Project

Use the [Project Explorer](#) window to add hyperlinks to a [LabVIEW project](#). You can use hyperlinks to link to files or directories that are not on the local computer but are accessible on the Internet or your local network. You can add hyperlinks that correspond to network, local, HTTP, FTP, mailto addresses and so on.

Complete the following steps to add a hyperlink to a project.

1. Right-click a target, or a folder or library under the target, and select **Add»Hyperlink** from the shortcut menu. The Hyperlink Properties dialog box appears. You also can select the target and select **Project»Add to Project»Hyperlink** to display the **Hyperlink Properties** dialog box.
2. Enter the hyperlink name in the **Name** text box and the corresponding address in the **Address** text box.
3. Click the **OK** button to save the changes and close the **Hyperlink Properties** dialog box. The hyperlink appears as an item in the **Project Explorer** window.

Double-click the hyperlink or right-click and select **Open** from the shortcut menu to open the hyperlink in a browser window.

You can edit an existing hyperlink by right-clicking it and selecting **Properties** from the shortcut menu.

You also can use the AddItem method and HyperLink Address property to add a hyperlink to a LabVIEW project and set the corresponding name and address programmatically.

Adding Items to a Project

Use the **Project Explorer** window to add LabVIEW files, such as VIs and *libraries*, as well as files not specific to LabVIEW, such as text files and spreadsheets, to a target in a **LabVIEW project**. When you add files to a project, LabVIEW creates items in the **Project Explorer** window to represent the files on disk. You can create an *organizational structure* for items in a project.

An item can appear only once under a target. For example, if you add a file from a directory on disk to the **My Computer** target and then add the entire directory to the **My Computer** target, LabVIEW does not include the file again.

From the **Items** page in the **Project Explorer** window, you can add items under a target in a project in the following ways:

- Right-click a target and select **New»VI** from the shortcut menu to add a new, blank VI. You also can select **File»New VI** or **Project»Add To Project»New VI** to add a new, blank VI. You also can add a VI to the project by selecting the VI icon in the upper right corner of a front panel or block diagram window and dragging the icon to the target. When you add a VI to the project, LabVIEW automatically adds its entire hierarchy to the **Project Explorer** window under **Dependencies**.
- Right-click a target and select **New** from the shortcut menu. Then select the item you want to add to the project.
- Right-click a target and select **New»Virtual Folder** from the shortcut menu to add a virtual folder under a target. A virtual folder is a folder in the project that organizes project items and does not represent files on disk. You also can create subfolders by right-clicking a virtual folder and selecting **New»Virtual Folder** from the shortcut menu.

- Right-click a target or a folder under the target, select **Add»File** from the shortcut menu, and select the file you want to add from the dialog box. You also can select the target, select **Project»Add To Project»File**, and select the file you want to add from the dialog box.
- Right-click a target or a folder under the target, select **Add»Folder (Snapshot)** from the shortcut menu, and select the file(s) you want to add from the dialog box. You also can select the target, select **Project»Add To Project»Folder (Snapshot)**, and select the file(s) you want to add from the dialog box.
- Right-click a target or a folder under a target and select **Add»Folder (Auto-populating)** from the shortcut menu to display a file dialog box. You also can select a target and then select **Project»Add To Project»Folder (Auto-populating)**.
- Right-click a target, or a folder or library under a target, and select **Add»Hyperlink** from the shortcut menu. The Hyperlink Properties dialog box appears. You also can select the target and select **Project»Add to Project»Hyperlink** to display the **Hyperlink Properties** dialog box.
- Use the Add File, Add Item, or Add Item From Memory method to add items to a project programmatically. Use the HyperLink Address property to set the address of a hyperlink item.
- (Windows and macOS) Select an item or directory on disk and drag it to the target.

**NOTE**

You cannot drag and drop items in the **Project Explorer** window to the file system.

You also can add new LabVIEW files to a project from the New dialog box. Select **File»New** or **Project»Add To Project»New** to display the **New** dialog box. In the **New** dialog box, select the item you want to add and place a checkmark in the **Add to project** checkbox. If you have multiple projects open, select the project to which you want to add the item from the **Projects** list.

Items you add to the **Project Explorer** window can include icons.

Adding LLBs to a Project

Complete the following steps to add an **LLB** to a **LabVIEW project** as a folder or as a file in the **Project Explorer** window.

Adding an LLB to a Project as a Folder

1. Right-click a target or a folder under the target and select **Add»Folder (Snapshot)** from the shortcut menu to add an LLB to the project as a folder. A file dialog box appears.
2. Navigate to the LLB you want to add.
3. Click the **Open** button in the file dialog box. The folder appears in the **Project Explorer** window. LabVIEW uses the filename of the LLB to name the folder and adds the VIs in the LLB as items within the new folder.

Adding or **removing** items from the virtual folder in the **Project Explorer** window does not affect the .llb file on disk. Use the LLB Manager window or an **auto-populating folder** to edit an LLB on disk. Right-click a target or a folder under the target and select **Add»Folder (Auto-populating)** from the shortcut menu to add an LLB to the project as an auto-populating folder.

Adding an LLB to a Project as a File

1. Right-click a target or a folder under the target and select **Add»File** from the shortcut menu to add an LLB to the project as a file. A file dialog box appears.
2. Navigate to the LLB you want to add.
3. Click the **Add File** button in the file dialog box. The LLB browser opens.
4. Click the open folder icon labeled **VI** with a single period  in the LLB browser.
5. Click the **Select** button to select the file and close the LLB browser. The file appears in the **Project Explorer** window. When you add an LLB as a file, the VIs in the LLB do not appear in the **Project Explorer** window.

You also can use the Add Item method to add an LLB to a project as a folder or as a file programmatically.

Adding Targets to a Project (Windows)

To add a target to a **LabVIEW project**, you must have a module or driver that supports targets installed.

Complete the following steps to add a target or device to a project.

1. **Create a project.**
2. Right-click the project root and select **New»Targets and Devices** from the shortcut menu to display the Add Targets and Devices dialog box. If a target in the project supports other targets, you also can right-click the target and select **New»Targets and Devices** from the shortcut menu to add a target under the existing target. For example, if you have an NI PCI device installed on a computer, you can add the device under **My Computer**.
3. Select the type of target or device you want to add and click the **OK** button. An item representing the target appears in the **Project Explorer** window.



NOTE

If a target you want to add conflicts with another target, the **Add Targets and Devices** dialog box displays the **Options** column next to the **Targets and Devices** list. Click the down arrow next to the selected conflict in the **Options** column and select from the available options to resolve the conflict. You only can resolve conflicts for one target at a time. LabVIEW dims the **OK** button until you resolve all conflicts in the **Options** column for the selected target.

You can **add items** under a target. You also can use a target folder to organize one or more targets in your LabVIEW project. To add a target folder to the project, right-click the project root in the **Project Explorer** window and select **New»Target Folder** from the shortcut menu. A target folder appears in the **Project Explorer** window.

Finding Items in a Project

Complete the following steps to search for items in the contents tree of an open **LabVIEW project**.

1. From the **Project Explorer** window, select **Edit»Find Project Items** to display the Find Project Items dialog box. To search for LabVIEW project items within a specific folder, project library, LabVIEW class, or XControl, right-click the item in the **Project Explorer** window and select **Find Project Items** from the shortcut menu.
2. Enter the text you want to find in the **Type in text to search for** text box. The title bar of the **Find Project Items** dialog box displays the root item of the search. Searches include all items under this root item.



NOTE

LabVIEW only searches for items that are members of the LabVIEW project, project library, LabVIEW class, or XControl. For example, if the LabVIEW project you want to search contains `Foo.vi`, and `Foo.vi` contains a subVI named `Bar.vi`, you must add `Bar.vi` to the LabVIEW project to return search results for the subVI.

1. (Optional) After you enter the text you want to find, choose from the **Case sensitive**, **Match whole word**, and **Regular expression** options to configure the scope of the search.
2. Click the **Find** button. The **Search results** list contains all instances of the text that LabVIEW finds.



Note If only one item matches the search string, LabVIEW automatically highlights the search result in the **Project Explorer** window and closes the **Find Project Items** dialog box.

1. Select a search result and click the **Go To** button. LabVIEW highlights the item in the **Project Explorer** window. You also can double-click the entry in the **Search results** list to highlight the item in the **Project Explorer** window.
2. (Optional) Select **Edit»Find Next** or **Edit»Find Previous** to highlight other search results in the **Project Explorer** window.

LabVIEW retains the previous search results until LabVIEW or the **Project Explorer** window closes. View the previous search results by selecting **Edit»Show Search Results** to display the **Find Project Items** dialog box and the list of **Previous search results**.

You also can access the **Find Project Items** dialog box from a stand-alone project library window or class window.

Sorting Items in a Project

You can use sort options to organize files within a project. The sort option you select does not reorganize the project on disk, which allows you to merge and compare fewer changes in your project. Sort options apply automatically to any new items that you add to the sorted project.

You also can sort unlocked [project libraries](#) and content within [auto-populating folders](#).

Complete the following steps to sort items in a [LabVIEW project](#).

1. From the **Project Explorer** window, right-click the item you want to sort.
2. Select **Arrange By** and select an available sort category from the shortcut menu.

Use one of the following sort options to sort items in a project:

- **Name** —Sorts items alphabetically by file name.
- **Type** —Sorts items by file type.
- **Path** —Sorts items by path name.
- **Custom** —Sorts items according to user preference. Unlike the other sort categories, **Custom** sort alters the layout of a project on disk.
- **Same As Parent** —Sorts items in the same way as the parent. Sort by **Same As Parent** to let items automatically inherit a sort category from the parent folder. You can sort only items with a parent by the **Same As Parent** category.

These sort categories apply to targets, files, and folders within the project.



NOTE

By default, LabVIEW sorts all new projects and project libraries by **Name**. Sorting by **Name**, **Type**, **Path**, or **Same As Parent** does not alter the organization of the project on disk. However, if you move items in a project without using a sort option, LabVIEW automatically changes to **Custom** sort, which modifies the organization of the project on disk. If you sort a project library by **Type** or by **Path**, LabVIEW automatically re-sorts the project library by **Name** the next time you open the project library.

Removing Items from a Project

Use the [Project Explorer](#) window to remove items from a LabVIEW project. You can remove items in the following ways:

- Right-click the item you want to remove and select **Remove from Project** from the shortcut menu.
- Select the item you want to remove and press the **Remove from Project** button on the Standard toolbar.

- From the **Items** page, select the item you want to remove and press the <Delete> key.

**NOTE**

Items in the project may refer to other items. For example, if you remove a subVI from the project that another VI in the project calls, the subVI will move to **Dependencies**.

You also can use the Delete method to delete the item through a reference from the project programmatically.

Removing Dependencies from a Project

You can remove items under **Dependencies** in a LabVIEW project in the following ways:

**NOTE**

If you previously hid **Dependencies** in the **Project Explorer** window, you must *display the item* again to access it in the **Project Explorer** window.

- Add the item under the target by dragging the item from **Dependencies** to the target. You also can right-click the target, select **Add»File** from the shortcut menu, and select the item you want to add under the target.
- Remove the VI that calls the item, right-click **Dependencies**, and select **Refresh** from the shortcut menu.

You can use the Refresh Dependencies method to refresh **Dependencies** programmatically.

Saving a Project

You can save a **LabVIEW project** in the following ways:

- Select **File»Save**.
- Select **Project»Save Project**.
- Right-click the project root and select **Save** from the shortcut menu.
- Click the **Save All (this Project)** button on the Project toolbar. You also can select **File»Save All (this Project)** or right-click the project root and select **Save All (this Project)** from the shortcut menu.
- Use the Save method to save a project programmatically.

You must save new, unsaved files in a project before you can save the project.



NOTE

Make a backup copy of a project when you prepare to make major revisions to the project or before you upgrade to a new version of LabVIEW.



TIP

To save a project for a previous version of LabVIEW, refer to *Saving for a Previous Version*.



RELATED INFORMATION

[Saving for a Previous Version](#) on page 938

You can save VIs, LabVIEW projects and project libraries for a previous version of LabVIEW.

Selecting an Application Instance

LabVIEW creates an application instance for each target in a [LabVIEW project](#). When you open a VI from the **Project Explorer** window, the VI opens in the application instance for the target. LabVIEW also creates a main application instance, which contains open VIs that are not part of a project and VIs that you did not open from a project. You can select the application instance for a VI by using the application instance shortcut menu.

Complete the following steps to select an application instance for a VI.

1. Navigate to the bottom left corner of the front panel window or block diagram. The application instance selection shortcut menu displays the current application instance of the VI.
2. Right-click the shortcut menu and select the instance in which to run the VI. The VI now runs in the selected application instance.

You also can use the Application:Default:Application property to return the default application reference programmatically. Use the Application property to open the target application instance programmatically.

Setting the Default Palette File for a Project Library

If you include a palette file (.mnu) in a project library, you can set it as the default palette file for all VIs that the project library owns. Complete the following steps to set the default palette file for a [LabVIEW project library](#).

1. From the [Project Explorer](#) window, locate the project library icon above the palette file .mnu you want to set as the default palette file. If you do not have a .mnu file, [add the palette file](#).

2. Right-click the project library and select **Properties** from the shortcut menu to display the Project Library Properties dialog box.
3. On the General Settings page, select the palette file you want to set as default from the **Default Palette** pull-down menu. You also can set the default palette file from the Item Settings page by selecting the .mnu file in the **Contents** tree and place a checkmark in the **Default Palette** checkbox.
4. Click the **OK** button to close the **Project Library Properties** dialog box and save your changes.

You also can use the Library>Create method to set the default palette file programmatically.

Showing and Hiding Items in a Project

You can show and hide the **Dependencies** and **Build Specifications** items in the *Project Explorer* window in the following ways:

- Select **Project»Filter View** and select the item you want to show or hide.
- Right-click the project root in the **Project Explorer** window, select **View** from the shortcut menu, and select the item you want to show or hide.
- Click the **Filter Project View** button on the Project toolbar and select the item you want to show or hide.

Right-click the project root and select **Expand All** or **Collapse All** from the shortcut menu to expand or collapse all items in the **Project Explorer** window. You also can right-click an item in the **Project Explorer** window and select **Expand All** or **Collapse All** from the shortcut menu to expand or collapse the item and all folders within the item.

Viewing Files in a Project

The *Project Explorer* window includes two pages. The **Items** page displays the contents of the project. The **Files** page displays the project items that have a corresponding file on disk. You can switch from one page to the other by right-clicking a folder or item under a target and selecting **Show in Items View** or **Show in Files View** from the shortcut menu.

When you *add a file* to a LabVIEW project, LabVIEW includes a reference to the file on disk. From the **Items** page in the **Project Explorer** window, select **Project»Show Item Paths** to display the **Paths** column and view the file path. You also can right-click the project root and select **View»Full Paths** from the shortcut menu to display the **Paths** column and view the file path. Use the Path property to return the location of a file on disk programmatically.

Use the **Files** page in the **Project Explorer** window to view the disk location of project files. You can organize filenames and folders on the **Files** page and perform disk operations such as rename, reorganize, and remove. For example, if you rename a file on the **Files** page, LabVIEW renames that file on disk. You also can use the Project File Information dialog box to view the disk location of project files. Select **Project»File Information** to display the **Project File Information** dialog box. You also can right-click the project root and select

View»File Information from the shortcut menu to display the **Project File Information** dialog box.

Working with Project Libraries

Creating a Project Library

LabVIEW project libraries are collections of VIs, type definitions, shared variables, palette files, and other files, including other project libraries. When you create and save a new project library, LabVIEW creates a project library file `.lvlib`, which includes the properties of the project library and the references to files that the project library owns.

Complete the following steps to create a project library.

1. From the **Project Explorer** window, right-click **My Computer** and select **New»Library** from the shortcut menu. LabVIEW creates a project library file that appears in the **Project Explorer** window as part of the LabVIEW project. From the Getting Started Window or a blank VI, you also can select **File»New** to display the New dialog box. In the **Other Files** folder of the **Create New** tree, double-click the **Library** option. A stand-alone project library window for the new project library file appears.
2. To add files to a project library, right-click the project library icon and select **Add»File** or **Add»Folder (Snapshot)** from the shortcut menu. You can add the same types of files that you can add to projects. You also can drag an item in the project so it appears under the project library. Only one project library can own a specific VI. However, you can associate files not specific to LabVIEW, such as a text file or HTML file, with multiple project libraries.



NOTE

If you add a VI that is not in memory to a project library, a dialog box appears that prompts you to save changes to the VI. Save changes to the VI so it will link correctly to the owning project library.

1. Select **File»Save All** or right-click the project library icon and select **Save»Save As** from the shortcut menu to name and save the project library file.
2. Right-click the project library icon and select **Properties** from the shortcut menu to display the Project Library Properties dialog box.
3. From the General Settings page, set the version number, **create an icon**, and **assign security settings** to the project library.



Note LabVIEW applies the project library icon you create to the icon of all existing objects in the library. In addition, the icon you create becomes the default icon for any new VIs you create from within the project library. You can create a template icon to edit and use in all VIs that a project library owns.

1. From the Documentation page, enter information for context help for the project library.
2. From the Item Settings page, [set access options](#) for files in the project library.
3. Click the **OK** button to update the project library with the edited settings and close the dialog box.

You also can [create a project library from a project folder](#).

You also can use a [sort option](#) to organize items within an unlocked project library.

Use the CreateOrAddLibrary polymorphic VI in the `labview\vi.lib\Utility\Variable` directory to programmatically add a library to a project or a parent item such as a target, folder, or another library.



TIP

To save your project library for a previous version of LabVIEW, refer to *Saving for a Previous Version*.



RELATED INFORMATION

[Saving for a Previous Version](#) on page 938

You can save VIs, LabVIEW projects and project libraries for a previous version of LabVIEW.

Creating a Project Library from a Project Folder

You can create [LabVIEW project libraries](#) from virtual folders in a LabVIEW project. The new project library owns the items that the folder contained.

From the [Project Explorer](#) window, right-click a virtual folder to convert and select **Convert to Library** from the shortcut menu. LabVIEW converts the folder to a project library, which appears in the **Project Explorer** window with the items the library owns listed under the library.



NOTE

You cannot convert an auto-populated folder into a project library.

You can name the new project library file when you save it. Right-click the project library and select **Save** from the shortcut menu.

Configuring Access Options in Project Libraries

You can configure access settings for items and folders that a [LabVIEW project library](#) owns as public or private. If you set an item as private and [lock the project library](#), the item is not

visible in the project library or in palettes. You cannot use a private VI as a subVI in other VIs or applications that the project library does not own, even if the project library is unlocked.

Determine which items in the project library you want to set as public and which as private. Public items might include palette VIs, XControls, instrument drivers, and other tools you want users to find and use. Private items might include support VIs, copyrighted files, or items you might want to edit later without taking the risk of breaking users' code.

You can set the access of a [project sublibrary](#) file `.lvlib` as private within the owning project library, but when you edit the project sublibrary itself, items the sublibrary owns retain public or private access settings.

Complete the following steps to configure access options in a project library.

1. Right-click the project library icon in the [Project Explorer](#) window or stand-alone project library window and select **Properties** from the shortcut menu to display the Project Library Properties dialog box.
2. From the Item Settings page, click an item in the **Contents** tree to select it. The current access settings for the item appear in the **Access Scope** box. Click one of the following radio buttons in the **Access Scope** box to apply to the item.
 - **Public** —The item is visible when users view the project library. Other VIs and applications can call public VIs.
 - **Community** —This item is visible when users view the project library. Only [friends](#) and VIs within the project library can call community VIs. Designate a VI as a friend of a library to give the VI the ability to call any member of the library that is in community scope.
 - **Private** —The item does not appear visible when users view the project library or palettes if you lock the project library. Other VIs and applications that the project library does not own cannot call a private VI.
 - **Not specified** —This option appears only when you select a folder. The folder does not have access items specified. Access is public. By default, folders in a project library do not have access specified, which means the folders are publicly accessible.



NOTE

If you specify access options for a folder, the access setting applies to all items in the folder and overrides access options for individual items in the folder.



Note You can set individual instances of a polymorphic VI as private and set the primary polymorphic VI as public. The polymorphic VI does not break even though instance VIs are private. Setting instance VIs as private is useful if you want users to access instances only through the polymorphic VI selector, so you can edit instance order without causing user problems.

1. Click the **OK** button to incorporate the changes into the project library and close the dialog box.

Items set as private appear in the **Project Explorer** window with a private icon . If you lock the project library, private items do not appear in the **Project Explorer** window.

You also can use the Source Scope:Set method to configure access options programmatically.

Protecting Project Libraries

You can limit editing permission by locking or password-protecting **LabVIEW project libraries**. When you lock a project library, users cannot add or remove items and cannot view items that you **set as private**. When you assign a password to a project library, users cannot add or remove items or edit project library properties without a password. Users can open the Project Library Properties dialog box, but all dialog box components except protection options are disabled. Users must unlock the project library or enter a password to enable the dialog box components.



NOTE

Adding password protection to a project library does not add password protection to the VIs it owns. You must **assign password protection to individual VIs**.

Complete the following steps to set levels of protection for a project library.

1. Right-click the project library icon in the **Project Explorer** or stand-alone project library window and select **Properties** from the shortcut menu to display the **Project Library Properties** dialog box.
2. From the Protection page, select one of the following options to apply to the project library.
 - **Unlocked (no password)** —Users can view public and private items that the project library owns and can edit the project library and its properties.
 - **Locked (no password)** —Users cannot add or remove items from the project library, edit project library properties, or view private items that the project library owns. For example, if you are developing a project library and do not want anyone to view private files, you should lock the project library.
 - **Password-protected** —Users cannot add or remove items from the project library, edit project library properties, or view private items that the project library owns. Users must enter a password to edit the project library. For example, if you are developing a project library and want only a few people on the development team to have editing permission, set a password for the project library and give the password to those people.
3. If you select **Password-protected**, click the **Enter Password** button to enter the password.
4. Enter a password in the **Authentication** dialog box and click **Verify**.
5. Click the **OK** button to incorporate the changes into the project library and close the dialog box.



Note Applying a password for a project library automatically stores the password in the LabVIEW password cache, which gives you access to make changes to the project library as if you had entered the password. You can remove the password from the cache by restarting LabVIEW or by clearing the password cache from the Environment page of the Options dialog box.

You also can use the Lock State:Set method to configure protection options programmatically.

Converting an LLB to a Project Library

Consider the ways in which you might use an [LLB](#) before you decide whether to convert it to a [LabVIEW project library](#). LLBs and project libraries serve different purposes and are not mutually exclusive.

For example, you might want to convert a self-contained collection of files that serves as an application programming interface (API). Converting an LLB to a LabVIEW project library provides several advantages. You can perform [source control](#) operations on individual items that the project library owns. However, you cannot perform source control operations on individual VIs in an LLB. You also can configure access options on items that the project library owns so they are public or private.

If you convert an LLB to a project library, you lose the ability to set a top-level VI to run automatically when you open the LLB. You cannot mark a VI as top-level in a project library. For example, you might not want to convert an LLB that consists of VIs that behave as a single application, which runs automatically when you open the LLB. Consider organizing this type of LLB in a LabVIEW project instead.

If an LLB consists of a single top-level VI and a collection of subVIs, consider adding the VIs in the LLB to a project library. Save the project library file `.lvlib` inside the LLB with the VIs. In the project library, set the access for the subVIs as private. Organizing the files in this way allows users to run the top-level file in the VI without having access to the subVIs.

Complete the following steps to convert an LLB to a project library.

1. [Convert the LLB to a directory](#).
2. In the [Project Explorer](#) window, right-click a target and select **Add»Folder (Snapshot)** from the shortcut menu to display a file dialog. Navigate to the directory you created in step 1 and click the **Select Folder** button. LabVIEW creates a [virtual folder](#) in the project.
3. Right-click the project folder and select **Convert to Library** from the shortcut menu to [create a project library from the project folder](#).
4. Right-click the project library and select **Save»Save** from the shortcut menu to name and save the new project library file.

Converting VIs in a LabVIEW Project to a Packed Project Library

If you determine that you *need to build a packed project library* for a group of VIs, complete the following steps to convert the VIs for a large application to a packed library.

1. Place all VIs for the application in a LabVIEW Project.
2. Create a project library for each module, or group of VIs that work together. If you use a subVI among multiple modules, duplicate the subVI for each project library or place all shared subVIs in a separate project library.
3. Update VIs that reference subVIs in the project libraries to *avoid incorrect dependencies*.
4. Create a separate LabVIEW project for each project library, adding each project library to a LabVIEW project.
5. For each LabVIEW project, build a packed library with the project library as the top-level file.
6. In the LabVIEW project with all the project libraries, right-click each project library and select **Replace with** from the shortcut menu to *replace each project library* with a packed library.
7. Remove the original project libraries and VIs from the project.



NOTE

If you need to make changes to a file in a packed library, make the change to the VI in the LabVIEW project with that project library. After you save the change, test your changes, rebuild the packed library, and copy the most recent packed library to the LabVIEW project with all the packed libraries.

Editing a VI That Calls a VI in a Packed Project Library

When you edit a VI that calls a VI in a packed project library, you might need to access *VIs exported from a packed library*. Complete the following steps to add exported VIs to a caller VI.

1. Access a VI exported from a packed library in one of the following ways:
 - From the packed library, select an exported VI from the list of files in the library.
 - From the *Project Explorer* window, navigate to the **Dependencies** under **My Computer** and open the packed library folder in the project.
 - From the packed library, *add a new palette* to the **Controls** or **Functions** palette.
2. Drag the exported VI to the block diagram of the caller VI.

Caveats and Recommendations for Packed Project Libraries

The following list describes some of the caveats and recommendations to consider when you build a *packed project library*:

- If a VI calls a packed project library compiled for one target and you *open the VI on another target* that has a different operating system, the packed project library fails to load.
- When you *create a debug build* for a packed library, you can view the block diagram of each VI. Even though you can use the block diagrams to debug issues, you cannot edit the VI or save changes to the packed library. Save changes to a VI from the LabVIEW project with the project library. If you want to update the packed library, you must rebuild the library.
- Include only related VIs in a packed library. When you open a VI in a packed library, all VIs in the packed library load. If you include non-related VIs, load time increases because more VIs need more time to load.
- To link to a palette when you build a packed library, save the palette in the same version, target, and operating system as the packed library.
- LabVIEW includes dependencies of the top-level library in the packed library, but they do not export with the build and are not visible in the packed library. To export the dependencies of a packed library, place the dependencies in the top-level project library before building the packed library.
- If an *exported VI* has a connector pane with inputs or outputs of type definitions or classes, include the type definitions or classes in the top-level project library. When you right-click a project library and select **Replace with** from the shortcut menu, LabVIEW does not replace type definitions and classes, only exported items of the packed library. As a result, VIs in the packed library might break when you replace the project library with the packed library.
- After you create a project library in a LabVIEW project, *create a separate LabVIEW project for the project library* before you build the packed library. If you build the packed library in the same LabVIEW project where you create the project library and you replace the project library with the packed library, you cannot rebuild the packed library.

Additional Project Options

Accessing Recently Opened Files

Complete the following steps to find recently opened files.

1. Select **File»Recent Files** and select a filename *.vi*, *.ctl*, *.vit*, *.ctt* from the submenu that appears. Select **File»Recent Projects** to find recently opened project files *.lvproj*.
2. Click the filename to open it.

The files are listed in chronological order, with the most recently opened file listed first. If the lists are empty, the **Recent Files** and **Recent Projects** menu items are dimmed.

When two or more files have the same filename and reside in different directories, the **Recent Files** or **Recent Projects** list displays the full path.

Disabling the Getting Started Window

When you start LabVIEW, the Getting Started window appears. Complete the following steps to launch LabVIEW without the **Getting Started** window.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Environment from the **Category** list.
3. Place a checkmark in the **Skip Getting Started window on launch** checkbox.
4. Click the **OK** button.

Recovering Backup Files Manually

The Automatic Recovery Error dialog box appears when LabVIEW encounters an error and cannot recover automatically saved files.

Complete the following steps to attempt to recover the backup files manually.

1. Copy the backup files to a safe location on your computer.
2. Click the **Delete Backup Files** button to remove the backups from the LabVIEW default data directory and continue launching LabVIEW. To locate the default data directory select **Tools»Options»Paths** and select the default data directory in the drop down menu.
3. In LabVIEW, open the files you copied in step 1 by selecting **File»Open** and navigating to the copied files.

LabVIEW File Extensions

The following table includes common file types that LabVIEW creates and uses to save data.

File Extension	Description
.aliases	File that maps computer names to IP addresses. For example, My Computer typically maps to localhost or the actual IP address of the host computer running LabVIEW. LabVIEW creates the .aliases file automatically when you create a LabVIEW project. LabVIEW creates a new .aliases file each time you open the project. If you build an application, the .aliases file is copied into the same directory as the application. You do not need to check the .aliases file into source control unless you extend the mappings in the file. For more information, refer to <i>Using Source Control in LabVIEW</i>.  NOTE Deleting the .aliases file does not affect the project.
.bin3	File to include example VIs in NI Example Finder For more information, refer to <i>Preparing Example VIs to Appear in the NI Example Finder</i>.
.ctl	Control
.ctt	Control template
.llb	File that groups multiple VIs, controls, or libraries together in a specialized and compressed format. You can group several VIs together and save them in an LLB. Unlike a packed project library (.lvlibp) in which the files are not editable, the files in an LLB are editable. For more information, refer to <i>Creating LLBs</i>.
.lvclass	LabVIEW Class library. When you create and save a LabVIEW class, LabVIEW creates a class library file that defines a new data type. For more information, refer to <i>Creating LabVIEW Classes</i>.
.lvlib	LabVIEW project library. LabVIEW project libraries are collections of VIs, type definitions, shared variables, palette files, and other files, including other project libraries. For more information, refer to <i>Using Libraries in LabVIEW Projects</i> and <i>Understanding Shared Variable Technology</i>.
.lvlibp	LabVIEW packed project library. LabVIEW packed project libraries are project libraries that package multiple files into a single file. The top-level file of a packed library is a project library. For more information, refer to <i>Using Packed Project Libraries in LabVIEW Projects</i>.
.lvnm	The text-based measurement file is a tab-delimited text file you can open with a spreadsheet application or a text-editing application.
.lvproj	LabVIEW project For more information, refer to <i>Managing a Project in LabVIEW</i>.

File Extension	Description
.lv1ps	File that stores project settings that are specific to the local machine. You should not check .lv1ps files into source control because .lv1ps files contain settings specific to the computer you are using; for example, .lv1ps files contain the local source code control configuration. LabVIEW saves .lv1ps files when you save a project, and correctly renames the file when you rename a project. You can remove or delete .lv1ps files without affecting the performance or behavior of a project because .lv1ps files only contain project settings specific to the local machine. If you build an application, LabVIEW does not copy the .lv1ps file into the built application.  NOTE Deleting the .lv1ps file does not affect the project.
.mnu	Palette For more information, refer to <i>Creating Subpalettes in a Palette Set</i>.
.rc	Configuration file used to store Property Node information for IVI Instrument Drivers and define refnum data types for instrument drivers.
.rtm	Run-time menu For more information, refer to <i>Customizing Menus</i>.
.tdm/.tdms	TDM/TDMS file For more information, refer to <i>TDM/TDMS Files</i>.
.vi	Virtual instrument For more information, refer to <i>Anatomy of a VI and Common VI Objects</i>.
.vim	Malleable VI For more information, refer to <i>Malleable VIs</i>.
.vit	Virtual instrument template For more information, refer to <i>Creating Template VIs</i>.
.xctl	XControl For more information, refer to <i>Using XControls</i>.
.xnode	File used internally by LabVIEW. You cannot modify .xnode files, so .xnode files are locked in the Project Explorer window.



RELATED INFORMATION

[Real-Time Module on NI Linux Real-Time Targets](#)

[Using Source Control in LabVIEW](#) on page 384

[Preparing Example VIs to Appear in the NI Example Finder](#) on page 943

[Creating LLBs](#) on page 933

[Creating LabVIEW Classes](#) on page 565

[Using Libraries in LabVIEW Projects](#) on page 370

[Understanding Shared Variable Technology](#) on page 610

[Using Packed Project Libraries in LabVIEW Projects](#) on page 374

[Managing a Project in LabVIEW](#) on page 365

[Creating Subpalettes in a Palette Set](#) on page 841

[Customizing Menus](#) on page 1407

[TDM/TDMS Files](#) on page 883

[Anatomy of a VI and Common VI Objects](#) on page 251

[Malleable VIs](#) on page 211

[Creating Template VIs](#) on page 979

[Using XControls](#) on page 174

Renaming Files and Project Items

You must follow different steps to rename different types of files or project items in LabVIEW. Use one of the following ways to rename a file or project item:

- **VIs** —Select **File»Save As** to open the Save As dialog box. Select the **Rename** option in the **Save As** dialog box. You also can right-click a VI and select **Rename** from the shortcut menu to rename the VI. This option closes and deletes the original VI and opens the VI with the new name, so only the new VI exists on disk and in memory.
- **LabVIEW project libraries, XControls, and LabVIEW classes** —Right-click a project library, an XControl, or a LabVIEW class in the **Project Explorer** window and select **Save As** from the shortcut menu to open the **Save As** dialog box. Select the **Rename** option in the **Save As** dialog box. This option closes and deletes the original file and opens the file with the new name, so only the new file exists on disk and in memory. You also can right-click a project library, an XControl, or a LabVIEW class and select **Rename** from the shortcut menu to enter the new name.
- **LabVIEW projects** —In the [Project Explorer](#) window, select **File»Save As** to open the Save As dialog box. Select the **Rename** option in the **Save As** dialog box.
- **Shared Variables** —In the **Project Explorer** window, right-click the shared variable, select **Rename** from the shortcut menu, and enter the new name for the shared variable. You also can double-click the shared variable in the **Project Explorer** window to display the Shared Variable Properties dialog box and enter the new name for the shared variable in the **Name** text box.

- **Virtual Folders** —In the **Project Explorer** window, right-click a virtual folder, select **Rename** from the shortcut menu, and enter the new name for the folder.
- **Build specifications** —Right-click the build specification in the **Project Explorer** window and select **Properties** from the shortcut menu to display the properties dialog box for the build specification. Edit the **Build specification name** text box, which usually is located on the default page of the dialog box, depending on the type of build specification. For example, you can rename a source distribution build specification from the Information page of the Source Distribution Properties dialog box.

**NOTE**

If you previously hid **Build Specifications** in the **Project Explorer** window, you must *display the item* again to access it.

Customizing LabVIEW

Setting Font Styles

To set the font characteristics for individual objects on the front panel or block diagram, select the object, and select **Font Dialog** from the **Text Settings** pull-down menu on the toolbar to display the **Selection Font** dialog box. This dialog box includes the same options as the Font Style dialog box.

To set the font characteristics for all new objects on the front panel or block diagram, deselect any objects that you have selected, and select **Font Dialog** from the **Text Settings** pull-down menu on the toolbar to display the **Panel Default Font** dialog box. This dialog box includes the same options as the Font Style dialog box.

To set the font characteristics for all new objects in LabVIEW, LabVIEW dialog boxes, or LabVIEW menus and palettes, complete the following steps:

1. Select **Tools»Options** to display the Options dialog box.
2. Select Environment from the **Category** list of the **Options** dialog box.
3. In the **Fonts** section, select which type of text you want to manipulate from the **Font Chooser** pull-down menu.
4. Remove the checkmark from the **Use default font** checkbox.
5. Click the **Font Style** button to display the Font Style dialog box.

Selecting Custom Colors

Complete the following steps to select a custom color.

1. In the **Tools** palette, click the foreground or background color box of the Coloring tool. A color picker appears. You also can access the color picker by right-clicking an object, a front panel pane, or block diagram background with the Coloring tool.

**TIP**

Press the <Shift> key and right-click to display a temporary version of the **Tools** palette at the location of the cursor.

2. Click the **More Colors** button in the bottom right corner of the color picker to select a specific color using the color utility on the operating system.
3. Enter values between 0 and 255 in the **Red**, **Green**, and **Blue** fields. You can combine red, green, and blue in various proportions to obtain any color in the visible spectrum. The higher the value, the more intense the color. For example, if you enter 255 in the **Red** field and 0 in the **Green** and **Blue** fields, you create an intense red.

You also can select custom colors from a color box constant. If you are using a color box constant, use the Operating tool to click the color box constant and display the color picker, and proceed to step 2.

Configuring User-Defined Colors

Color the front panel background and objects with the user-defined colors in the **color picker** to maintain a consistent color scheme across the VIs in an application.

You can change the color associated with a user-defined color name, and you can define new colors to display in the color picker. However, if you change the color associated with a user-defined color name, LabVIEW does not apply that change to any objects on the **Controls** palette. For example, if you change the LED On color to a bright blue instead of the default bright green and you add an LED control to the front panel, the On color of the control is the default bright green.

Complete the following steps to change the user-defined colors that appear in the color picker or to define new colors to display in the color picker.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Environment from the **Category** list.
3. In the **Colors** section, remove the checkmark from the **Use default colors** checkbox to change any of the colors associated with the color names listed under the checkbox. Click the color box next to the color name to display the color picker and select a new color.
4. Click the **Custom Colors** button to display the User Colors dialog box.
5. Configure the user-defined colors in any of the following ways:
 - To define a new color, enter a name in an empty color name text box and click the color box next to the new name to select a color to associate with that name. You must name user-defined colors to display them in the color picker.
 - To change the color associated with a color name, click the color box next to the color name to display the color picker and select a new color.

- To remove a user-defined color from the color picker, delete the color name.
 - To restore the default user-defined colors, delete the color names from every text box in the dialog box.
6. Click the **OK** button twice to save the changes.

Viewing System Colors

The system colors vary across computers, so any VI that uses system colors reflects the specific system colors of the current computer. The system colors define the appearance of **Panel & Object**, **Window**, and **Highlight**, as well as the color of the text that appears on top of these system colors.

To view the available system colors:

1. Select **Tools»Options** to display the Options dialog box.
2. Select Environment from the **Category** list.
3. In the **Colors** section, remove the checkmark from the **Use default colors** checkbox.
4. Click one of the color boxes to display the color picker. You can view the available system colors in the lower right corner of the color picker.



TIP

To change the system colors that appear in the color picker, use the color utility on the operating system of the computer.

Refer to ni.com for more information about system colors.

Setting the Color in a Color Box

A color box is a numeric control or indicator that displays a color that corresponds to a specified value. For example, you can use color boxes to indicate different conditions, such as out-of-range values. The color value is expressed as a hexadecimal number with the form RRGGBB. The first two digits control the red color value. The second two digits control the green color value. The last two digits control the blue color value.

Complete the following steps to set the color of the color box.

1. Click the color box with the Operating tool to display the *color picker*. You also can right-click the color box with the Coloring tool to display the color picker.



NOTE

Click the color box with the Coloring tool to apply the most recent color you selected.

1. Select a color from the color picker. Alternately, click the **More Colors** button in the bottom right corner of the color picker to *select from additional or custom colors* using the color utility on the operating system.

Setting LabVIEW Options

Complete the following steps to customize LabVIEW.

1. Select **Tools»Options** to display the Options dialog box.
2. Select a category of options from the **Category** list.
3. Set the options you want to change.
4. Click the **OK** button.



NOTE

Changes to some options take effect the next time you start LabVIEW.

Adding a Directory to VI Search Path

LabVIEW first searches for a VI in memory. If the VI is in memory, LabVIEW opens that version. If the VI is not in memory, LabVIEW uses the VI search path to locate the VI. If LabVIEW cannot find the VI, you can select a VI of any name from the file dialog box. LabVIEW replaces all instances of the missing VI with the VI that you select. If you want to open a different version of a VI already in memory, the versions must have different names.

Complete the following steps to add a new directory to the VI search path.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Paths from the **Category** list.
3. Select **VI Search Path** from the **Path Chooser** pull-down menu.
4. Remove the checkmark from the **Use default** checkbox.
5. Use the **Browse** button, which displays a file dialog box, or the pull-down menu to the right of the **Browse** button, which contains a list of paths, to navigate to the directory that the LabVIEW development environment searches.
6. Click the **Select Folder** button to select the directory.
7. After you select the directory, LabVIEW inserts the path in the text box to the right of the **Browse** button. You can edit the path in this text box. When you select a path, LabVIEW normally searches that directory but not its subdirectories. You can make the search hierarchical by appending an asterisk (*) to the new path item. For example, enter LabVIEW/*, LabVIEW*, or LabVIEW:* to search LabVIEW and its subdirectories.
8. Use the **Insert Before**, **Insert After**, or **Replace** buttons to add and move directories in the **Path List**. Click the **Remove** button to place the removed path in the text box in case you decide to re-insert it elsewhere in the list.

- Click the **OK** button when you finish adding a new directory to the search path. Changes to the VI search path take effect when you restart LabVIEW.

Restoring Default Menu Shortcuts

If you changed default shortcuts for menu items, you can restore the default LabVIEW menu shortcuts. Complete the following steps to restore the default menu shortcuts.

- Select **Tools»Options** to display the Options dialog box.
- Select Menu Shortcuts from the **Category** list.
- In the **Menu Item** list, select the menu item for which you want to restore the default shortcut.
- Click the **Restore** button to restore the default LabVIEW shortcut for the selected menu item.
- To restore the default LabVIEW shortcuts for all menu items, click the **Restore All** button.
- Click the **OK** button.

Setting Menu Shortcuts

Complete the following steps to customize keyboard shortcuts for edit-time menu items.

- Select **Tools»Options** to display the Options dialog box.
- Select Menu Shortcuts from the **Category** list.
- In the **Menu Item** window, select the menu item for which you want to create a shortcut.
- Select the **Shortcut** text box and press the combination of keys on your keyboard that you want to assign to the selected menu item as a shortcut.
- Click the **Set** button to apply the change.
- Click the **OK** button.

Press the combination of keys on your keyboard that you want to assign to the selected menu item as a shortcut. When you press <alphanumeric key> or <alphanumeric key-Shift>, LabVIEW automatically appends <Ctrl> as a modifier key. (macOS) LabVIEW automatically appends the <Command> as a modifier key. For function keys <F1> to <F24>, LabVIEW does not automatically append <Ctrl> as a modifier key, but you can use these function keys with <Ctrl> as a modifier key. Currently, LabVIEW does not support key combinations with <Alt> as shortcuts.

If you select a key combination that LabVIEW reserves as a shortcut, a warning appears and LabVIEW disables the **Set** button. If you select a key combination assigned to another menu item, a conflict warning appears and displays the menu item that already has that keyboard shortcut, but the **Set** button remains enabled. You also can [restore](#) the default LabVIEW menu shortcuts.

LabVIEW does not respond to keyboard shortcuts for dimmed menu items.



NOTE

(macOS) Keyboard shortcuts using the function keys work only if Exposé does not assign the function keys as shortcut keys. LabVIEW does not support <VolumeUp>, <VolumeDown>, and function keys <F15> to <F24> as shortcut keys on macOS.

Setting Path Options

Complete the following steps to specify the order of the directories LabVIEW searches.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Paths from the **Category** list.



NOTE

The format of pathnames—the use of colons, slashes, or backslashes—differs slightly on different platforms.

1. Select the path directory you want to view or edit from the pull-down menu. You also can add a directory to a [VI search path](#).
2. Remove the checkmark from the **Use default** checkbox to access the path preferences options.
3. Enter a new path in the text box or click the **Browse** button to select a directory.
4. Click the **OK** button.



Note All changes to the **Temporary Directory**, **Default Directory**, and **Default Data Directory** options take effect the next time you start LabVIEW.

Changing Passwords (Windows)

Complete the following steps to change the password for the current (logged in) user.

1. Select **Tools»Security»Change Password** to display the NI Security Change Password dialog box.
2. In the **Old password** text box, enter the password for the current (logged in) user.
3. In the **New password** text box, enter the new password for the user.
4. In the **Confirm password** text box, re-enter the new password for the user.
5. Click the **OK** button to change password.

Customizing your Palette

Adding Palette Categories

In addition to the built-in palette categories, you can create other categories to organize palette items. Complete the following steps to add a category to the top level of the **Controls** or **Functions** palette.

1. Create a new folder for the category you want to create in the `labview\menus\Controls` directory for the **Controls** palette or the `labview\menus\Categories` directory for the **Functions** palette.
2. Add items that you want to include in the category to the new folder. Items for a category on the **Functions** palette can include VIs, functions, LLBs, and `.mnu` files. Items for a category on the **Controls** palette can include controls and `.mnu` files.



NOTE

Do not copy a `dir.mnu` file into the category folder. When you add a category, LabVIEW creates a `dir.mnu` file, if needed.

1. Restart LabVIEW. LabVIEW updates the palettes to contain the categories you added.

Adding Palette Items to the Favorites Category

Use the **Favorites** category to group together items on the **Functions** palette that you access frequently. You can add items to the **Favorites** category using the **Category (Standard)**, **Category (Icons and Text)**, **Icons**, and **Icons and Text** formats.

Complete the following steps to add a palette or object to the **Favorites** category.

1. On a pinned **Functions** palette, right-click an object and select **Add Item to Favorites** from the shortcut menu. In the **Category (Standard)** and **Category (Icons and Text)** formats, you also can expand a palette to display a subpalette, right-click the title of the subpalette, and select **Add Subpalette to Favorites** from the shortcut menu.
2. Select the **Favorites** category. Notice that the items you selected now appear in the **Favorites** category.

Right-click an object in the **Favorites** category and select **Remove Item from Favorites** or **Remove Subpalette from Favorites** from the shortcut menu to remove an item. In the **Category (Standard)** and **Category (Icons and Text)** formats, you also can remove objects and palettes from the **Favorites** category using the Organize Favorites dialog box. Right-click the **Favorites** category and select **Organize Favorites** from the shortcut menu to display the **Organize Favorites** dialog box.

Adding VIs and Controls to the User and Instrument Drivers Subpalettes

The User Controls and User Libraries palettes contain subpalettes for each directory, LLB, or palette file `.mnu` in `labview\user.lib` and icons for each file in `labview\user.lib`.

Save subVIs in the `user.lib` directory only if they are usable, without modification, across projects. Paths to VIs in `user.lib` are relative to the `labview` directory. Paths to subVIs you save elsewhere are relative to the parent VI. Therefore, copying a VI from `labview\user.lib` to modify it for a special case does not change the path to its subVIs located in `labview\user.lib`.



CAUTION

Do **not** save your own VIs and other files in *LLB files installed with LabVIEW*. Save your controls and VIs in the `user.lib` directory to add them to the **Controls** and **Functions** palettes.

Complete the following steps to add new VIs and controls to the user palettes.

1. Save the controls and VIs in the `labview\user.lib` directory to add them to the **User Controls** and **User Libraries** palettes, respectively.
2. Restart LabVIEW. LabVIEW updates the palettes to contain the controls and VIs in the `labview\user.lib` directory.

The **Instrument Drivers** palette corresponds to the `labview\instr.lib` directory. Save instrument drivers in this directory to add them to the **Functions** palette.



NOTE

If you add a subdirectory or LLB to the `labview\user.lib` or `labview\instr.lib` directory and the subdirectory or filename begins with an underscore `_`, the contents of the subdirectory or file do not appear on the **User Libraries** or **Instrument Drivers** palettes.

Adding VIs or Objects from the Functions or Controls Palette

After you add a VI, custom function, or custom control to the Functions or Controls palette, you can add it to a new or existing VI.

Complete the following steps to add a VI or function from the **Functions** palette:

1. Create or open a VI you want to add the VI or function to.
2. Navigate to the location on the **Functions** palette where you added the VI or function.

3. Click the VI or function icon and place it on the block diagram of the VI.

Complete the following steps to add a VI or control from the **Controls** palette:

1. Create or open a VI you want to add the VI or control to.
2. Navigate to the location on the **Controls** palette where you added the VI or control.
3. Click the VI or control icon and place it on the front panel of the VI.



NOTE

If the VI, function, or control you add from either the **Functions** or **Controls** palette contains content on both the front panel and the block diagram, LabVIEW adds those contents to both the front panel and block diagram.

Related Information

Adding VIs or Objects to the Functions or Controls Palette

Adding VIs or Objects to the Functions or Controls Palette

Add VIs and objects to the Functions or Controls palette or subpalettes to use in a new or existing VI. A subpalette can contain VIs, functions, controls, and subpalettes. A subpalette can also contain VIs from different locations.

Adding VIs or Functions to the Functions Palette

You can add any VI block diagram or custom function to the **Functions** palette. You can then place the block diagram contents of one VI on the block diagram of another VI straight from the **Functions** palette instead of calling a subVI.

Complete the following steps to add a VI or function to the **Functions** palette:

1. Create a VI or function you want to add to the **Functions** palette. Save and close the file.
2. Select **Tools»Advanced»Edit Palette Set** to display the **Edit Controls and Functions Palette Set** dialog box.
3. Navigate to the location on the **Functions** palette where you want to add the VI or function.
4. Right-click the palette and select **Insert»VI(s)** from the shortcut menu.
5. Navigate to the VI or function you want to add to the palette.
6. Select the VI or function you want to add to the palette and click **Open**.
7. Right-click the VI or function icon and select **Place VI Contents** from the shortcut menu.

8. Click **Save Changes** in the **Edit Controls and Functions Palette Set** dialog box to open the **Preview Palette Changes** dialog box.
9. Click **Continue** to accept the changes and close the **Edit Controls and Functions Palette Set** dialog box.

Adding VIs or Controls to the Controls Palette

You can add any VI front panel or custom control to the **Controls** palette. Unlike adding custom controls, adding VIs allows you to add multiple controls or decorations to the front panel at a time.



NOTE

To add custom decorations to the **Controls** palette, create a VI and add the decorations to the front panel. Then add the VI to the **Controls** palette.

Complete the following steps to add a VI or control to the **Controls** palette:

1. Create a VI or control you want to add to the **Controls** palette. Save and close the file.
2. Select **Tools»Advanced»Edit Palette Set** to display the **Edit Controls and Functions Palette Set** dialog box.
3. Navigate to the location on the **Controls** palette where you want to add the VI or control.
4. Right-click the palette and select **Insert»Custom Control(s)** from the shortcut menu.
5. Navigate to the VI or control you want to add to the palette.

Note You may have to select **All Files (*.*)** from the file type pull-down menu to include the `.vi` file extension in your search.

6. Select the VI or control you want to add to the palette and click **Open**.
7. Click **Save Changes** in the **Edit Controls and Functions Palette Set** dialog box to open the **Preview Palette Changes** dialog box.
8. Click **Continue** to accept the changes and close the **Edit Controls and Functions Palette Set** dialog box.

Automatically Updating Palette Sets

LabVIEW automatically updates the palettes linked to an LLB as you add files to or remove files from the LLB. You can set a palette to automatically update if you add new files to or remove files from a directory.

Complete the following steps to set a `.mnu` file to update based on the contents of a directory.

1. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
2. Right-click an icon on the palette and select **Synchronize With Directory** from the shortcut menu.
3. Navigate to and select the directory with which you want the palette set to synchronize. Although National Instruments recommends you have the .mnu file in the same directory, you can select any directory with which to synchronize the palette set.

**NOTE**

If you add a subdirectory or .llb file to the directory with which you synchronize and the subdirectory or filename begins with an underscore (_), the contents of the subdirectory or file do not appear on the palette.

If an item is synchronized to a directory, you can display the path to the source directory by right-clicking the item and selecting **Display Synchronization Path** from the shortcut menu.

If an item is synchronized with a directory, you cannot delete the item from the palette. To hide an item that is synchronized with a directory, right-click the item and select **Hide Synchronized Item** from the shortcut menu. Items that are hidden do not appear on the palettes and are not included when you [search the palettes](#).

Changing Palette Formats

Complete the following steps to change all palettes to another format.

1. Select **Tools»Options** to display the Options dialog box and select **Controls/Functions Palettes** from the **Category** list to display the Controls/Functions Palettes page. You also can click the **Customize** button on the **Controls** or **Functions** palette toolbar and select **Options** from the shortcut menu to display this page.
2. Select a palette format from the **Palette** pull-down menu, such as **Category (Standard)**, **Category (Icons and Text)**, **Icons**, **Icons and Text**, **Text**, or **Tree**.
3. Click the **OK** button. The **Controls** and **Functions** palettes change to the palette format you selected.

To change only the **Controls** palette or only the **Functions** palette to another format, click the **Customize** button on that palette and select a format from the **View This Palette As** shortcut menu.

**NOTE**

When you upgrade, LabVIEW attempts to restore the palette format from the previously installed version of LabVIEW.

Subpalettes

Creating .NET Subpalettes



NOTE

The following content applies to .NET Framework only. LabVIEW has been gradually adding support for .NET Core since LabVIEW 2024 Q3. Refer to *LabVIEW New Features and Changes* for specific .NET Core support added in different releases.

You can convert .NET Framework controls to custom controls and add them to a palette.



NOTE

Creating and communicating with .NET Framework objects in LabVIEW requires the [.NET CLR 4.0](#) that installs with LabVIEW. You must use a [.NET 2.0 configuration file](#) if you want to load .NET 2.0 mixed-mode assemblies. Refer to the [requirements for using .NET with LabVIEW](#) for more information about .NET restrictions in LabVIEW. NI strongly recommends that you use .NET objects only in [LabVIEW projects](#).

Complete the following steps to add a .NET Framework control to a palette.

1. Select **Tools»Import» .NET Controls to Palette** to display the Add .NET Controls to Palette dialog box.
2. Select an assembly from the **Assembly** pull-down menu. You also can click the **Browse** button to access private assemblies. .NET assemblies are .dll file types. After you select a private assembly, the assembly appears in the **Assembly** pull-down menu the next time you launch this dialog box.



NOTE

If you move a VI that uses a private assembly to a different folder or machine, you either must move the associated private .NET assembly files to [a subdirectory that LabVIEW can search](#), or you must save the files in the Global Assembly Cache (GAC). If you build a VI that uses a private assembly into a shared library or stand-alone application, LabVIEW copies the associated private .NET assembly files to the data subdirectory in the same directory as the library or application.

1. Select the .NET Framework control you want to convert from the **Controls** list.
2. In the **Destination** text box, enter the directory or .llb file in which you want to store the control. The default destination is the labview\menus\Controls\DotNet & ActiveX directory, which places controls that you add on the **.NET & ActiveX** palette. You also can click the **Browse** button and navigate to the directory in which you want to store the control.
3. Click the **OK** button to add the control to a palette.

**NOTE**

Do **not** save your own VIs and other files in *.llb files installed with LabVIEW*.

Creating ActiveX Subpalettes

You can convert ActiveX controls to custom controls and add them to a palette.

Complete the following steps to add a set of ActiveX controls to a palette.

1. Select **Tools»Import»ActiveX Controls to Palette** to display the Add ActiveX Controls to Palette dialog box, which lists the ActiveX controls registered on the computer.
2. Select one or more controls in the **Controls** list and click the **OK** button.
3. From the file dialog box that appears, select the directory or .llb file in which you want to store the controls. The default destination is the labview\menus\Controls\DotNet & ActiveX directory, which places controls that you add on the **.NET & ActiveX** palette.

**CAUTION**

Do **not** save your own VIs and other files in *.llb files installed with LabVIEW*.

Creating and Editing Subpalette Descriptions

Create descriptions for subpalettes on the **Controls** and **Functions** palettes to describe the purpose and contents of the subpalette and to give users instructions for which items to select. The subpalette description appears in the **Context Help** window when you move the cursor over the subpalette icon in the **Controls** and **Functions** palettes. The description also appears in the **Context Help** window when you move the cursor over an open area of the subpalette.

Complete the following steps to create and edit the description of a subpalette.

1. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
2. Right-click the subpalette and select **Edit Palette Documentation** from the shortcut menu.
3. Enter or edit a description of the subpalette in the **Palette Description** text box.
4. (Optional) [Link from the subpalette](#) to an HTML file or compiled help file.
5. Click the **OK** button.
6. Click the **Save Changes** button in the **Edit Controls and Functions Palette Set** dialog box to open the Preview Palette Changes dialog box and save the changes.

Creating Subpalettes in a Palette Set

Complete the following steps to insert a new, empty subpalette on the **Controls** or **Functions** palette. After you create a subpalette, you can [add objects to the subpalette](#).

1. [Create an icon](#) for each new object you want to add to the palette set.
2. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
3. Right-click the palette and select **Insert»Subpalette** from the shortcut menu to display the Insert Subpalette dialog box.
4. Select **Create a new palette file** to insert a new, empty subpalette and click the **OK** button.
5. In the file dialog box that appears, enter the name to use for the .mnu file. Add a .mnu extension to the filename to indicate that it is a palette. You must store each subpalette you create in a separate .mnu file.
6. Click the **OK** button. The Edit Palette Item Name dialog box appears.
7. Enter the name of the new subpalette and click the **OK** button. The new, empty subpalette appears on the palette.

Empty subpalettes or subpalettes that contain only items that LabVIEW is unable to find are only visible when editing palettes. If the palette uses an icon format, a question mark on the palette indicates that LabVIEW cannot find the object on the palette. If the palette uses a text format, objects with an X mark on a pinned palette and dimmed objects on a temporary palette indicate that LabVIEW cannot find the object on the palette.

Deleting Empty Slots on a Palette

Complete the following steps to delete an empty slot from the **Controls** or **Functions** palettes.

1. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
2. Right-click in an empty space on the **Controls** or **Functions** palettes and select **Delete Empty Slot** from the shortcut menu.
3. After you delete an empty slot from the palette, click the **Save Changes** button in the **Edit Controls and Functions Palette Set** dialog box to open the Preview Palette Changes dialog box and save the changes. Click the **Cancel** button to cancel any changes you made in this editing session and close this dialog box.

Editing a Palette Set

Complete the following steps to edit the **Controls** or **Functions** palettes.

1. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
2. Right-click the **Controls** or **Functions** palettes and select from the options on the shortcut menu. Right-click an open area on the palette to insert an object, *row, or column*, toggle the standard menu view, or *delete an empty slot* from the palette. Right-click an icon to *move a subpalette*, delete a subpalette, copy an icon, edit a subpalette icon, rename a subpalette, set a file to *update based on the contents of a directory*, or *edit an icon short name*. To move an object, drag it to a new location on the palette.
3. After you edit the palettes, click the **Save Changes** button in the **Edit Controls and Functions Palette Set** dialog box to open the Preview Palette Changes dialog box and save the changes. Click the **Cancel** button to cancel any changes you made in this editing session and close this dialog box.

To edit the palette set for a specific target, select **Tools»Advanced»Edit Palette Set** from a VI on the target whose palette set you want to modify.

If you edit the palettes, LabVIEW saves the edits to the `labview\menus` directory. LabVIEW saves the edits in a copy of the original palette. The protection of the original palettes ensures that you can experiment with the palettes without corrupting the original palettes. To revert to the original palettes, click the **Restore to Default** button on the **Edit Controls and Functions Palette Set** dialog box.

To edit the visible palette categories on the **Controls** or **Functions** palette, click the **Customize** button on the palette toolbar and select **Change Visible Palettes** from the shortcut menu to display the Change Visible Palettes dialog box. You also can click the **Change Visible Palettes** link at the bottom of a pinned palette to display the **Change Visible Palettes** dialog box.



NOTE

To edit a palette inside a *packed project library*, make changes to the palette and rebuild the packed library to update those changes in the library.

You also can use the Palette Editing VIs to edit the **Controls** or **Functions** palette set programmatically.

Editing Icon Short Names in a Palette Set

Complete the following steps to configure the text labels that appear below each icon when the palette format is **Icons and Text**.

1. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
2. Right-click an icon on the palette and select **Edit Short Name** from the shortcut menu to display the Edit Palette Item Name dialog box.
3. Enter the text label that you want to appear below the icon and click the **OK** button.

Inserting Rows or Columns in a Palette Set

Complete the following steps to insert a row or column on the **Controls** and **Functions** palettes.

1. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
2. Right-click an open area on the palette and select **Insert»Empty Slot** or **Insert»Empty Row** from the shortcut menu.

You also can create rows or columns by pressing the <Shift> key as you click an object and drag it to the right or bottom of the palette.

If a row or column appears at the end of a palette and only contains synchronized items that are hidden and/or items that LabVIEW cannot find, the row or column is only visible when editing palettes. If the palette uses an icon format, a question mark on the palette indicates that LabVIEW cannot find the object on the palette. If the palette uses a text format, objects with an X mark on a pinned palette and dimmed objects on a temporary palette indicate that LabVIEW cannot find the object on the palette.

Linking Subpalettes to HTML Files or Compiled Help Files

If you link a subpalette on the **Controls** and **Functions** palettes to an HTML file or to a [compiled help file](#), a link to that file appears in the **Context Help** window when you move the cursor over the subpalette icon or when you move the cursor over an open area of the subpalette.

Complete the following steps to link a subpalette to an HTML file or to a compiled help file.

1. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
2. Right-click the subpalette and select **Edit Palette Documentation** from the shortcut menu to display the Palette Documentation dialog box.
3. Click the **Browse** button.
4. Navigate to the .chm, .hlp, .htm, or .html file you want to link to and click the **Open** button. The file path appears in the **Help Path** text box.
5. If **Help Path** contains a path or [symbolic path](#) to a .htm or .html file, LabVIEW ignores the **Help Tag** text box. If **Help Path** contains a path to a compiled help file .chm or .hlp, use the **Help Tag** text box to link the subpalette to a specific topic in that help file:
 - For .chm files, you can enter a .htm or .html filename or an index keyword in the **Help Tag** text box. The .htm or .html filename must match the name of an individual, uncompiled HTML file in the HTML Help project. The index keyword must match a keyword in the index of the HTML Help project. To link to a bookmark within an HTML file, add # followed by the

name of the bookmark to the end of the filename. If you want to link to a .chm file on Windows and link to the uncompiled HTML files on macOS and Linux, enter the HTML filename in the **Help Tag** text box. (Windows) LabVIEW links to the HTML file inside the .chm file. (macOS and Linux) LabVIEW links to the HTML file, which must be in the `help\html\help.chm` directory, where `help.chm` is the .chm filename you entered in the **Help Path** text box in step 4.

- For .hlp files, you can enter an index keyword in the **Help Tag** text box. The index keyword must match a keyword in the index of the WinHelp or HyperHelp project.
6. Click the **OK** button.
 7. Click the **Save Changes** button in the **Edit Controls and Functions Palette Set** dialog box to open the Preview Palette Changes dialog box and save the changes. When you move the cursor over the subpalette icon or over an open area of the subpalette, a blue **Detailed help** link appears in the **Context Help** window, and the **Detailed help** button is enabled.
 8. Click the blue link or the **Detailed help** button in the **Context Help** window to test the link from the subpalette to the HTML or compiled help file.

Moving Subpalettes in a Palette Set

Complete the following steps to move a subpalette on the **Controls** and **Functions** palettes.

1. Select **Tools»Advanced»Edit Palette Set** to display the Edit Controls and Functions Palette Set dialog box.
2. Right-click the subpalette and select **Move Subpalette** from the shortcut menu.
3. Click the palette where you want to paste the subpalette.

You also can move a subpalette by pressing the <Shift> key as you click an object and drag it.

Organizing Palette Categories

You can change the category order in the **Category (Standard)** and **Category (Icons and Text)** formats. On a pinned palette, use the following methods to change the order in which categories appear on the palettes:

- Right-click a category and select **Move this Category Up** or **Move this Category Down** from the shortcut menu.
- Click the double lines to the left of a category and drag the arrow to where you want the category to appear.
- Right-click a category and select **Move to Top (Expand by Default)** from the shortcut menu to move the category to the top of the palette. By default, LabVIEW expands the top-most category.

Sharing Palette Sets

After you [edit a palette set](#), you can share the customized palettes with other users.

1. Copy the `labview\menus` directory from the [default data directory](#).
2. Have users place the copy in their default data directory.

When the users restart LabVIEW, the customized palettes are available.

Showing and Hiding Palette Categories

You can specify which categories to show or hide on the **Controls** or **Functions** palettes. On a pinned palette, use the following methods to specify which categories to show or hide:

- Click the **Customize** button on the palette toolbar, and then select **Change Visible Palettes** from the shortcut menu to display the Change Visible Palettes dialog box. In this dialog box, you can select the categories you want to show. LabVIEW hides the categories that you do not select.
- Click the **Change Visible Palettes** link at the bottom of a pinned **Controls** or **Functions** palette to display the **Change Visible Palettes** dialog box.
- Click the double arrows at the bottom of a **Controls** or **Functions** palette to display hidden categories on the current palette.



NOTE

Hidden categories remain hidden in all palette view formats. You can show or hide categories in all view formats.

Configuring Text Comparison for File Recovery

Complete the following steps to configure text comparison for recovered project (.lvproj, project library (.lvlib, XControl (.xctl, and LabVIEW class (.lvclass files.



NOTE

If LabVIEW automatically recovers one of the previously listed file types before you configure text comparison, LabVIEW disables the **Compare Backup with Original** button in the Select Files to Recover window.

1. In the LabVIEW Data directory, create a new text file called `LVAutoSave_Text_Compare.txt`.
2. Use [format specifier syntax](#) to enter a string that calls the third-party text comparison software that you want to use to compare recovered files. Enter the path to the text comparison executable within quotation marks, then enter the string "%s" "%s". On Windows, for example, if you want to call a program located on

the C:\ drive, enter the following text, where xxx represents the name of the executable to use for text comparison. "C:\\Program Files\\ Folder Name \\ xxx.exe" "%s" "%s"

3. Do not enter any other text or make any carriage returns. Save LVAutoSave_Text_Compare.txt and close the file.

Source Controls with LabVIEW

Configuring LabVIEW to Use Source Control

Before you use source control with LabVIEW, you must configure LabVIEW to work with the third-party source control provider you [select](#). Refer to the KnowledgeBase for the most current list of third-party source control providers that work with LabVIEW.



NOTE

Source control is available only with the Professional Development System.

Complete the following steps to configure LabVIEW to work with a source control provider.

1. Make sure the computer on which you are running LabVIEW also includes a third-party source control provider, which you should install and configure to meet the requirements of the current software project. Consult your source control administrator for additional assistance.
2. From a LabVIEW project or a blank VI, select **Tools»Source Control»Configure Source Control** to display the Source Control page of the Options dialog box. You also can select **Tools»Options** from the **Getting Started** window to display the Options dialog box, then click the **Source Control** category to display the Source Control page.
3. Select the source control provider you want to use with LabVIEW from the **Source Control Provider Name** pull-down menu. A dialog box from the third-party source control provider appears so you can configure connections, user information, and other settings for LabVIEW files under source control. For example, if you select **Perforce Command Line**, the Perforce Project Options dialog box appears. (Windows) LabVIEW scans the Windows registry to determine which source control providers are installed and uses that information to populate the **Source Control Provider Name** pull-down menu. (macOS and Linux) LabVIEW runs a query of environment variables to determine if Perforce is installed. LabVIEW searches the default environment path before login scripts run. If so, the **Perforce Command Line** option appears in the **Source Control Provider Name** pull-down menu.
4. The source control project settings you select in the source control provider dialog box appear in the **Source Control Project** text box. Click the **Change** button to reconfigure the settings. If the provider does not support this functionality, the **Change** button is disabled.
5. Click the **Advanced** button to set advanced optional configuration options specific to the source control provider. A dialog box from the source control provider appears. For example, if you are using the Perforce command line interface, the Perforce Configuration Options dialog box appears. If the provider does not support this functionality, the **Advanced** button is disabled.

6. Set the source control display and prompt options you want. Close the provider dialog box.
7. Click the **OK** button to save configuration settings and close the Options dialog box.



Tip Consider [separating compiled code](#) from VIs and other file types that you store in source control. By default, when you edit a VI, LabVIEW automatically recompiles all VIs that call the changed VI, which introduces unsaved changes to the calling VIs. If the calling VIs are stored in source control, you must check them out and then check them back in. However, if you separate compiled code from the VIs within a [VI hierarchy](#), changes to one VI do not produce unsaved changes elsewhere in the VI hierarchy. This behavior allows you to check out only the VI you want to modify.

After you configure LabVIEW to work with the source control provider, you can [add files to source control](#), [check out](#) files, [check in](#) edited files, [rename](#) files and [get the latest version](#) of files in source control within LabVIEW. You also can [configure source control for individual LabVIEW projects](#). If you change the provider in the Source Control page, the change affects all VIs and projects you use in LabVIEW.

If you want to perform source control operations that are not available in LabVIEW, work directly from the source control provider for those specific functions.

Configuring Source Control for LabVIEW Projects

Before you configure source control settings for a [LabVIEW project](#), you must [configure LabVIEW to use source control](#). LabVIEW projects use the source control configuration you specify for the LabVIEW environment by default.



NOTE

Source control is available only with the Professional Development System.

Complete the following steps to configure source control settings for an individual LabVIEW project.

1. [Configure LabVIEW to use source control](#) if you have not already done so.
2. From the [Project Explorer](#) window, select **Project»Properties** or right-click the project root and select **Properties** from the shortcut menu to display the Project Properties dialog box.
3. Select **Source Control** from the **Category** list to display the Source Control page.
4. Select one of the following options:
 - **Use environment source control settings** —Uses the source control settings you configure for the LabVIEW environment for the LabVIEW project. This is the default setting.
 - **Use project-specific source control settings** —Specifies a different source control project than the one specified by the LabVIEW environment source control settings to use with the LabVIEW project. Click the **Change** button to display a configuration dialog box from which you can specify a source

control project. The dialog box that appears is specific to the source control provider you use.

- **Do not use source control for this project** —Disables source control for the LabVIEW project, regardless of the source control settings you configure for the LabVIEW environment.
5. Click the **OK** button to apply the source control configuration to the LabVIEW project.



Note You cannot change the source control provider from a LabVIEW project. Source control settings on LabVIEW projects must match the provider you specify for the LabVIEW environment. If you change the source control provider for the LabVIEW environment, any LabVIEW project with configuration settings specific to a different source control provider acts as if source control is not enabled, and a warning appears in the **Source Control Properties** page. Use the Source Control page of the Options dialog box to change the source control provider for the LabVIEW environment. In LabVIEW, you can use only one source control provider at a time.

Checking out Files from Source Control

You should check out a file from source control when you want to make changes to the file. If you try to edit a file in source control that you did not check out, LabVIEW prompts you to check out the file if you [configured source control](#) to enable prompts.

To check files out from source control, LabVIEW must be [configured to use source control](#).



CAUTION

If you modify a local file before you check out the file from source control, the version of the file on the local directory is different than the version in source control. Depending on the source control provider, you might lose the changes in the local file when you check out the file from source control. Refer to the source control provider documentation for more information about the effects of checking out files.

Complete the following steps to use LabVIEW to check out a file.

1. If you are working in a LabVIEW project, right-click the folder or files you want to check out and select **Check Out** from the shortcut menu. You also can click the **Check Out** button on the Source Control toolbar or select **Tools»Source Control»Check Out**. If you are working on individual VIs outside a project, open the VI you want to check out. Select **Tools»Source Control»Check Out**.
2. If you are working in a project, notice that a red checkmark  appears on the file icon to show the file is checked out of source control.

In general, avoid checking out files for long periods of time. Make incremental changes to files when possible. When you check in a file, make sure you have tested it thoroughly. If you need to modify other files before you can check in a specific file, check out the other files, make the changes, and test them before checking in any of the files.

If you need to make several changes to a file, consider checking in the file after you complete one modification and then checking out the file again to start the next modification. Other users can access your changes and you have a checkpoint you can return to if you later decide that subsequent changes were incorrect.

Undoing File Check-out from Source Control

You can undo a file [check-out operation](#) if you want to undo changes you made after you checked out a file. The source control provider retains no record of the check-out operation or the changes you made.

Complete the following steps to undo file check out.

1. If you are working in a LabVIEW project, right-click the file or files for which you want to undo the check-out operation and select **Undo Check Out** from the shortcut menu. You also can click the **Undo Check Out** button on the Source Control toolbar or select **Tools»Source Control»Undo Check Out**. If you are working on individual VIs outside a project, open the VI for which you want to undo the check-out operation. Select **Tools»Source Control»Undo Check Out**.
2. If you are working in a project, notice that the red checkmark in the file icon disappears because the file is no longer checked out of source control.

Checking in Files to Source Control

When you finish making changes to a file you [checked out](#), check in the file to copy that version into source control so it is available to other users.

To check files into source control, LabVIEW must be [configured to use source control](#).

Complete the following steps to check a file in to source control.

1. If you are in a project, select the folder or files you want to check in. If you are working on individual VIs outside a project, open the VI you want to check in.
2. Select **Tools»Source Control»Check In** to display the Source Control Operations dialog box. From within a project, you also can click the **Check In** button on the Source Control toolbar or right-click the file or folder and select **Check In** from the shortcut menu.
3. The **Files** tree contains the names and paths of all files to check in to source control. Remove the checkmark from the checkbox next to the name of any file you do not want to check in to source control.
4. Enter a summary of the changes you made in the **Comment** text box.
5. Click the **OK** button to check in the files. If any files in the source control operation have unsaved changes, the Unsaved Files dialog box appears when you click the **OK**

button. If you are in a project, the red checkmark disappears from the file icon to show that the file no longer is checked out of source control.



NOTE

Depending on the source control provider, if you attempt to close the Source Control Operations dialog box and the **Comment** text box is empty, a provider-specific dialog box might display so you can enter the information.

Depending on the source control provider you are using, the provider might automatically set the file as read-only while it is checked in to prevent you or others from accidentally modifying the file without checking it out again.

Getting the Latest Version of Files in Source Control

When another user has edited a file or added a new file to source control, you can use LabVIEW to copy the latest version of the file from source control to the local directory.

Complete the following steps to get the latest version of a file from source control.

1. If you are in a LabVIEW project, select the folder or files for which you want to retrieve the latest version. If you are working on an individual VI outside a project, open the VI.
2. Select **Tools»Source Control»Get Latest Version**. A dialog box appears to remind you that the latest versions of the files will overwrite the current files on the computer. Click the **Yes** button to continue the operation and display the Source Control Operations dialog box.
3. The **Files** tree contains the names and paths of all files for which you want to retrieve the latest version from source control. Remove the checkmark from the checkbox next to the name of any file for which you do not want to retrieve the latest version.
4. Click the **OK** button. If the file exists on the computer, the latest version in source control overwrites the local version and any changes that you made to the file are lost.
5. If you selected a VI for which you want to retrieve the latest version and the VI was open during the operation, LabVIEW prompts you to reload the VI from the disk. If you reload the VI, any unsaved changes you made are lost.

You can verify that a file is the latest version if you view the source control properties. Select **Tools»Source Control»Properties** from the menu in the front panel or block diagram window of a VI to display the source control properties for a file.

Retrieving Missing or Out-of-Date Files in Source Control

You also can retrieve the latest version of missing or out-of-date files in a LabVIEW project. LabVIEW retrieves the latest version of any missing or out-of-date files from source control and copies the files to the local directory.

Complete the following steps to locate and retrieve missing or out-of-date files in a project:

1. Right-click the project root and select **Get Latest Version of All Files** from the shortcut menu. If there are missing items, the Missing File Information dialog box appears and lists missing files that LabVIEW cannot retrieve, such as dependencies or files not in source control.
2. To retrieve any remaining files, click **Continue** in the Missing File Information dialog box to display the **Get Latest Version** dialog box, which is a modified version of the Source Control Operations dialog box.
3. Select the files you want to retrieve. By default, LabVIEW selects all the files.
4. Click the **OK** button. LabVIEW retrieves the selected files and copies them to your local directory.

Comparing Files in Source Control

You can use LabVIEW to compare a local version of a file in source control with the latest version from the source control provider. Comparing files is useful when you are working on a large project in which files might change frequently. You can perform this operation on only one file at a time.

You can use a [sort option](#) to organize items within a project. The sort option you select preserves the organization of the project on disk, which allows you to merge and compare fewer changes when you submit the project to source control.

If you select a VI to compare, LabVIEW displays the results in the Differences dialog box.



NOTE

If LabVIEW and the third-party source control provider are not compatible, LabVIEW launches the default comparison tool of the third-party source control provider. Perforce SCM and Rational ClearCase are not compatible with the procedure LabVIEW uses to complete a graphical differencing of VIs.

Complete the following steps to compare two VIs in source control.

1. If you are in a LabVIEW project, right-click the VI you want to compare and select **Show Differences** from the shortcut menu to display the Differences dialog box. If you are working on individual VIs outside a project, open the VI you want to compare. Select **Tools»Source Control»Show Differences** to display the Differences dialog box.



Note If the VI contains any unsaved changes, the Unsaved Files dialog box appears and prompts you to save changes before continuing with the compare operation.

1. [Select comparison options](#) in the Differences dialog box.

Complete the following steps to compare two non-VI files in source control. You can compare non-VI files only within a project.

1. From the project, right-click the file you want to compare and select **Show Differences** from the shortcut menu. The default comparison tool of the source control provider launches.
2. When the comparison is complete, the third-party provider displays the differences between the two files.

Viewing Revision History in Source Control

You can view a brief summary of transactions for a file in source control.

Complete the following steps to view the history for a file in source control.

1. If you are in a LabVIEW project, select a file for which you want to view the source control history. If you are working on an individual VI outside a project, open the VI.
2. Select **Tools»Source Control»Show History** to display a provider-specific dialog box that contains a list of changes made to the file after it was added to source control. Depending on the source control provider, the information in the list might include the number of revisions, the users who made the changes, dates of the changes, and any comments attached to the revisions.
3. Close the dialog box.

Adding Files to Source Control

You can add folders, individual VIs, or any supported files in a LabVIEW project to source control. When you [configure LabVIEW to work with a source control provider](#), you can configure LabVIEW to prompt you to add files to source control when you add files to a project.

After a file is in source control, you can perform source control operations on the file in LabVIEW.

Complete the following steps to add files in LabVIEW to source control.

1. Save the files in the local directory that corresponds to the source control provider directory where you want to store the files. You cannot add new VIs or LabVIEW project libraries to source control until you save and name them.
2. Select **Tools»Source Control»Add to Source Control** to display the Source Control Operations dialog box. You also can click the **Add to Source Control** button on the Source Control toolbar. From within a project, right-click the folder or files you want to add and select **Add to Source Control** from the shortcut menu.
3. The **Files** tree contains the names and paths of all files to add to source control. Remove the checkmark from the checkbox next to the name of any file you do not want to add to source control.

4. Click the **OK** button. LabVIEW adds the file to source control. If you are working in a project, a white square  appears on the file icon to show the file is in source control.

**NOTE**

If you are using Perforce, LabVIEW adds the file or files to the default changelist. You must use Perforce to perform any advanced changelist actions, such as creating new changelists, or to navigate between changelists.

After a file is in source control, you can [check out files](#), [check in files](#) you edited, and [get the latest version](#) of a file in source control.

Removing Files from Source Control

Be careful when you remove files from source control. Some source control providers delete the local directory copy of the file, all previous versions of the file that the provider maintains, and the history log for the file. Refer to the source control provider documentation for more information about the effects of removing files from source control.

Complete the following steps to remove a file from source control.

1. If you are in a LabVIEW project, select the folder or files you want to remove from source control.
2. Select **Tools»Source Control»Remove from Source Control**.
3. A dialog box appears to remind you that removing a file from source control might cause the file to be deleted entirely. Click the **Yes** button to confirm that you want to continue and to display the Source Control Operations dialog box.
4. The **Files** tree contains the names and paths of all files to remove from source control. Remove the checkmark from the checkbox next to the name of any file you do not want to remove from source control.
5. Click the **OK** button.

**NOTE**

If you are using Perforce, LabVIEW adds the file or files to a changelist. You must use Perforce to submit the changelist.

Renaming Files in Source Control

Source control stores revision history based on the name of a file. As a result, to preserve the revision history when you modify the name of a file, follow the renaming process of the [source control provider](#).

Renaming a file can cause the file to lose [links to dependencies](#). To avoid this problem, complete the following steps when you rename a file in source control.

1. Open the LabVIEW project that contains the file you want to rename. The project must be open so that LabVIEW can detect when you rename a file.
2. **Check out** all project files from the source control provider.
3. Rename the file with the renaming process of the source code control provider.
4. Return to the LabVIEW project and press the <F5> key to refresh the project. The file you renamed has a warning that indicates you renamed it.
5. Right-click the file and select **Replace with** from the shortcut menu.
6. Navigate to and select the new version of the file.

Building Stand-Alone Applications with Source Control VIs

You can include Source Control VIs when you build stand-alone applications and shared libraries with the Application Builder tools.

Include the following steps when you build an application or shared library with Source Control VIs:

- Add the LabVIEW project library and its folder dependencies for the source control provider you configured to the LabVIEW project from which you are building the application or shared library. If you use the Perforce command line interface with LabVIEW, add `NI_SCC_P4CMD.lvlib` located in the `labview\vi.lib\SourceControl\Providers\P4CMD` directory. If you use a Windows-based source control provider, add `NI_SCC_WinCI.lvlib` located in the `labview\vi.lib\SourceControl\Providers\WinCI` directory.
- Add `NI_SCC_ConnSrv.lvlib` and its dependencies located in the `labview\vi.lib\SourceControl\sccconnsrv` directory to the LabVIEW project from which you are building the application or shared library. `NI_SCC_ConnSrv.lvlib` is the project library for the source control server.
- If you are building a stand-alone application, add the folder under each source control project library to the **Always Included** listbox on the **Source Files** page of the **Application Properties** dialog box. For example, if you use `NI_SCC_WinCI.lvlib`, then add the `WinCI` folder to the **Always Included** listbox.
- If you are building a shared library, add the folder under each source control project library to the **Always Included** listbox on the **Source Files** page of the **Shared Library Properties** dialog box. For example, if you use `NI_SCC_WinCI.lvlib`, then add the `WinCI` folder to the **Always Included** listbox.



NOTE

Refer to the NI KnowledgeBase at ni.com/kb for more information about copying source control configuration information from the `labview\labview.ini` file to the `.ini` file of the application you are building.

LabVIEW Classes

Creating a LabVIEW Class or Interface

Complete the following steps to create a LabVIEW *class* or *interface*.

1. [Create a new LabVIEW project](#) or open an existing project.
2. Right-click **My Computer** in the **Project Explorer** window and select **New»Class** or **New»Interface** from the shortcut menu.



NOTE

You also can select **File»New** to display the New dialog box and select **Other Files»Class** or **Other Files»Interface** from the **Create New** list to create a new class or interface.

3. In the New Class/Interface dialog box, enter a name to identify the purpose of the class or interface.
 - For classes, name all classes as nouns. For example, name a class Waveform Reader.lvclass rather than Reads Waveform.lvclass.
 - For interfaces, refer to [Best Practices for Naming Interfaces](#) for naming guidelines.
4. In the **Parent Class** and **Parent Interfaces** sections, select the class and interfaces for the new class or interface to inherit from.
5. In the **Save Options** section, choose from the following options:
 - Select **Save to Folder** and specify the path to save the class or interface immediately.
 - Select **Do Not Save** if you do not want to save the class or interface.
6. Click **OK**. The LabVIEW class or interface appears in the **Project Explorer** window. For LabVIEW classes, a private data control `.ctl` with the same name as the class also appears.

You can complete the following actions:

- (Class only) [Define the private data of the class](#).

Note You also can [create a LabVIEW class from a control or type definition](#).

- [Create member VIs](#).
- Edit the class or interface properties.

You can define data, create member VIs, or edit the class or interface properties in the order that you need. Some classes may not need private data, but may need member VIs so that you never define the private data control. You may change the wire appearance and other class or interface properties before developing the class or interface. You can develop aspects of the class or interface in the order that best suits the individual project and development plan.



Note You can create a directory on disk named the same as the LabVIEW class or interface to save the class library file, class member VIs, and *class custom default probes*. You can include the files that the class library owns in the directory. If you include files for more than one class library in the same directory, conflicts might occur if you try to include VIs of the same name in different class libraries. Naming conflicts can occur in the development process when you override *dynamic member VIs*.

Converting the Contents of a Control to a LabVIEW Class

You can create LabVIEW classes from existing controls and type definitions. Complete the following steps to convert the contents of a control or type definition to a LabVIEW class.

1. Open a LabVIEW project containing an existing custom control or type definition.
2. Right-click the control or type definition in the *Project Explorer* window and select **Convert Contents of Control to Class** from the shortcut menu. LabVIEW creates a class with a private data control. The new class retains the name of the control that you create it from.



NOTE

LabVIEW adds the new class to the front panel of the control and continues to use the control.

While LabVIEW leaves the control from which you create a class in the **Project Explorer** window, LabVIEW modifies the control on disk. Because LabVIEW updates the control to use the LabVIEW class, all references to the control might be broken. You need to provide accessors for the private data for those VIs to use or move those VIs into the newly created class.

Defining the Private Data of a LabVIEW Class

You can add controls to the cluster in the private data control of a LabVIEW class to create a data type.



NOTE

You cannot add controls to an interface because interfaces do not contain any private data.

Complete the following steps to define a LabVIEW private data control.

1. *Create a new LabVIEW class* or open an existing class by selecting **File»Open** and navigating to the class you want to open. The private data control appears underneath the class icon in the *Project Explorer* window.
2. Double-click the private data control. The *Control Editor* window appears.
3. Add controls and indicators to the **Cluster of class private data** to define your data type, if necessary.

Note A private data control cannot contain an *XControl*.

4. Select **Edit»Make Current Values Default** to set the default value for the entire cluster. LabVIEW sets the current values in each element as the default value for those elements.

Note The value you set as the default value of the private data control is the default value for all controls, indicators, and constants of the class type.

5. Save the private data control.



Note LabVIEW saves the private data control inside the `.lvclass` file. The control does not have a separate file on disk. Saving the private data control with the LabVIEW class definition helps to prevent user errors.

Creating a Member VI in a LabVIEW Class or Interface

Create a member VI to define an operation that a *class* or *interface* can perform. Member VIs are privileged to call other methods in private or protected scope. Class member VIs are privileged to access the class private data. You can create a member VI in several ways.

Creating a Member VI from a Blank VI

Complete the following steps to create a member VI from a blank VI.

1. *Create a new LabVIEW class or interface* or open an existing class or interface by selecting **File»Open** and navigating to the class or interface you want to open.



NOTE

If you open the class or interface using **File»Open**, the class or interface opens in a class window instead of the **Project Explorer** window. You can develop the class or interface in the class window just like you do in the **Project Explorer** window.

2. (Class only) *Define the private data of the class*.
3. In the *Project Explorer* window, right-click the project item for the class or interface in which you want to create a member VI and select **New»VI** from the shortcut menu. A new member VI  appears in the **Project Explorer** window, and LabVIEW opens the VI.
4. You now can develop the member VI to define an operation that a class or interface can perform.

(Class Only) Creating a Member VI in a Property Definition Folder

If you create a member VI in a property definition folder, the private data of a LabVIEW class becomes a property of the class that you can access using a Property Node.



Note The name that you specify for the property definition folder determines the property name that appears on the Property Node. To modify the property name, you must rename the corresponding property definition folder. Renaming the private data does not modify the property name that appears on the Property Node.

Complete the following steps to create a member VI in a property definition folder.

1. [Create a new LabVIEW class](#) or open an existing class by selecting **File»Open** and navigating to the class you want to open.

Note If you open the class using **File»Open**, the class opens in a class window instead of the **Project Explorer** window. You can develop the class in the class window just like you do in the **Project Explorer** window.

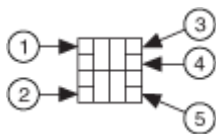
2. [Define the private data of the class](#).
3. In the [Project Explorer](#) window, right-click the icon for the class  in which you want to create a member VI and select **New»Property Definition Folder** from the shortcut menu. A new property definition folder  appears in the **Project Explorer** window.
4. Right-click the property definition folder and select **New»VI** from the shortcut menu. You also can create a new VI from a [dynamic dispatch template](#) or a [static dispatch template](#). A new member VI  appears in the **Project Explorer** window, and LabVIEW opens the VI.

Note You also can add existing member VIs to the property definition folder by right-clicking the property definition folder and selecting **Add»File** from the shortcut menu.

5. You now can develop the member VI to access and interact with the private data of the class. Each property definition folder can contain one VI to read and/or one VI to write the private data of the class.

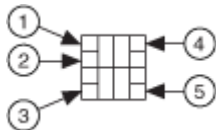
Note If you use a Property Node to access a LabVIEW class, you can quickly view the implementation of a LabVIEW class property by right-clicking the class property and selecting **Open Accessor VI** from the shortcut menu. This option appears only if you wire a LabVIEW class or a data value reference that points to a LabVIEW class to the **reference** input of the Property Node and the block diagram of the accessor VI is available. If the property is dynamic with more than one implementation, selecting this option displays the Choose Implementation dialog box. Use this dialog box to view all implementations of the property, or dynamic dispatch member VI, and [open one or more implementations](#).

Member VIs that enable you to read or write private data using a Property Node must have a 4 × 2 × 2 × 4 connector pane pattern. The terminals of a member VI that reads private data must have the following connector pane terminals.



1 Class In	3 Class Out	5 Error Out
2 Error In	4 Value Out	

The terminals of a member VI that write private data must have the following connector pane terminals.



1 Class In	3 Error In	5 Error Out
2 Value In	4 Class Out	



Note **Class In** and **Class Out** of a member VI must be the same LabVIEW class. If a property definition folder contains both a read and write member VI, **Value Out** of the read VI and **Value In** of the write VI must be the same data type.

Creating a Member VI from the Dynamic Dispatch Template

You can use any VI template to create a new member VI using the New dialog box; however, LabVIEW provides a quick shortcut to creating a dynamic dispatch member VI that includes several features that many dynamic methods need. LabVIEW populates the new member VI with **error in** and **error out** clusters, a Case structure for error handling, the input class or interface, and the output class or interface. LabVIEW sets both the input and output terminals as *dynamic on the connector pane of the VI*. Complete the following steps to create a member VI from a dynamic dispatch template.

1. *Create a new LabVIEW class or interface* or open an existing class or interface by selecting **File»Open** and navigating to the class or interface you want to open.

Note If you open the class or interface using **File»Open**, the class or interface opens in a class window instead of the *Project Explorer* window. You can develop the class or interface in the class window just like you do in the **Project Explorer** window.

2. (Class only) *Define the private data of the class*.
3. In the **Project Explorer** window, right-click the project item for the class or interface in which you want to create a member VI and select **New»VI from Dynamic Dispatch Template** from the shortcut menu. A new VI appears in the **Project Explorer** window, and LabVIEW opens the VI. The front panel of the VI displays a class or interface indicator, a class or interface control, and **error in** and **error out** clusters.
4. Press the <Ctrl-E> keys to display the block diagram. (macOS) Press the <Option-E> keys. (Linux) Press the <Alt-E> keys. The block diagram displays two class or interface terminals, **error in** and **error out** cluster terminals, and a Case structure for error handling.

5. You now can develop the member VI to access and interact with the private data of the class.

Creating a Member VI from the Static Dispatch Template

LabVIEW also provides a quick shortcut to creating a static dispatch member VI. LabVIEW populates the new member VI with **error in** and **error out** clusters, a Case structure for error handling, the input class or interface, and the output class or interface. Contrary to creating a dynamic dispatch VI, LabVIEW does not set the input and output terminals as dynamic on the connector pane of the static dispatch VI. Complete the following steps to create a member VI from a static dispatch template.

1. [Create a new LabVIEW class or interface](#) or open an existing class or interface by selecting **File»Open** and navigating to the class or interface you want to open.

Note If you open the class or interface using **File»Open**, the class or interface opens in a class window instead of the [Project Explorer](#) window. You can develop the class or interface in the class window just like you do in the **Project Explorer** window.

2. (Class only) [Define the private data of the class](#).
3. In the **Project Explorer** window, right-click the project item for the class or interface in which you want to create a member VI and select **New»VI from Static Dispatch Template** from the shortcut menu. A new VI appears in the **Project Explorer** window, and LabVIEW opens the VI. The front panel of the VI displays a class or interface indicator, a class or interface control, and **error in** and **error out** clusters.
4. Press the <Ctrl-E> keys to display the block diagram. (macOS) Press the <Option-E> keys. (Linux) Press the <Alt-E> keys. The block diagram displays two class or interface terminals, **error in** and **error out** cluster terminals, and a Case structure for error handling.
5. You now can develop the member VI to access and interact with the private data of the class.

(Class Only) Creating a Member VI for Accessing Data Members

You can quickly create a read or write accessor VI that is a member of the LabVIEW class. The read accessor VI unbundles LabVIEW class data so you can access the data in a calling VI. The write accessor VI rebundles new values for class data that you pass from the calling VI. You might use an accessor VI to read from or write to class data in VIs that are not members of the LabVIEW class. Complete the following steps to create a VI to access data of the class.

1. [Create a new LabVIEW class](#) or open an existing class by selecting **File»Open** and navigating to the class you want to open.

Note If you open the class using **File»Open**, the class opens in a class window instead of the [Project Explorer](#) window. You can develop the class in the class window just like you do in the **Project Explorer** window.

2. [Define the private data of the class](#).

3. In the **Project Explorer** window, right-click the icon for the class  in which you want to create a member VI and select **New»VI for Data Member Access** from the shortcut menu to display the Create Accessor dialog box. You also can right-click a data member in the private data control of the LabVIEW class and select **Create Accessor** from the shortcut menu.

Note You must save a new LabVIEW class before using this option. LabVIEW dims the **VI for Data Member Access** and **Create Accessor** options if you have not saved the new class.

4. Select the data member you want to access, the type of access, whether the new VI is *dynamic or static*, and whether you can access the data member using a Property Node. If you specify that the selected data member is accessible through a Property Node, LabVIEW adds the member VI to a *property definition folder*.

Note You can quickly view the implementation of a LabVIEW class property if you use a Property Node to access a LabVIEW class and specify that the selected data member is accessible through a Property Node. Right-click the class property and select **Open Accessor VI** from the shortcut menu. This option appears only if you wire a LabVIEW class or a data value reference that points to a LabVIEW class to the **reference** input of the Property Node and the block diagram of the accessor VI is available. If the property is dynamic with more than one implementation, selecting this option displays the Choose Implementation dialog box. Use this dialog box to view all implementations of the property, or dynamic dispatch member VI, and *open one or more implementations*.

If you press the <Ctrl> key and select more than one data member, LabVIEW creates an accessor VI for each data member. (macOS) Press the <Option> key. (Linux) Press the <Alt> key. If you select **Read and Write** in the dialog box, LabVIEW creates an accessor VI that provides read access and another accessor VI that provides write access.

5. Click the **Create** button to create the accessor VI or VIs. A new VI appears in the **Project Explorer** window for each data member and each type of access you select, and LabVIEW opens each VI automatically. The names of the VIs consist of the access type you chose and the name of the data member, for example, Read Numeric.vi. The front panel of the VI displays a LabVIEW class indicator, a LabVIEW class control, the data member you chose, and **error in** and **error out** clusters.
6. Press the <Ctrl-E> keys to display the block diagram. (macOS) Press the <Option-E> keys. (Linux) Press the <Alt-E> keys. The block diagram displays two LabVIEW class terminals, the data member you chose, **error in** and **error out** cluster terminals, and a Case structure for error handling.
7. You now can manipulate the class data from calling VIs.

Creating a Member VI to Override an Ancestor Member VI

You can create a dynamic dispatch member VI that overrides, or extends, the functionality of an ancestor dynamic dispatch member VI. A member VI can override a member VI from an ancestor class or interface only if the following settings are the same for both VIs:

- reentrant settings
- preferred execution settings

- priority settings
- [access scope settings](#)
- [connector pane terminal](#)
- [connector pane pattern](#)

LabVIEW creates an override member VI that meets all of the requirements of a legal override if you select **New»VI for Override** from the shortcut menu. Complete the following steps to create an override member VI using the **New»VI for Override** option.

1. [Create a new LabVIEW class or interface](#) or open an existing class or interface by selecting **File»Open** and navigating to the class or interface you want to open. If you are creating a new class or interface, you must save the class or interface before you can create an override member VI for it.

Note If you open the class using **File»Open**, the class opens in a class window instead of the [Project Explorer](#) window. You can develop the class in the class window just like you do in the **Project Explorer** window.

2. In the **Project Explorer** window, right-click the project item for the class or interface in which you want to create a member VI and select **New»VI for Override** from the shortcut menu.

Note LabVIEW dims the **VI for Override** option if the class or interface you select does not [inherit member VIs](#) that are available for you to override. The VIs in the ancestor class or interface must be [dynamic dispatch member VIs](#) in order for you to override them.

3. The New Override dialog box appears. The dialog box displays dynamic methods that ancestor classes or interfaces define. Select the member VI you want to override.
4. Click the **OK** button.
5. A new VI appears in the **Project Explorer** window, and LabVIEW opens the VI. LabVIEW uses the ancestor VI icon and overlays the icon mask of the child class or interface to create the icon of the new VI. Additionally, the front panel of the VI displays the controls and indicators copied from the ancestor VI, LabVIEW class indicators, and **error in** and **error out** clusters.
6. Press the <Ctrl-E> keys to display the block diagram. (macOS) Press the <Option-E> keys. (Linux) Press the <Alt-E> keys. The block diagram displays terminals that LabVIEW copied from the ancestor member VI. LabVIEW also wires the terminals to a Call Parent Class Method node automatically.

You now can change the code on the block diagram of the override member VI as you need.



Note A newly created override member VI initially behaves exactly as the ancestor implementation of the member VI. You can use an override member VI to add or subtract functionality from a parent implementation of a member VI.

Creating a Member VI That Requires Override VIs in Descendant Classes or Interfaces

You can create a dynamic dispatch VI that requires descendant classes or interfaces to override, or extend, the functionality of that member VI. If you require the descendants to override the ancestor member VI, the member VI does not have to provide any implementation of this VI for descendants. By default, interfaces require descendants to override member VIs, whereas classes do not require the descendants to override member VIs.

Complete the following steps to require descendants to override a member VI.

1. [Create a new LabVIEW class or interface](#) or open an existing class or interface by selecting **File»Open** and navigating to the class or interface you want to open.

Note If you open the class or interface using **File»Open**, the class or interface opens in a class window instead of the [Project Explorer](#) window. You can develop the class or interface in the class window just like you do in the **Project Explorer** window.

2. In the **Project Explorer** window, right-click the project item for the class or interface in which you want to create a member VI and select **New»VI from Dynamic Dispatch Template** from the shortcut menu. A new VI appears in the **Project Explorer** window, and LabVIEW opens the VI. The front panel of the VI displays a class or interface indicator, a class or interface control, and **error in** and **error out** clusters.
3. In the **Project Explorer** window, right-click the project item for the class or interface in which you created a member VI and select **Properties** to display the Class/Interface Properties dialog box.
4. Select **Item Settings** to display the Item Settings page and select the dynamic dispatch VI you want descendants to override.
5. Place a checkmark in the **Descendants must override** checkbox. You now do not have to provide any implementation of this VI, because each descendant must override this VI with its own implementation.

Viewing the Inheritance Hierarchy of a LabVIEW Class or Interface

The LabVIEW Class Hierarchy window displays a graphical representation of the inheritance of LabVIEW classes and interfaces. Use this window to view the hierarchy of the classes or interfaces currently in memory. Complete the following steps to use the **LabVIEW Class Hierarchy** window.

1. Right-click the class or interface whose hierarchy you want to view and select **Show Class Hierarchy**. The **LabVIEW Class Hierarchy** window appears.
2. Select from the following options:
 - Move the cursor over the classes or interfaces in the **LabVIEW Class Hierarchy** window to view the names of the classes or interfaces.
 - Search the LabVIEW class hierarchy by typing the name of the class or interface you want to find anywhere in the window. The search string

appears and displays the text as you type. LabVIEW highlights one item at a time whose name begins with the search string.

- View the hierarchy of all classes and interfaces in the LabVIEW project by selecting **View»Show All Classes** in the **LabVIEW Class Hierarchy** window.
- Use the buttons on the **LabVIEW Class Hierarchy** window toolbar to select more ways to view the LabVIEW class hierarchy.

You also can use the Class Browser window to view the LabVIEW class hierarchy.

Changing the Inheritance of a LabVIEW Class or Interface

By default, all LabVIEW classes and interfaces *inherit from LabVIEW Object*. To create classes that inherit from other classes, you must change the inheritance of a class. Complete the following steps to change the inheritance of a LabVIEW class.

1. **Create two new LabVIEW classes or interfaces** or use existing classes or interfaces.
2. Right-click the project item for the class or interface whose inheritance you want to change and select **Properties** from the shortcut menu. The Class/Interface Properties dialog box appears.
3. Select **Inheritance** from the **Category** list. The **Inheritance** page displays the current inheritance of the LabVIEW class or interface. Skip to step 5 if you are changing the inheritance of an interface.
4. Complete the following steps to change the parent class.
 1. Click the **Change Parent Class** button to display the Change Parent Class dialog box. In the **All Classes in Project** list, the diamond glyph indicates the current class whose inheritance you want to change.



NOTE

The **Change Parent Class** button is not available for interfaces because interfaces always inherit from LabVIEW Object.

2. Select the class from which you want the current class to inherit.

Note

- A LabVIEW class cannot inherit from any of its descendants or from any classes that use the class as part of its private data. If you select a class in the **All Classes in Project** list that the LabVIEW class cannot inherit from, LabVIEW displays an invalid glyph next to the class.
- Avoid inheriting one concrete class from an existing concrete class. Instead, create an interface or an abstract class as the common parent to both concrete classes. If you need to define state transitions for the concrete child classes, create an abstract class as the common parent. Otherwise, create an interface as the common parent.

3. Click the **Inherit From Selected** button. LabVIEW returns to the **Inheritance** page of the **Class/Interface Properties** dialog box and updates the **Parent Class Hierarchy**.
5. Complete the following steps to change the parent interfaces.

1. Click the **Change Parent Interfaces** button to display the Change Parent Interfaces dialog box.
2. Configure the parent interfaces by adding interfaces from the **Available interfaces** list or removing interfaces from the **Inherit from these interfaces** list.

Note A LabVIEW class or interface cannot inherit from any of its descendants. If you add an interface to the **Inherit from these interfaces** list that the class or interface cannot inherit from, LabVIEW displays an invalid glyph next to the interface.

3. Click **OK**. LabVIEW returns to the **Inheritance** page of the **Class/Interface Properties** dialog box and updates the **Parent Interfaces** list.
6. Click **OK** to save the inheritance configuration and close the **Class/Interface Properties** dialog box.

Adding Items to a LabVIEW Class or Interface

You can add items that you created previously to a LabVIEW class or interface. Complete the following steps to add items to a LabVIEW class or interface.

1. Right-click the class or interface in the **Project Explorer** window and select **Add»File** from the shortcut menu. The **Select a File to Insert** dialog box appears.
2. Browse to the location of the file you want to add and click the **OK** button. The file is added under the class or interface you selected in the **Project Explorer** window.



NOTE

Consider saving files you add to the LabVIEW class or interface in the directory you create for the LabVIEW class or interface. Doing so can make distributing the class or interface easier.

Changing the Wire Appearance of a LabVIEW Class or Interface

By default, a class uses the default LabVIEW class wire or inherits the wire appearance of the parent class. An interface uses the default LabVIEW interface wire. You can change the wire appearance of a LabVIEW class or interface to keep the block diagram readable. You might consider changing the wire appearance sparingly, keeping in mind that having excessive wire colors and patterns can make the block diagram more difficult to read. Complete the following steps to change the wire appearance of a LabVIEW class or interface.

1. In the **Project Explorer** window, right-click the project item for the class or interface whose wire you want to change and select **Properties** from the shortcut menu to display the Class/Interface Properties dialog box.
2. Select **Wire Appearance** from the **Category** list.
3. To customize the wire appearance, select **Use custom design**.

4. In the **Settings** section, make the changes to the wire appearance that you want. You can change the wire appearance, the colors of the wire, and the width of the wire. You can preview the wire in **Wire preview**.
5. Click the **OK** button when you are done making changes.

**NOTE**

If you change the edge foreground color of a LabVIEW class or interface wire, LabVIEW changes the border color of class or interface block diagram terminals to match.

Refer to the [LabVIEW Style Checklist](#) for more tips on changing the class wire appearance to make block diagrams easy to read and manage.

Creating Dynamic Dispatch Terminals

To create a dynamic dispatch member VI, you must set one input terminal on the connector pane to **Dynamic Dispatch Input (Required)**. Complete the following steps to set a connector pane input terminal to **Dynamic Dispatch Input (Required)**.

1. [Assign a front panel control or indicator](#) to each of the connector pane terminals you want to assign, if they are not already.
2. Right-click the input terminal on the connector pane that you want to set as a dynamic dispatch input and select **This Connection Is»Dynamic Dispatch Input (Required)** from the shortcut menu.

**NOTE**

You can set a terminal to **Dynamic Dispatch Input (Required)** only if the terminal is of the same class or interface type as the class or interface that owns the member VI.

You can create dynamic dispatch output terminals by right-clicking an output from the connector pane and selecting **This Connection Is»Dynamic Dispatch Output (Recommended)** from the shortcut menu.



Note Although a dynamic dispatch member VI can have no more than one dynamic input, the VI can include multiple dynamic outputs.

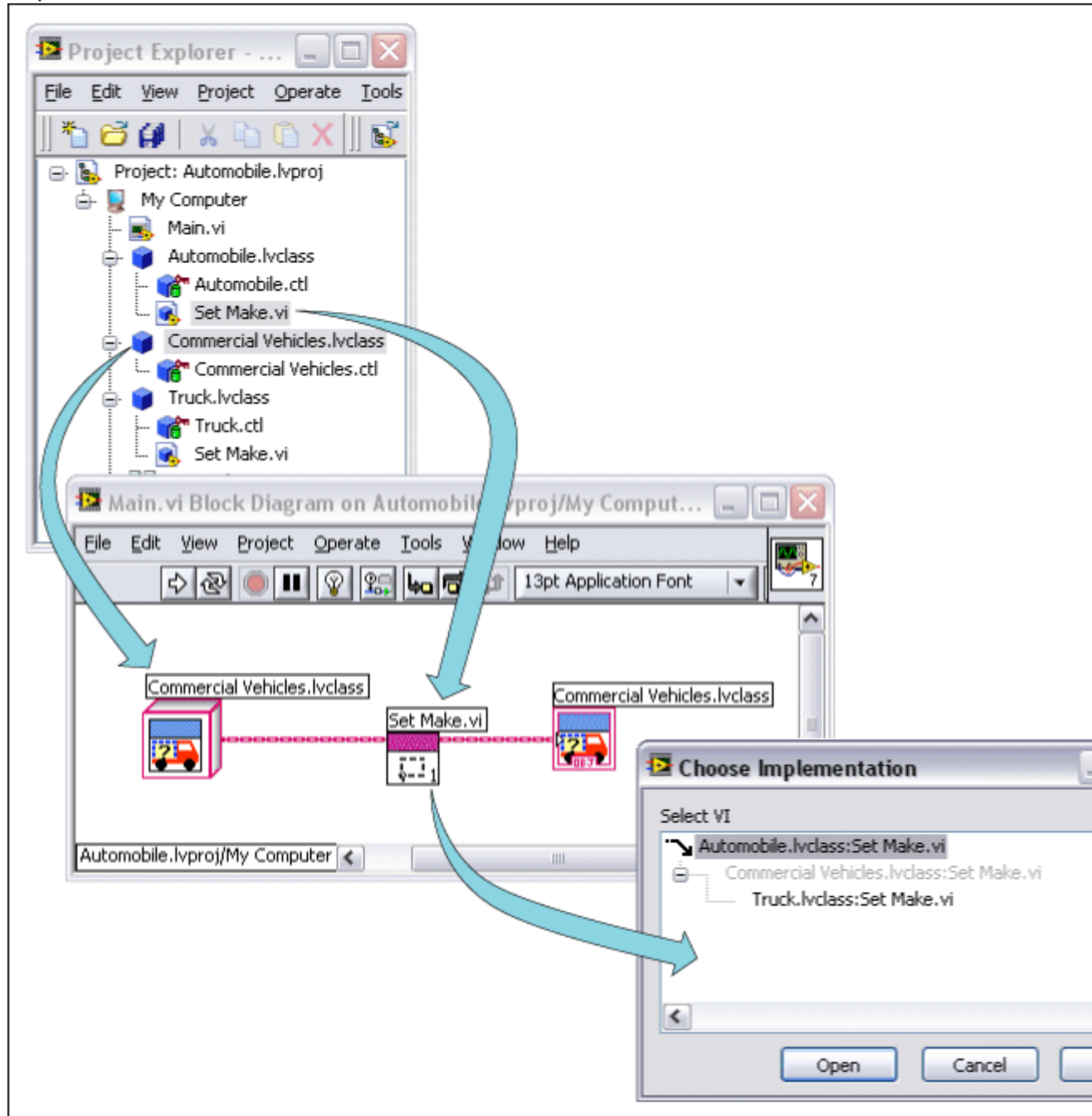
Viewing and Opening Implementations of Dynamic Dispatch Member VIs

[Dynamic dispatch subVIs](#) can call any one of a set of VIs in a LabVIEW class hierarchy. LabVIEW determines which implementation of the subVI to call at run time, depending on the class or interface data type flowing into the dynamic dispatch terminal.

Complete the following steps to open one or more implementations of a dynamic dispatch subVI using the Choose Implementation dialog box.

1. Double-click the dynamic dispatch VI on the block diagram. If one or more implementations exist for the VI, the **Choose Implementation** dialog box appears. The arrow glyph  indicates the implementation that executes if the run-time data type on the wire is the same as the edit-time data type on the wire. Because a LabVIEW class or interface wire can carry data of its own type or data of any child type, this glyph indicates that the implementation LabVIEW executes at run time will be at the glyph level or below in the class hierarchy. LabVIEW lists all classes and interfaces in the LabVIEW class hierarchy. If a class inherits the dynamic dispatch member VI from an ancestor class but does not have its own implementation, LabVIEW dims the class or interface in the **Select VI** list. Additionally, if the class or interface wired to the VI on the block diagram does not have its own implementation of the VI, the arrow glyph appears to the left of the ancestor class or interface from which it inherits the VI in the **Select VI** list. In the example below, the Commercial Vehicles class inherits the Set Make VI from the Automobile class. Because the Commercial Vehicles class is wired to the ancestor implementation of the Set Make VI on the block diagram, the arrow glyph appears next to the ancestor

implementation.



2. Select the VI in the **Select VI** list that you want to open. You also can open multiple VIs by holding the <Ctrl> key and selecting multiple VIs. (macOS) Hold the <Option> key. (Linux) Hold the <Alt> key.
3. Click the **Open** button to close the dialog box and open the VI or VIs you selected.

Setting the Access Scope of Member VIs

You can set the access scope of a member VI to control user access to the member VI. Complete the following steps to set the access scope of a member VI.

1. In the **Project Explorer** window, right-click the project item for the class or interface whose member VI access you want to change and select **Properties** from the shortcut menu. The Class/Interface Properties dialog box appears.
2. Select **Item Settings** from the **Category** list.
3. To customize the access scope of items in the LabVIEW class or interface, select an item from the **Contents** list on the Item Settings page.
4. In the **Access Scope** section, select from the following options:
 - **Public** —Any VI can call the member VI as a subVI.
 - **Community** —Only VIs that are friend of the class or interface, members of the class or interface, or members of a friended library can call the member VI. Community member VIs display a dark blue key glyph in the **Project Explorer** window.
 - **Protected** —Only VIs within the same or a descendant class or interface can call the member VI. Protected member VIs display a dark yellow key glyph in the **Project Explorer** window.
 - **Private** —Only VIs within the same class or interface can call the member VI. Private member VIs display a red key glyph in the **Project Explorer** window.
 - **Not specified** —This option appears only when you select a folder. The folder does not have access items specified. Access is public.
5. Repeat steps 3 and 4 for all items for which you want to change the access scope.
6. Click the **OK** button to close the **Class/Interface Properties** dialog box.

Locking and Unlocking LabVIEW Classes or Interfaces

You can lock LabVIEW classes or interfaces to prevent access to the private member VIs which in turn can help prevent errors being introduced to the application. As with [project libraries](#), adding password protection to LabVIEW classes or interfaces does not add password protection to the member VIs. You must password protect the member VIs individually.

Locking LabVIEW Classes or Interfaces

Complete the following steps to lock a LabVIEW class or interface:

1. Right-click a LabVIEW class or interface and select **Properties** from the shortcut menu. The Class/Interface Properties dialog box appears.
2. On the Protection page, select **Locked (no password)** or **Password-protected** to lock the LabVIEW class or interface.
 1. If you select **Password-protected**, enter and verify a password in the **Enter Password** dialog box.
 2. Click the **OK** button to close the **Enter Password** dialog box and return to the **Class/Interface Properties** dialog box.
3. Click the **OK** button.

Unlocking LabVIEW Classes or Interfaces

To view the private data of a locked class on a LabVIEW probe or member VIs of a locked class or interface, you must unlock the class or interface. Complete the following steps to unlock a class or interface:

1. Right-click a class or interface and select **Properties** from the shortcut menu. The **Class/Interface Properties** dialog box appears.
2. On the **Protection** page, select **Unlocked (no password)**. If the class or interface is locked with a password, enter a password in the **Authentication** dialog box and click **Verify** to return to the **Class/Interface Properties** dialog box.
3. Click the **OK** button.



NOTE

Locking a class or interface affects the appearance of the class or interface in the [Context Help](#) and the [generic probes](#). LabVIEW also hides the private member VIs of a locked class or interface on the [palettes](#). Changes to the palettes occur the next time you launch LabVIEW.



Note When you add the class private data control to the front panel of a VI, LabVIEW locks the class. You cannot unlock the class by deleting the control from the VI. Instead, you must delete the control and close or save the VI, to remove the class control from memory.

Setting the Default Probe for a LabVIEW Class or Interface

If you move a class or interface to a new machine or reinstall LabVIEW on the current machine, the default probe users of the class or interface can use is the generic probe. However, if you create a custom member probe for a LabVIEW class or interface and set the member probe as the default probe of the class or interface, when the class or interface is distributed to a new machine users can use the custom probe you set as the default probe.

You can set a [custom probe](#) you created as the default probe for LabVIEW class or interface data types. Complete the following steps to set a custom probe as a default probe for a LabVIEW class or interface.

1. [Add the custom probe to the LabVIEW class or interface](#) if the probe is not already a member.
2. Right-click the LabVIEW class or interface and select **Properties** from the shortcut menu. The Class/Interface Properties dialog box appears.
3. Select **Probes** from the **Categories** list.
4. Select the name of the custom probe you created from the **Default Probe** pull-down menu.

- Click **OK** to close the **Class/Interface Properties** dialog box.

**NOTE**

The probe must be a member of the LabVIEW class or interface.

Block Diagrams

Reading from Binary Files

Complete the following steps to read a double-precision, floating-point number from a *binary file*.

- Write to a binary file* to create the .bin file you need to complete the following steps.
- Add the Read from Binary File function to the block diagram.
- Wire the data type of the logged data to the Read from Binary File function.
 - Add a numeric constant to the block diagram.
 - Wire the numeric constant to the **data type** input of the Read from Binary File function.
 - Enter 0.0 into the numeric constant to inform LabVIEW that the data type you want to read is a double-precision, floating-point number.

You also can wire an array to the Read from Binary File function.

- Right-click the **data** output of the Read from Binary File function and select **Create»Indicator** from the shortcut menu.
- Run the VI. In the dialog box that appears, select a binary file with a *floating-point value*.

Refer to the Preallocated Read from Binary File function for information about reading binary data and placing it into a pre-allocated array without incurring a copy.

Refer to the Simple Binary File project in the labview\examples\File IO\Binary\Simple Binary directory for an example of how to open and read an existing file stored as a double-precision array in binary form.

Writing to Binary Files

Complete the following steps to write a double-precision, floating-point number to a *binary file*.

- Add the Write to Binary File function to the block diagram.
- Wire the data you want to write to the binary file.

1. Add a numeric constant to the block diagram.
2. Wire the numeric constant to the **data** input of the Write to Binary File function.
3. Enter 1.23 into the numeric constant. LabVIEW changes the representation of the numeric constant to a double-precision, floating-point number.

You also can wire an array to the binary file.

3. Run the VI.
4. In the dialog box that appears, create a name for the file, such as number.bin. LabVIEW creates the file in the directory specified by the **file** input and stores the double-precision, floating-point number in the file.

After you write to a binary file, you can **read** the binary file.

Refer to the Simple Binary File project in the labview\examples\File IO\Binary\Simple Binary directory for an example of how the Write to Binary File function writes an array of double-precision, floating-point values to a binary file.

Using Configuration File VIs

Use the Configuration File VIs to read and create Windows configuration settings .ini files and to write platform-specific data, such as paths, in a platform-independent format so that you can use the files these VIs generate across multiple platforms.



NOTE

You can use the Configuration File VIs for Windows configuration settings files only in the ANSI format.

Complete the following steps to use the Configuration File VIs.

1. Add the Open Config Data VI to the block diagram to open a reference to a configuration data object. A configuration data object is a preference or setting for the configuration of the operating system.
2. After you open the reference to the configuration data object, use the Read Key VI and the Write Key VI for each data type to read and write values entered by the user.

Note The Write Key VI modifies only data in memory.

3. Add the Close Config Data VI to the block diagram to close the reference to the configuration data object. You can write the current contents of the configuration data object to a configuration settings file by running the VI.



Note LabVIEW does not write the file until you call the Close Config Data VI with the **write configuration file?** input set to TRUE.

Data Logging

Reading from Datalog Files

Complete the following steps to read data from a datalog file.

1. Add the Open/Create/Replace Datalog function to the block diagram to open the datalog file for reading.
2. **Create a cluster** that matches the **data record** and wire the cluster to the **record type** input of the Open/Create/Replace Datalog function. LabVIEW uses the cluster to verify that the specified file is a datalog file containing records whose type matches **record type**. If they do not match, LabVIEW reports an error when you run the VI.
3. Add the Read Datalog function to the block diagram.
4. Wire the **refnum out** output of the Open/Create/Replace Datalog function to the **refnum** input of the Read Datalog function.
5. Close the datalog file.
 1. Add the Close File function to the block diagram.
 2. Wire the **refnum out** output of the Read Datalog function to the **refnum** input of the Close File function.
6. Right-click the **record(s)** output of the Read Datalog function and select **Create»Indicator** from the shortcut menu. An indicator is added to the front panel.
7. Run the VI. LabVIEW displays the data record in the front panel indicator.

Writing to Datalog Files

Complete the following steps to write each record to a file in the form of a cluster.

1. Add the Open/Create/Replace Datalog function to the block diagram.
2. Right-click the **operation** input of the Open/Create/Replace Datalog function and select **Create»Constant** from the shortcut menu. Select **create** from the enumerated constant that appears.
3. **Create a cluster** that matches the **data record** and wire the cluster to the **record type** input of the Open/Create/Replace Datalog function. The cluster specifies how LabVIEW writes the data to the file.
4. Add the Write Datalog function to the block diagram.
5. Wire the **refnum out** output of the Open/Create/Replace Datalog function to the **refnum** input of the Write Datalog function.
6. Create a control or constant to represent the data record and wire it to the **record(s)** input of the Write Datalog function.
7. Close the datalog file.
 1. Add the Close File function to the block diagram.

2. Wire the **refnum out** output of the Write Datalog function to the **refnum** input of the Close File function.
8. After you write to a datalog file, *read the file*.

Refer to the Datalog File project in the `labview\examples\File IO\Datalog` directory for an example of saving a simulated temperature reading and the time stamp in a LabVIEW datalog file.

Logging Front Panel Data

Each time a VI runs, *front panel data logging* saves the front panel data to a separate datalog file, which is in binary format. You can retrieve the data in the following ways:

- Use the same VI from which you recorded the data to retrieve the data interactively.
- Use the VI *as a subVI* to retrieve the data programmatically.
- Use the File I/O VIs and functions to retrieve the data.

Each VI maintains a log-file binding that records the location of the datalog file where LabVIEW maintains the logged front panel data. Log-file binding is the association between a VI and the datalog file to which you log the VI data.



NOTE

LabVIEW does not support front panel data logging through remote panels.

Changing the Log-File Binding

Complete the following steps to change the *log-file binding* to log front panel data or retrieve front panel data from a different datalog file.

1. Select **Operate»Data Logging»Change Log File Binding** to display a file dialog box.
2. Select a different datalog file or create a new one and click the **OK** button.
3. Run the VI and *log* data to the new log-file binding.

Clearing the Log-File Binding

Use *log-file binding* to associate a VI with the datalog file to use when logging or retrieving front panel data. You can have two or more datalog files associated with one VI. This might help you test or compare the VI data. For example, you could compare the data logged the first time you run the VI to the data logged the second time you run the VI.

Select **Operate»Data Logging»Clear Log File Binding** to clear the log-file binding. LabVIEW prompts you to specify a datalog file the next time you run the VI either with automatic logging enabled or when you choose to log data interactively.

Datalog Type

The data type of each record in the datalog file can be of any type. However, the data type varies depending on how you logged the data to the datalog file.

For a datalog file you [created using the Write Datalog function](#), the data type matches the data LabVIEW wrote to the datalog file.

For a datalog file you [created using front panel data logging](#), the data type is a cluster of two clusters. The first cluster contains a time stamp and the second cluster contains the front panel data. The time stamp includes a 32-bit unsigned integer that represents seconds and a 16-bit unsigned integer that represents milliseconds elapsed from LabVIEW system time. The front panel data cluster contains the data types corresponding to all front panel controls and indicators in their respective tabbing order.

Deleting a Datalog Record

Complete the following steps to delete a specific datalog record.

1. Select **Operate»Data Logging»Retrieve** and [view the record](#) you want to delete.
2. Click the **Delete Data Record** button to mark the record for deletion. If you click the button again, the record is no longer marked for deletion.
3. Select **Operate»Data Logging»Purge Data** to delete all the records you marked for deletion.
4. Click the **Commit & Exit Retrieval** button to exit retrieval mode. If you do not delete the marked records before you click the **Commit & Exit Retrieval** button, LabVIEW prompts you to delete the marked records.

Enabling Data Logging

You can perform data logging automatically or interactively.

Complete the following steps to enable automatic data logging.

1. Select **Operate»Log at Completion** to enable automatic data logging. When automatic data logging is enabled, a checkmark appears next to the **Log at Completion** menu item. Select **Operate»Log at Completion** again to disable automatic data logging.
2. Run the VI.

You also can use the Log at Finish property to enable automatic data logging programmatically.

The first time you log front panel data for a VI, LabVIEW prompts you to name the datalog file. You can use any file extension, for example .dat or .txt, for datalog files. LabVIEW

logs data each time you run the VI and appends a new record to the datalog file each additional time you run the VI. You cannot overwrite a record after LabVIEW writes it to a datalog file.

Complete the following steps to perform data logging interactively.

1. Run the VI whose data you want to log.
2. Select **Operate»Data Logging»Log** after the VI stops running.

If this is the first time you are logging the front panel data for this VI, LabVIEW prompts you to name the datalog file. Name the file with a .dat file extension.



NOTE

A waveform chart logs only one data point at a time with front panel data logging. If you wire an array to the chart indicator, the datalog file contains a subset of the array the chart displays.

Retrieving Logged Front Panel Data Using a SubVI

When you enable database access, the inputs and outputs of the subVI actually act as outputs, returning their logged data to the front panel of the current VI.



NOTE

If you display a subVI or Express VI as expandable, you cannot enable database access for that node.

Complete the following steps to retrieve logged front panel data using a subVI.

1. Click the **Select a VI** icon or text on the **Functions** palette.
2. Navigate to the *VI that logged the front panel data* and place it on the block diagram.
3. Right-click the VI and select **Enable Database Access** from the shortcut menu. A yellow box that looks like a file cabinet appears around the VI.
4. Right-click the **record #** terminal and select **Create»Control** from the shortcut menu. LabVIEW adds a numeric control to the front panel you use to enter a record number.
5. Right-click the **invalid record #** terminal and select **Create»Indicator** from the shortcut menu. LabVIEW creates a round LED that indicates whether the record is valid.
6. Right-click the **front panel data** terminal and select **Create»Indicator** from the shortcut menu. LabVIEW creates a cluster that contains the front panel data.
7. Right-click the **time stamp** terminal and select **Create»Indicator** from the shortcut menu. LabVIEW creates a cluster that contains the time stamp value.
8. Run the VI. The front panel cluster and time stamp cluster return the data to the front panel of the current VI, according to the record number wired at the input. It

also returns a Boolean value that indicates whether the indicated record number is invalid.

You can access records relative to the last logged record using negative record numbers. -1 represents the last record, -2 represents the second to the last, and so on, through -n, which represents the first record. If you wire a number outside the range -n to n - 1 to the **record #** terminal, the **invalid record #** output is TRUE, and the subVI retrieves no data.

Complete the following steps to retrieve values for specific inputs and outputs.

1. Click the **Select a VI** icon or text on the **Functions** palette.
2. Navigate to the VI that logged the front panel data and place it on the block diagram.
3. Right-click the VI and select **Enable Database Access** from the shortcut menu. A yellow box that looks like a file cabinet appears around the VI.
4. Right-click the **record#** terminal and select **Create»Control** from the shortcut menu. LabVIEW adds a numeric control to the front panel you use to enter a record number.
5. Instead of creating indicators to retrieve front panel data and time stamp data, wire directly to a terminal on the subVI and create indicators.
6. Run the VI. The VI returns data to the VI front panel according to the terminals you wired.

To disable database access, right-click a subVI and select **Disable Database Access** from the shortcut menu.

Selecting the Default Data Directory

Select a default data directory to specify the default location for any data files LabVIEW generates.



NOTE

Selecting a default data directory sets the directory for the Read From Measurement File Express VI, Write To Measurement File Express VI, Default Data Directory constant, and Default Data Directory property.

Complete the following steps to select the default data directory.

1. Select **Tools»Options** to display the **Options** dialog box.
2. Select **Paths** from the **Category** list.
3. Select **Default Data Directory** from the pull-down menu.
4. Remove the checkmark from the **Use default** checkbox to access the path preferences options.
5. Enter a new path in the text box or click the **Browse** button to select a directory and click the **Replace** button. You also can click the **!** button to enter the **<osdatadir>** tag

into the **Path** field automatically. `<osdatadir>` represents the path to the default file directory on the operating system.

6. Click the **OK** button.



Note All changes to the default data directory options take effect when you restart LabVIEW.

Viewing Logged Data Interactively

After you [log data](#), you can view it interactively.

Complete the following steps to view the logged front panel data after the VI that logged data stops running.

1. Select **Operate»Data Logging»Retrieve**. The toolbar of the **Front Panel** window becomes the data retrieval toolbar. The highlighted number indicates the data record you are viewing. The numbers in square brackets indicate the range of records you logged for the current VI. The date and time indicate when you logged the selected record.
2. View the next or previous record by clicking the increment or decrement buttons. You also can press the up or down arrow keys.
3. Click the **Commit & Exit Retrieval** button to exit data retrieval mode and return to the VI whose data log you were viewing.

LabVIEW Measurement Files

The LabVIEW Measurement `.1vm` format is a text-based file format for one-dimensional data. This text-based measurement file is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. The `.1vm` file includes header information about the data, such as the date and time the data was generated. LabVIEW saves data with up to six digits of precision in a `.1vm` file.

The `.1vm` format is easy to parse and easy to read when imported into a spreadsheet application such as Microsoft Excel or a text editor such as Notepad. The `.1vm` format supports multiple data sets, grouping of data sets, and the adding of data sets to existing files.

The `.1vm` file uses commas as a delimiter between numbers. To convert data in a `.1vm` file from a string to a number, specify the decimal separator as a period, using the localization code format specifier.

This file format is not designed for high-performance or for very large data sets, as is the case with all text-based formats.



NOTE

For very large data sets, use the binary file format.

Reading Data from LabVIEW Measurement Files

Complete the following steps to use the Read From Measurement File Express VI to read numeric or waveform data from a LabVIEW measurement file. Use the Write To Measurement File Express VI to *write data to* a measurement file.

1. Add the Read From Measurement File Express VI to the block diagram.
2. The **Configure Read From Measurement File** dialog box pops up. Inside the dialog box, the **Filename** text box at the top of the pane displays the full path to the input file. Click the **Browse** button if you want to select a different file.
3. In the **File Format** section, select the format of the file from which you want to read data.
4. Configure the remaining sections in the **Configure Read From Measurement File** dialog box and click the **OK** button to save the configuration and close the dialog box.
5. Right-click the **Signals** output of the Read From Measurement File Express VI and select **Create»Numeric Indicator** or **Create»Graph Indicator** from the shortcut menu.
6. *Wire all error in and error out terminals* on the block diagram.
7. Run the VI. LabVIEW displays the numeric or waveform data from the file specified in the **Filename** section of the **Configure Read From Measurement File** dialog box in a front panel indicator.

Writing Data to LabVIEW Measurement Files

Complete the following steps to use the Write To Measurement File Express VI to write numeric or waveform data to a LabVIEW measurement file. Use the Read From Measurement File Express VI to *read data from* a measurement file.

1. Add the Write To Measurement File Express VI to the block diagram. Notice that the **Filename** text box in the **Configure Write To Measurement File** dialog box displays the full path to the output file.
2. In the **File Format** section, select the format for the file to which you want to write data.
3. In the **Segment Headers** section, select the header option for the file.
4. Configure the **X Value Columns** section if you selected **Text (LVM)** in the **File Format** section and you plan to view the data in a spreadsheet or text editor.

5. Configure the remaining sections in the **Configure Write To Measurement File** dialog box and click the **OK** button to save the configuration and close the dialog box.

**NOTE**

LabVIEW automatically generates column titles from the data forms that are wired to this VI. To change the column titles, use the Set Dynamic Data Attributes Express VI.

6. Add the Simulate Signal Express VI to the block diagram and configure the VI.
7. Wire the output of the Simulate Signal Express VI to the **Signals** input of the Write To Measurement File Express VI.
8. *Wire all error in and error out terminals* on the block diagram.
9. Run the VI. The VI writes the data from the Simulate Signal Express VI to the file specified in the **Filename** section of the **Configure Write To Measurement File** dialog box.
10. To view the data you saved, open the output file with a spreadsheet or word processing application if it is a .lvm file. Use the file path displayed in the **Filename** section of the **Configure Write To Measurement File** dialog box to access the file.

Reading from Spreadsheet Files

Complete the following steps to read from a spreadsheet file.

1. Add the Read Delimited Spreadsheet VI to the block diagram.
2. *Select the polymorphic instance* of the Read Delimited Spreadsheet VI that corresponds to the type of data you want to read from the spreadsheet file.
3. Create a *relative path* to the file from which you want to read data.
4. Wire the **appended path** output to the **file path** input of the Read Delimited Spreadsheet VI.
5. Right-click the **all rows** output and select **Create»Indicator**.
6. Run the VI. LabVIEW returns the data from the spreadsheet file in the **all rows** indicator.

Writing to Spreadsheet Files

To write data to a text-based spreadsheet file, use LabVIEW to format data as a spreadsheet string and write the string to a file. The following are examples of using LabVIEW to write data to spreadsheet files depending on the type of data to write and your performance needs:

- To write 1D or 2D array data to a spreadsheet file one time, use the Write Delimited Spreadsheet File VI.
- To write data of different data types and of any dimension, use the Array to Spreadsheet String function.

- If you require multiple write operations, use a disk streaming programming pattern to stream data to spreadsheet files.



RELATED INFORMATION

[Streaming Data to Spreadsheet Files](#) on page 881

[Write Delimited Spreadsheet](#)

[Array To Spreadsheet String Function](#)

Streaming Data to Spreadsheet Files

Use **disk streaming** when you require multiple write operations to a spreadsheet file. Disk streaming reduces the number of times LabVIEW interacts with the operating system to open and close a file. You can use disk streaming while formatting data as a spreadsheet string and writing the string to a file.

What to Use

Use the following block diagram objects to format data and to stream the data to a file.

Open/Create/Replace File Function	While Loop	Format Into String Function	Write To Text File Function	Close File Function
				

What to Do

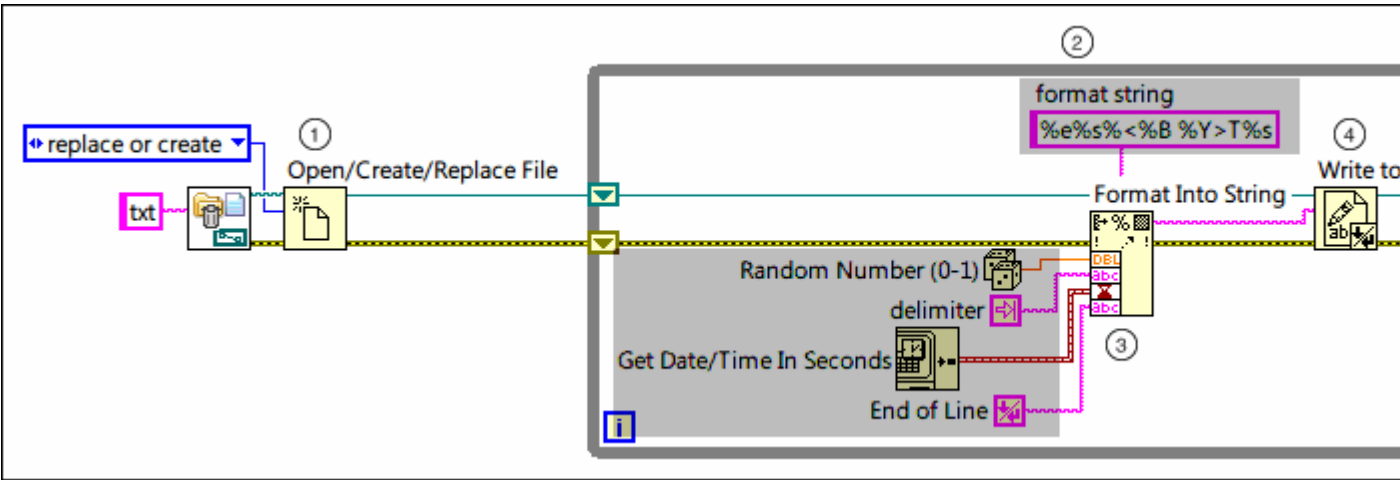
Create the following block diagram to format data of different data types as a spreadsheet string and to stream the string to a file.



NOTE

To format data of one data type as a spreadsheet string, use the Array to Spreadsheet String function.

Customize the gray sections for your unique programming goals.



The following list describes important details about the previous diagram:

① ⑤	Place the Open/Create/Replace File function and Close File function outside the While Loop to create a typical disk streaming operation.
②	The While Loop enables this VI to continuously format and write data to a file without the overhead associated with opening and closing the file in each iteration. Use shift registers to pass values from previous iterations to the next iteration.
③	Wire input parameters from outside the While Loop to the input 1..n inputs of the Format Into String function. You can wire delimiters, such as end-of-line constant, tab constant, and so on, as input parameters to separate input data. Specify the format string input using <i>format specifiers</i> to format the input parameters as a spreadsheet string.
	Note You can also use the Format Into File function in place of the Format Into String function and Write to Text File function to format data as text and write the text to a file.
④	The Write to Text File function writes the spreadsheet string to the file.

You can improve the performance of disk streaming by buffering data using a *shift register*. First, initialize the shift register with a buffer of data. Next, write data to the buffer. When the buffer is full, write the buffer to disk.

Examples

Refer to the Tab-Delimited Data project in the labview\examples\File IO\Spreadsheet\Tab-Delimited Data directory for an example of streaming data to spreadsheet files. This example uses the Array to Spreadsheet String function which converts data of one data type to a spreadsheet string and writes the string to a file.

Refer to the Buffered Stream to Tab-Delimited Text File VI in the labview\examples\File IO\Spreadsheet\Tab-Delimited Data directory for an example of buffering data when streaming data to spreadsheet files.

TDM/TDMS Files

To reduce the need to design and maintain your own file format for measurement data, National Instruments defines the flexible technical data management (TDM) data model, which is accessible through LabVIEW, LabWindows™/CVI™, Measurement Studio, SignalExpress, and DIAdem. The TDM data model also enables you to write well-organized data regardless of which third-party products you are using.

The TDM data model supports the following two file formats:

- (Windows) **TDM** —The TDM file format saves the raw data that the measurement hardware acquires and the meta data in separate files. Meta data can be the name and unit of a measurement channel, the name of the test engineer, and other information. The meta data is in the XML format with the `.tdm` extension. The raw data is in a compact binary format with the `.tdx` extension.
- **TDM Streaming (TDMS)** —The TDMS file format saves both the raw data and the meta data in the binary format in one file with the `.tdms` extension.



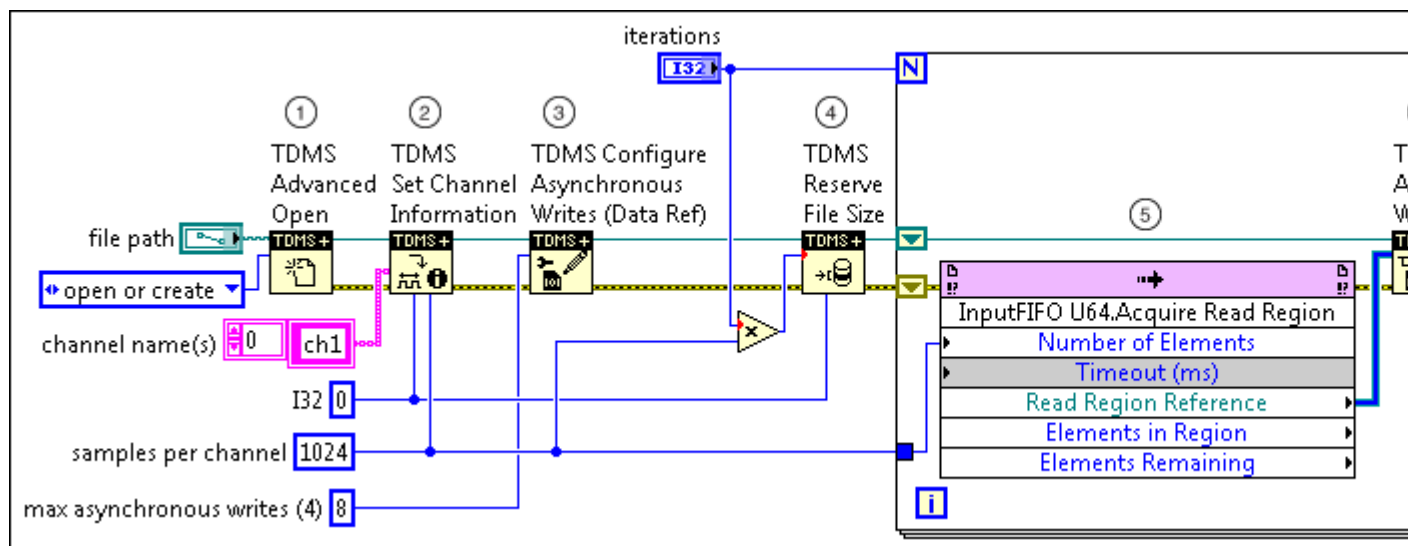
NOTE

When creating or opening a `.tdms` file, LabVIEW might automatically create a `.tdms_index` file. LabVIEW uses the `.tdms_index` file to speed up random access to the `.tdms` file. The `.tdms_index` file is optional in TDMS applications. If you have disk space limitations, configure the TDMS Open function to stop generating this file.

The TDM data model structures the meta data into groups and channels, and adds user-defined properties. You can identify whether an element in a file is a group, a channel, or a property.

Streaming External Data to a TDMS File (Windows)

The following figure illustrates guidelines for using the TDMS Advanced Data Reference I/O functions to stream external data to a `.tdms` file.



The following events occur in the previous figure:



NOTE

You must have NI-RIO device drivers and the LabVIEW FPGA Module installed to transfer data using direct memory access.



RELATED INFORMATION

[TDM/TDMS Files](#) on page 883

Reading from Text Files

Complete the following steps to read characters or strings from a text file.

1. Add a string indicator to the front panel.
2. Add the Read from Text File function to the block diagram.
3. Right-click the **file** input of the Read from Text File function and select **Create»Constant**. A path constant appears. Enter an absolute path for the file you wish to read from into the constant.



TIP

To avoid typing out the path, you can drag a file from the desktop and drop it into the constant.

4. Wire the **text** output of the Read from Text File function to the string indicator.
5. Run the VI. The VI reads the text from the file you specified in the constant and displays the text in the front panel string indicator.

Refer to the Tab-Delimited Data project in the `labview\examples\File IO\Spreadsheet\Tab-Delimited Data` directory for an example that reads data from a text file and plots the data.

Writing to Text Files

Complete the following steps to write to a text file.

1. Add a string control to the front panel window.
2. Enter into the string control the text that you want to write to the text file.
3. Add the Write to Text File function to the block diagram.
4. Wire an absolute path to the **file** input of the Write to Text File function.
5. Wire the string control to the **text** input of the Write to Text File function.
6. Run the VI. The VI writes the text from the front panel window string control to the file wired to the **file** input.

After you write to a text file, [read the file](#).

Refer to the Tab-Delimited Data project in the `labview\examples\File IO\Spreadsheet\Tab-Delimited Data` directory for an example of writing a text spreadsheet of simulated acquired data to a file.

Converting LabVIEW Data to XML

Converting LabVIEW data to XML formats the data so that when you save the data to a file, you easily can identify the value(s), name(s), and type of the data from the tags that describe the data.

Complete the following steps to convert LabVIEW data to XML format.

1. Add the Flatten To XML function to the block diagram.
2. Wire the data you want to convert to the **anything** input of the Flatten To XML function.
3. Right-click the **xml string** output of the function and select **Create»Indicator** from the shortcut menu. The block diagram now has a string indicator terminal, and the front panel has a string indicator.
4. Run the VI. LabVIEW converts the data type to XML format according to the [LabVIEW XML schema](#). The front panel **xml string** indicator displays the XML-formatted data.

Refer to the `labview\examples\File IO\XML\Flatten and Unflatten XML\Flatten and Unflatten XML.lvproj` for examples of converting LabVIEW data to XML format.

Converting XML to LabVIEW Data

The XML string you want to convert to LabVIEW data must conform to the [LabVIEW XML schema](#). You cannot convert more than one data type from an XML string.

Complete the following steps to convert an XML string to LabVIEW data.

1. Add the Unflatten From XML function to the block diagram.
2. Wire the XML string that contains the data you want to convert to the **xml string** input of the Unflatten From XML function.
3. Specify the data type of the data in the XML string by wiring a constant or control to the **type** input of the function. For example, if the XML string is an array of double-precision, floating-point numbers, wire that data type to the **type** input.
4. Right-click the **value** output of the function and select **Create»Indicator** from the shortcut menu to create an indicator of the same data type as the **type** input. A block diagram indicator terminal appears, and a front panel indicator appears.
5. Run the VI. LabVIEW converts the **xml string** to a LabVIEW data type of **type** according to the LabVIEW XML schema. The **value** indicator displays the LabVIEW data.

Refer to the `labview\examples\File IO\XML\Flatten and Unflatten XML\Flatten and Unflatten XML.lvproj` for examples of converting XML to LabVIEW data.

Automatically Cleaning Up the Block Diagram

LabVIEW can reroute all existing wires and reorder objects on the block diagram.

Complete the following steps to reroute all existing wires and reorder all existing objects on the block diagram.

1. Configure the clean up options. Select **Tools»Options** to display the Options dialog box and select **Block Diagram** from the **Category** list. On the Block Diagram page, you can configure LabVIEW to automatically move controls to the left side of the containing diagram and indicators to the right side of the containing diagram, place a certain number of pixels between block diagram objects and wires, and compact the block diagram layout.
2. (Optional) Right-click a structure and select **Exclude from Diagram Cleanup** to allow LabVIEW to move the structure without reorganizing the contents of the structure.
3. Select **Edit»Clean Up Diagram**. You also can click the **Clean Up Diagram** button  on the block diagram toolbar or press the <Ctrl-U> keys. (macOS) Press the <Command-U> keys. (Linux) Press the <Alt-U> keys.



NOTE

You also can select specific objects to clean up, such as wires or individual nodes. To select multiple objects, press <Shift>-click. When you select objects to clean up, the **Clean Up Diagram** button changes to indicate **Partial Diagram Clean Up**. LabVIEW cleans up only the objects that you select and not the entire block diagram. When you select specific wires to clean up, LabVIEW reorganizes only the wires and not objects attached to the wires.

You also can reroute wires and reorder objects you select on the block diagram.

Complete the following steps to reroute and reorder a selection of existing wires and objects on the block diagram automatically.

1. (Optional) Right-click a structure and select **Exclude from Diagram Cleanup** to allow LabVIEW to move the structure without reorganizing the contents of the structure.
2. Click and drag a selection rectangle around the items you want to clean up.



Note LabVIEW does not clean up the block diagram if you do not select any structures and none of the objects you select overlap each other, or if the selection contains only wires without corresponding inputs and outputs.

1. (Optional) Press the <Shift> key and click and drag a selection rectangle around additional objects to create multiple selections. You also can press the <Shift> key while you click objects you want to deselect.
2. Select **Edit»Clean Up Selection**.

Creating Controls and Indicators on the Block Diagram

You can create controls and indicators on the block diagram. Position inputs, or controls, on the left side of the VI, function, or Express VI on the block diagram. Position outputs, or indicators, on the right side.

If you *display subVIs or Express VIs as expandable nodes*, *resize* the node to view the terminals to make wiring easier. If you *display subVIs or Express VIs as icons*, *display the terminals* to make wiring easier.

Complete the following steps to create a control interactively on the block diagram.

1. Right-click the input terminal of a VI, function, or Express VI on the block diagram.
2. Select **Create»Control** from the shortcut menu to create a control of the correct data type. LabVIEW creates the control and wires it to the input terminal you right-clicked.

Complete the following steps to create an indicator interactively on the block diagram.

1. Right-click the output terminal of a VI, function, or Express VI on the block diagram.
2. Select **Create»Indicator** from the shortcut menu to create an indicator of the correct data type. LabVIEW creates the indicator and wires it to the output terminal you right-clicked.

Right-click a dynamic data type output terminal of an Express VI, VI, or function on the block diagram and select **Create»Graph Indicator** from the shortcut menu to display the

data in a graph or select **Create»Numeric Indicator** from the shortcut menu to display the data in a numeric indicator.

Select **File»VI Properties** and select Editor Options from the **Category** pull-down menu to change the style of the control or indicator LabVIEW creates when you right-click a terminal and select **Create»Control** or **Create»Indicator** from the shortcut menu. Select **Tools»Options** and select Front Panel from the **Category** list to change the style of the control or indicator LabVIEW creates in new, blank VIs when you right-click a terminal and select **Create»Control** or **Create»Indicator** from the shortcut menu.

Creating and Editing User-Defined Constants

Constants are terminals on the block diagram that supply fixed data values to the block diagram. Universal constants are constants with fixed values, such as pi π and infinity ∞ . User-defined constants are constants you define and edit before you run a VI.

Complete the following steps to create a user-defined constant.

1. Right-click an input terminal of a VI or function and select **Create»Constant** from the shortcut menu. LabVIEW creates a constant of the default data type of the input terminal and highlights the value.
2. Type the value you want to assign to the constant and press the <Enter> key. You also can press the up and down arrow keys to assign a value to the constant. LabVIEW resizes the constant to fit its contents. You can resize a string constant the same way you [resize a label](#).
3. Use the Operating or Labeling tool to click the constant and edit its value. If [automatic tool selection is enabled](#), double-click the constant to switch to the Labeling tool and edit the value.



NOTE

LabVIEW uses [constant folding](#) to optimize the performance of VIs. You can display constant folding on the block diagram by selecting **Tools»Options**, selecting **Block Diagram** from the **Category** list, and placing a checkmark in the **Show hash marks for constant folded and loop invariant wires** checkbox.

You also can create a constant in the following ways:

- [Search](#) for a constant or select it on the **Functions** palette and add it to the block diagram. Most constants are located at the bottom or top of their palettes.
- Search the LabVIEW Help for a constant and click the **Add to the block diagram** button to place the constant on the cursor so you can add it to the block diagram. Click the **Find on the palette** button to highlight the constant on the **Functions** palette.
- Drag a front panel control to the block diagram. LabVIEW creates a constant with the value of the front panel control at the time you dragged it to the block diagram. The value of a refnum constant is Not A Refnum. The control remains on the front panel. Changing the value of the front panel control does not affect the constant value on the block diagram and vice versa.

Displaying Terminals as Icons or Data Types

You can configure front panel controls or indicators to appear as icon or data type terminals on the block diagram. By default, front panel objects appear on the block diagram as icon terminals. Use icon terminals to display the types of front panel objects on the block diagram, in addition to the data types of the front panel objects. Use data type terminals to conserve space on the block diagram.

Complete the following steps to display a terminal as a data type on the block diagram.

1. Right-click a terminal on the block diagram.
2. Select **View As Icon** from the shortcut menu to remove the checkmark. To display terminals as icons, select **View As Icon** from the shortcut menu again.



NOTE

Icon terminals are larger than data type terminals, so you might unintentionally obscure other block diagram objects when you convert a data type terminal to an icon terminal.

You can configure LabVIEW to display terminals for new front panel objects you create as icons or data types by default by setting the **Place front panel terminals as icons** option on the Block Diagram page of the **Options** dialog box.

Removing and Rewiring Block Diagram Objects

When you remove and rewire a block diagram object, LabVIEW removes the selected object and any wires and constants connected to the object, and connects wires of identical data types that were wired to the inputs and outputs of the deleted object.



NOTE

Deleting block diagram objects by pressing <Delete> or <Backspace> only removes objects. LabVIEW does not rewire inputs and outputs of the deleted objects.

To remove and rewire one or more block diagram objects, drag a selection rectangle around the object(s) you want to remove. Choose one of the following methods to remove and rewire the objects in the selection rectangle:

- Right-click the selection rectangle and select **Remove and Rewire**.
- Press <Ctrl-Space> to open the Quick Drop dialog box and press the **Quick Drop keyboard shortcut** <Ctrl-R>.

LabVIEW also removes any decorations, including free labels, that are in the selection rectangle.

Inserting Block Diagram Objects

Complete the following steps to insert an object into a wired block diagram.

1. Right-click the wire at the position where you want to insert the object.



TIP

To connect the wires to the most appropriate terminals at the bottom of the object, right-click the wire itself or directly above the wire. To connect the wires to the most appropriate terminals at the top of the object, right-click directly below the wire.

1. Select **Insert** from the shortcut menu to display the **Functions** palette.
2. Select a VI or function on the **Functions** palette to insert the new object on the wire. If the number of terminals or data types in the objects are different, you might have *broken wires*.

Replacing Block Diagram Objects

You can replace a block diagram object with a different object without losing all the editing options you selected for the object up to that point.

Complete the following steps to replace a block diagram object.

1. Right-click a block diagram object.
2. Select **Replace** from the shortcut menu to display the **Functions** palette.



NOTE

Depending on the block diagram object you select, you might see different replacement options. For example, if you right-click a While Loop, you can select to replace it with only a For Loop or a (Real-Time, Windows) Timed Loop.

1. Select a VI or function on the **Functions** palette to replace the object on the block diagram. If the number of terminals or data types in the objects are different, you might have *broken wires*.



Note When you select a VI with the same name as a VI in memory, a dialog box prompts you to replace the VI or cancel the operation. If you select **Replace**, LabVIEW replaces all the VIs in memory with the VI you selected.

Resizing Nodes

You can resize most nodes on the block diagram, such as subVIs, Express VIs, Property Nodes, and several functions such as the Array functions.



NOTE

You only can resize subVIs and Express VIs when you view them as expandable nodes. The Index Array, Replace Array Subset, Insert Into Array, and Array Subset functions automatically resize to match the dimensions of the input array you wire.

Complete the following steps to resize a node on the block diagram.

1. Move the Positioning tool over the node. Resizing handles appear at the top and bottom of the node.
2. Move the cursor over a resizing handle to change the cursor to the resizing cursor.
3. Use the resizing cursor to drag the border of the node down to display additional terminals.
4. Release the mouse button.

To cancel a resizing operation, drag the node border past the block diagram window before you release the mouse button.

You also can insert a single input or output in a subVI or Express VI.



RELATED INFORMATION

[Displaying SubVIs and Express VIs as Icons or Expandable Nodes](#) on page 1385

[Inserting and Removing Inputs and Outputs](#) on page 1385

Reusing Snippets of Code

You can save snippets of code, or a VI snippet, from the block diagram to reuse later or to share with other LabVIEW users. LabVIEW embeds the code into a .png image file. The image shows a picture of your code and contains the actual code you specify.

Complete the following steps to reuse a snippet of code from the block diagram.

1. Select a section of code on the block diagram.



NOTE

Do not select a section of code that contains an unsaved subVI. If you include a subVI in the section of code, you first must save the subVI on disk.

1. Select **Edit»Create VI Snippet from Selection**.

2. In the dialog box that appears, select or browse to the directory where you want to save the VI snippet.
3. Click the **OK** button.
4. (Windows and macOS) (Optional) Drag the .png file from the directory where you saved it and drop it onto the block diagram. LabVIEW drops the actual code that is in the file and automatically wires controls and indicators to wires in the section of code that have loose ends.
5. (Optional) Share the .png file with other LabVIEW users in one of the following ways.
 - Attach the file to any Web document such as an email. (Windows and macOS) Other users can download the file and then drag and drop the file onto the block diagram to use the code.
 - Embed the image in an HTML page such as a message forum. (Windows and macOS) Other users can drag the image onto the block diagram to use the code.



Note (Linux) You can save and share VI snippets to show images of code, but you cannot drag and drop VI snippets onto the block diagram to use the code.

LabVIEW does not embed subVIs in the .png file. Any subVIs called in the VI snippet must be present on the computer when a user drops the .png file onto the block diagram. Otherwise, the VI snippet contains broken code.

When you open a VI snippet .png file with image editing software, LabVIEW displays the version of LabVIEW in which the VI snippet was saved in the upper right-hand corner of the image. To use the code of a VI snippet, the version of LabVIEW you are running must be the same as or later than the version of LabVIEW in which the VI snippet was saved.



Note If you save a VI snippet .png file with the image editing software, that software might remove the code from the .png file. If you then add the .png file to the block diagram, only the image appears on the block diagram, not the code. To avoid losing the code, do not save the .png file with the image editing software.

Selecting the Instance of a Polymorphic VI Manually

When you place a polymorphic VI on the block diagram, the default polymorphic VI is the first instance in the **Instance VIs** list of the Polymorphic VI window. When you create a control, indicator, or constant from the polymorphic VI terminals, the instance determines the data type of each object. To change the default instance, use the arrow buttons in the **Polymorphic VI** window to move a selected instance in the **Instance VIs** list to the top of the list.

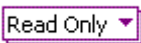
For most polymorphic VIs, the data types you wire to the inputs of the polymorphic VI determine the instance to use. For example, if you wire a string constant to the **default value** input terminal of the Read Key VI, LabVIEW selects the String instance of the polymorphic VI, and the **value** output terminal changes to a string data type.

If the data types you wire to the polymorphic VI inputs do not determine the instance to use, you must select the instance manually. If you manually select an instance of a polymorphic VI, the VI no longer behaves as a polymorphic VI because it accepts and returns only the data types of the instance you select.

Use the following ways to select the instance of a polymorphic VI manually. For built-in polymorphic VIs, the Context Help window displays a description of each instance as you move the cursor through the shortcut menus.

- Right-click the polymorphic VI on the block diagram, select **Select Type** from the shortcut menu, and select the instance to use.
- Use the Operating tool to click the polymorphic VI selector and select an instance from the shortcut menu. Right-click the polymorphic VI on the block diagram and select **Visible Items»Polymorphic VI Selector** from the shortcut menu to display the selector. The following illustration shows an example of a polymorphic VI

selector.



To change the polymorphic VI to accept all the handled data types again, right-click the polymorphic VI and select **Select Type»Automatic** from the shortcut menu, or use the Operating tool to click the polymorphic VI selector and select **Automatic** from the shortcut menu.



TIP

To find the polymorphic VI node associated with a polymorphic VI selector, right-click the selector and select **Find Polymorphic VI** from the shortcut menu.

Using Wires

Automatically Wiring Objects

As you move a selected object close to other objects on the block diagram, LabVIEW draws temporary wires to show you valid connections. When you release the mouse button to add the object to the block diagram, LabVIEW automatically connects the wires. You also can automatically wire objects already on the block diagram. LabVIEW connects the terminals that best match and does not connect the terminals that do not match.

Toggle automatic wiring by pressing the spacebar while you move an object using the Positioning tool. By default, automatic wiring is enabled when you select an object from the **Functions** palette or when you *clone an object* already on the block diagram. Automatic wiring is disabled by default when you use the Positioning tool to move an object already on the block diagram. When automatic wiring is enabled, the selected object retains its appearance when you drag it. When automatic wiring is disabled, the selected object appears as a dotted outline when you drag it.

Complete the following steps to practice wiring objects automatically as you add them to the block diagram.

1. If necessary, **enable automatic wiring** and set the maximum and minimum distances for objects to wire automatically.
2. Add a numeric control to the front panel window.
3. Press the <Ctrl-E> keys to display the block diagram. (macOS) Press the <Command-E> keys. (Linux) Press the <Alt-E> keys.
4. Select the Add function from the **Functions** palette.
5. Move the left side of the function close to the right side of the numeric control terminal on the block diagram. As you move the function close to the terminal, LabVIEW draws temporary wires to show you valid connections. Make sure the function is within the maximum and minimum distances you set in step 1.
6. When a temporary wire appears between the numeric control terminal and the top left terminal of the Add function, click the mouse button to add the function and wire the objects.
7. Make sure you wire all required terminals. Otherwise, the VI is broken and will not run. Use the **Context Help** window to see which terminals a block diagram node requires. The labels of **required terminals** appear bold in the **Context Help** window.

Configuring Automatic Wiring

Complete the following steps to enable or configure automatic wiring.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Block Diagram from the **Category** list.
3. Place a checkmark in the **Enable auto wiring** checkbox to enable automatic wiring.
4. Edit the **Minimum distance** and **Maximum distance** to set the minimum distance in pixels objects must have between them to wire automatically and the maximum distance in pixels objects can be separated and still wire automatically.

Deleting Wires

Complete the following steps to delete a portion or all of a wire.

1. Use the Positioning tool to single-click the wire to select one segment of the wire, double-click to select a branch, or triple-click to select the entire wire.
2. Press the <Delete> key or the <Backspace> key.

You also can right-click a wire branch and select **Delete Wire Branch** from the shortcut menu to delete the wire branch.



NOTE

You can change the path of an existing wire without first deleting the wire. Create the new wire path between the same terminals, and LabVIEW deletes the previous wire path.

Displaying Terminals for VIs, Functions, and Express VIs

Display the connector pane terminals for a subVI, function, or Express VI to make wiring easier.



NOTE

If you *display a subVI or Express VI as expandable*, you cannot display the terminals for that node.

Complete the following steps to display the terminals for a subVI, function, or Express VI.

1. Right-click the VI or function.
2. Select **Visible Items»Terminals** from the shortcut menu.

To return to the icon, right-click the VI or function and select **Visible Items»Terminals** from the shortcut menu to remove the checkmark next to the **Terminals** shortcut menu item.

Moving Wires

Complete the following steps to reposition a portion or all of the wire.

1. Use the Positioning tool to single-click the wire to select one segment of the wire, double-click to select a branch, or triple-click to select the entire wire.
2. Drag the wire to the new location. You also can *restrict the direction and increments of movement*.

Notice that adjacent, unselected segments stretch to accommodate the new location.

Complete the following steps to reposition one or more wire segments.

1. Use the Positioning tool to single-click the wire to select one segment of the wire.
2. (Optional) Press the <Shift> key and use the Positioning tool to single-click the wire to select another segment of the wire.
3. Drag the segments to the new location. You also can *restrict the direction and increments of movement*.

You can select and drag multiple wire segments, even discontinuous segments, simultaneously.

Moving a tunnel sometimes creates an extra wire segment, called a *hidden wire segment*, that lies beneath the structure border. You cannot select and drag this segment because it

is hidden. If you are unsure about which wire is connected to a tunnel, triple-click the wire to view the entire wire path.



NOTE

You can change the path of an existing wire without selecting and moving the wire. Create the new wire path between the same terminals, and LabVIEW deletes the previous wire path.

Routing Wires

LabVIEW automatically finds a route for a wire as you wire it. LabVIEW routes a wire around existing objects on the block diagram, such as loops and structures. LabVIEW also routes a wire to decrease the number of bends in the wire. When possible, automatically routed wires from control terminals exit the right side of the terminal, and automatically routed wires to indicator terminals enter the left side of the terminal.

To automatically route an existing wire, right-click the wire and select **Clean Up Wire** from the shortcut menu.



NOTE

You also can *reroute all existing wires and rearrange all existing objects* on the block diagram automatically.

Complete the following steps to temporarily disable automatic wire routing and route a wire manually.

1. Use the Wiring tool to click a terminal and release the mouse.
2. Press the <A> key. LabVIEW temporarily disables automatic wire routing for the current wire.
3. (Optional) Click the mouse button to set the wire and then move the cursor in a new direction. Press the spacebar to switch the wire direction between horizontal and vertical.
4. Click another terminal to wire the objects and end the wire. LabVIEW enables automatic wire routing again.

You also can temporarily disable automatic routing after you click to start or set a wire by holding down the mouse button while you wire to another terminal or set point and then releasing the mouse button. After you release the mouse button, LabVIEW enables automatic wire routing again.

You can disable automatic wire routing for all new wires by *setting the appropriate block diagram option*.

Selecting Wires

Use the Positioning tool to single-click the wire to select one segment of the wire, double-click to select a branch, or triple-click to select the entire wire.

To add a wire to a group of selected wires, press the <Shift> key while you select the new wire.

Use the Positioning tool to drag a selection rectangle around the parts of the wire you want to select.

After you select the wire, you can [delete](#) it or [move](#) it.

Wiring Block Diagram Objects Manually

You transfer data between block diagram objects through wires. Use the Wiring tool to connect the terminals of a VI or function to the terminals of another VI or function. Use [automatic wiring](#) to wire objects as you place them on the block diagram.

Complete the following steps to practice wiring on the block diagram.

1. Place two VIs or functions on the block diagram.
2. Display the Context Help window and move the cursor over a VI or function. The **Context Help** window lists labels next to each terminal of the VI or function.
3. Use the labels next to each terminal in the **Context Help** window to determine to which terminal you want to wire.
4. Use the Wiring tool to move the cursor over an output terminal of a VI or function. When you move the Wiring tool over the terminal, the terminal blinks, and a tip strip appears.
5. Use the Wiring tool to click the terminal and release the mouse. As you move the cursor across the block diagram, LabVIEW draws a wire between the terminal and the Wiring tool as though the wire were unwinding from a spool.



TIP

To cancel a wire you started, press the <Esc> key or right-click the mouse button.

1. Without holding down the mouse button, move the cursor to an input terminal of another VI or function. The second terminal blinks, and a tip strip appears.
2. Use the Wiring tool to click the second terminal. Wiring to the terminal might create a broken wire. You must [correct the broken wire](#) before you can run the VI.
3. Make sure you wire all required terminals. Use the **Context Help** window to see which terminals a block diagram node requires. The labels of [required terminals](#) appear bold in the **Context Help** window. Otherwise, the VI is broken and will not run.



Tip To bend a wire while wiring, click to tack down the wire and move the cursor in a perpendicular direction. To tack down a wire and stop building it without connecting the wire, double-click.

To wire from or to an existing wire, complete the previous steps, starting or ending the operation on the existing wire. The wire blinks when the Wiring tool is correctly positioned to fasten a new wire to the existing wire. To wire from an existing branch when *automatic tool selection is enabled*, move the cursor toward the wire until the cursor changes to the Wiring tool. You also can right-click the existing wire and select **Create Wire Branch** from the shortcut menu.

You can *delete wires*, *move wires*, *select wires*, and *wire structures*.

Wiring Techniques to Avoid

Improve the appearance of the block diagram by wiring efficiently. Poor wire organization might not produce errors, but it can make the block diagram difficult to read and debug or make the VI appear to do things it does not do.



NOTE

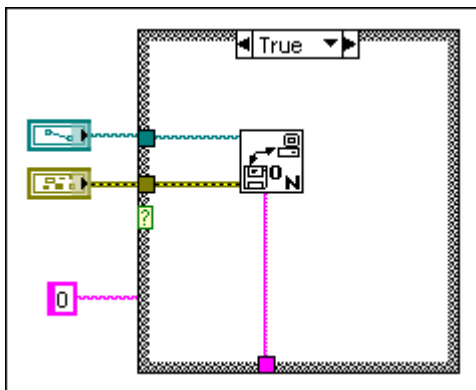
If you are unsure of what connects to a wire, double-click to select a branch or triple-click to select the entire wire.

Wires Under Objects

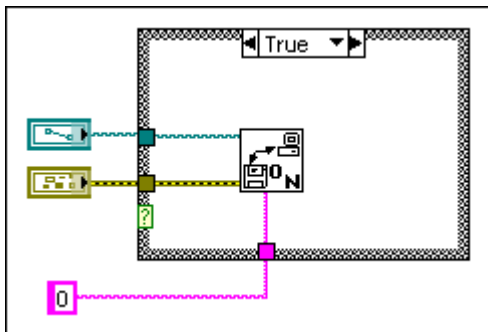
Avoid placing objects on top of wires. Placing a terminal or icon on top of a wire makes it appear as if a connection exists when it does not. Dragging a wire through an icon or terminal also appears to make a connection, but the wire is actually behind the icon.

Hidden Wire Segments

Avoid wiring under a structure border or between overlapped objects because LabVIEW might hide some segments of the resulting wire. You inadvertently can create hidden wire segments by moving a tunnel or by enlarging a structure, as shown in the following illustration.



You can drag the wire segment connected to the constant, move the constant, or resize the structure so the hidden wire segments reappear, as shown in the following illustration. You also can right-click the wire and select **Clean Up Wire** from the shortcut menu to show hidden segments of the wire. To automatically reroute all existing wires and rearrange objects on the block diagram, select **Edit»Clean Up Diagram**. You also can clean up the block diagram by clicking the **Clean Up Diagram** button  on the block diagram toolbar or pressing the <Ctrl-U> keys. (macOS) Press the <Command-U> keys. (Linux) Press the <Alt-U> keys.



Wiring Structures

You can wire directly from a terminal outside a structure to a terminal inside the structure. LabVIEW creates a tunnel where the wire crosses the structure boundary. A tunnel allows data to flow in and out of loops and structures.

Wiring to Avoid

Avoid the following when wiring structures:

- Overlapping tunnels.
- Wiring from multiple frames of a Stacked Sequence structure.
- Failing to wire a tunnel in all cases of a Case structure.
- Assigning more than one value to a sequence local terminal.

For structures wired to only constants, LabVIEW uses constant folding to optimize performance. You can configure LabVIEW to display hash marks inside constant folded

structures by selecting **Tools»Options»Block Diagram** and placing a checkmark in the **Show hash marks for constant folded structures** checkbox in the Block Diagram page.



RELATED INFORMATION

[Overlapping Tunnels](#) on page 901

[Wiring from Multiple Frames of a Stacked Sequence Structure](#) on page 901

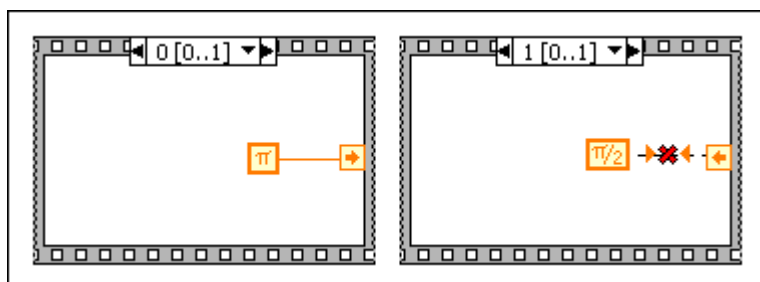
[Failing to Wire a Tunnel in All Cases of a Structure](#) on page 900

[Assigning More Than One Value to a Sequence Local Terminal](#) on page 900

[Block Diagram Objects](#) on page 126

Assigning More Than One Value to a Sequence Local Terminal

You can assign a value to the sequence local terminal of a Stacked Sequence structure in only one frame, but you can use the value in all subsequent frames. On the left side of the following example, the value π is assigned to the sequence local terminal in frame 0. If you try to assign another value to this same sequence local terminal in frame 1, you create a *broken wire*, as shown on the right side of the example.



Failing to Wire a Tunnel in All Cases of a Structure

A common error with structures with multiple cases is neglecting to complete the wiring in all the cases. When you wire to an object outside the structure, an error results if you do not connect a source in all the cases.

Consider configuring the tunnels to wire the input and output tunnels automatically in unwired cases.



RELATED INFORMATION

[Executing One or More Sections of Code Using Structures](#) on page 419

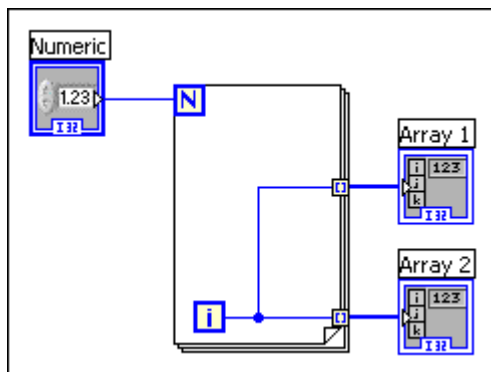
[Wiring Tunnels Automatically in Structures with Multiple Cases](#) on page 902

Overlapping Tunnels

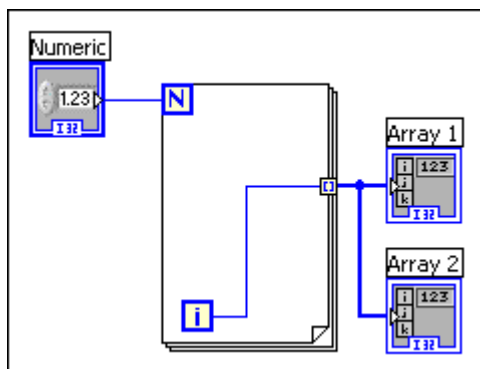
Because LabVIEW creates tunnels as you wire, tunnels sometimes overlap. Overlapping tunnels do not affect the execution of the block diagram, but they can make editing difficult because it can be hard to tell which tunnel belongs to which wire path.

If tunnels overlap, move one tunnel to expose the other. You also can remove all the wires from a tunnel to remove the tunnel and then rewire the *structure* correctly.

If you must wire from an object inside a structure to an object outside the structure and the object inside the structure already has a wire that passes data outside the structure, do not wire through the structure again, as shown in the following block diagram.



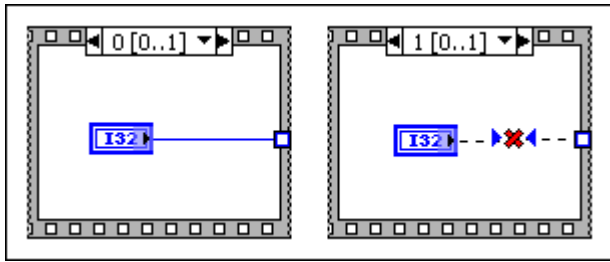
Instead, begin the second wire at the tunnel, shown as follows.



In a Case structure, overlapping tunnels might appear wired in each case.

Wiring from Multiple Frames of a Stacked Sequence Structure

This error occurs when two Stacked Sequence structure frames attempt to assign values to the same tunnel. The tunnel turns white to signal this error, as shown in the following block diagram.



Avoid wiring two sources to the same tunnel.

Wiring Tunnels Automatically in Structures with Multiple Cases

Complete the following steps to wire input and output tunnels automatically on any *structure that can contain multiple cases* except the Stacked Sequence structure.

1. After wiring the input and output tunnel in one case of the structure, right-click the output tunnel to display the **Linked Input Tunnel** shortcut menu.
2. Select one of the following wiring options from the **Linked Input Tunnel** shortcut menu.
 - **Create & Wire Unwired Cases** —LabVIEW automatically wires the tunnels in any existing cases that you have not wired previously. LabVIEW also wires the tunnels in any new cases you create.
 - **Create** —LabVIEW automatically wires the tunnels in any new cases you create. Existing cases remain unwired.

The tunnel selector cursor appears.



NOTE

LabVIEW disables these options if there are no tunnels to which you can link.

1. Click the input tunnel you want to associate with, or link to, the output tunnel. White triangle glyphs appear on the input and output tunnels to indicate they are linked.

If you selected **Create & Wire Unwired Cases**, notice that LabVIEW wires the tunnels in all existing cases. If you *add a subdiagram*, or case, to the structure, LabVIEW wires the tunnels in the new cases. If you selected **Create**, notice that LabVIEW wires the tunnels only in new cases you add. Existing cases remain unwired.

Right-click a linked output tunnel and select **Linked Input Tunnel»Find** to locate the corresponding input tunnel. Right-click a linked output tunnel and select **Linked Input Tunnel»Clear** to remove the association between the input and output tunnels. LabVIEW disables these options if the tunnel you select does not have a linked input tunnel.



Note If you right-click a linked output tunnel and select **Use Default If Unwired**, LabVIEW removes the association between the input and output tunnels.

Searching Options in LabVIEW

Finding a Block Diagram Terminal from a Front Panel Object

Complete the following steps to find a block diagram terminal from a front panel object.

1. Right-click the front panel control or indicator.
2. Select **Find Terminal** from the shortcut menu. LabVIEW highlights the block diagram terminal.

Finding a Front Panel Object from a Block Diagram Terminal

Complete the following steps to find a front panel object from a block diagram terminal.

1. Right-click the terminal on the block diagram.
2. Select **Find Control** or **Find Indicator** from the shortcut menu. LabVIEW highlights the front panel control or indicator.



TIP

You also can double-click the block diagram terminal to find the front panel object.

Finding All Instances of a VI or Type Definition

Complete the following steps to find all instances of a VI or type definition.

1. Right-click a VI or a type definition on the block diagram or in the VI Hierarchy window. You also can right-click the VI icon or connector pane in the upper right corner of the front panel or block diagram. (Statechart) Right-click the icon in the upper right corner of the Statechart Editor window.
2. Select **Find All Instances** from the shortcut menu. LabVIEW displays the results in the Search Results window.

You can replace some or all instances of the VI or type definitions. You also can find and replace block diagram objects or text.



RELATED INFORMATION

[Creating Custom Controls, Indicators, and Type Definitions](#) on page 165

[Finding and Replacing Block Diagram Objects](#) on page 907

[Finding and Replacing Block Diagram Objects](#) on page 907

[Finding and Replacing Text in VIs](#) on page 906

Finding an Object on the Palettes

Searching by Object Name

If you know the name of the palette object for which you are searching, use the Quick Drop dialog box to quickly add the object to the block diagram or front panel window. Use one of the following keyboard shortcuts to display the **Quick Drop** dialog box:

- (Windows) Press the <Ctrl-Space> keys.
- (macOS) Press the <Command-Shift-Space> keys.
- (Linux) Press the <Alt-Space> keys.
- On Chinese keyboards, press the <Ctrl-Shift-Space> keys.



TIP

You can speed up the initial launch of the **Quick Drop** dialog box by enabling the **Load palettes during launch** radio button on the Controls/Functions Palette page of the Options dialog box. Selecting to load the palettes while LabVIEW launches might slow down the launching of LabVIEW slightly.

Searching by Functionality

If you do not know the name of the palette object for which you are searching, you can browse the **Controls** or **Functions** palettes to locate the object.

Finding an Object or Terminal Local Variable

Complete the following steps to find an object or terminal local variable.

1. Right-click the control, indicator, or terminal.
2. Select **Find»Local Variables** from the shortcut menu. This appears only if you have multiple local variables.
3. Select the local variable from the Search Results window and click the **Go To** button. LabVIEW highlights the local variable.

Finding Property Nodes Linked to an Object or Terminal

Complete the following steps to find Property Nodes linked to an object or terminal.

1. Right-click the control, indicator, or terminal.
2. Select **Find»Property Nodes** from the shortcut menu. This appears only if you have multiple Property Nodes.
3. Select the Property Node from the Search Results window and click the **Go To** button. LabVIEW highlights the Property Node.

Finding VIs in a Directory

Select **Tools»Find VIs on Disk** to find a VI by filename in a directory. You can search for the exact filename, or you can use wildcards to search for part of the filename. You also can search for VIs inside LLB files.

Complete the following steps to find a VI in a directory and its subdirectories.

1. Select **Tools»Find VIs on Disk** to display the Find VIs on Disk window.
2. Click the browse button next to the **Directory to search** field, navigate to the directory in which you want to search for VIs, and click the **Current Folder** button.
3. Enter the text for which you want to search in the **Filenames to search for** text box. Use an asterisk wildcard * to match zero or more arbitrary characters.
4. Click the **Find** button. The **Search Results** listbox displays the results as LabVIEW finds them. Click the **Stop** button to cancel a search in progress.
5. Select one or more search results and click the **Open Selected VIs** button to open the VIs. You also can double-click a search result to open the VI. Press the <Ctrl> key and click additional search results to select more than one result. (macOS) <Command-Shift>-click. (Linux) <Alt-Shift>-click. Press the <Shift> key and click a search result above or below the current result to select all results in between the current result and the second result you selected.
6. Click the **Close** button in the **Find VIs on Disk** window to close the window.

You also can [search VIs in memory](#) for a particular object or string of text.

Finding and Replacing VIs, Objects, and Text

You can search a single VI or multiple VIs in an application instance for a particular object or string of text and replace that item with another. Select **Edit»Find and Replace** to display the Find dialog box. The **Find** dialog box displays different search options depending on whether you search for objects or text. Use the **Find** dialog box to search VIs in memory for VIs, Express VIs, functions, controls, indicators, type definitions, global variables, Property Nodes, breakpoints, or text. You can search the current VI, selected VIs in the application instance, or all VIs in the application instance.

(Statechart) You also can search a statechart diagram, including the guard and action code of the statechart, for a particular object or string of text.



NOTE

You also can search for a VI by filename in a directory.

You can replace all items that LabVIEW finds, or you can select the items you want to replace.



Note You cannot replace objects or text in running VIs.



RELATED INFORMATION

[Working with Application Instances](#) on page 382

[Finding and Replacing Block Diagram Objects](#) on page 907

[Finding and Replacing Text in VIs](#) on page 906

[Finding VIs in a Directory](#) on page 905

Finding and Replacing Text in VIs

You can [search VIs](#) for text in the front panel window, block diagram window, VI revision history, and VI and object descriptions and replace the existing text with another piece of text. You also can find and replace text in the configuration of a Call Library Function Node.

(Statechart) You can find text in the guard and action code of a statechart. You can find and replace text in the Statechart Editor window.

Complete the following steps to find and replace text in the VIs in an application instance.

1. Select **Edit»Find and Replace** to display the Find dialog box. You also can select a piece of text in the current VI and select **Edit»Find and Replace** to display the **Find** dialog box with the text preselected for the search.
2. Click the **Text** button.
3. From the **Application Instance** selection menu, select the [application instance](#) in which you want to search for the text. The default application instance is the application instance from which you opened the **Find** dialog box.
4. Enter the text you want to find in the **Type in the word(s) to search for** text box.
5. Click the **More Options** button to display the Text Search Options dialog box and configure where LabVIEW searches for the text.
6. Configure the scope of the search from the **Search Scope** pull-down menu.
7. Click the **Find** button. The Search Results window lists all instances of the text that LabVIEW found. If only one instance is found, LabVIEW highlights the result on the front panel window or block diagram and the Search Results window is not displayed.

8. Select an entry in the **Search Results** list and click the **Go To** button to find the text in the VI. You also can double-click the entry in the **Search Results** list to find the text in the VI.
9. In the **Replace with** list, enter the text with which you want to replace the original text or select a previous text entry.
10. To replace all instances of the original text with the new text, click the **Replace All** button. To replace only selected instances of the original text with the new text, select the instances you want to replace from the **Search Results** list and then click the **Replace Selected** button. After you replace a piece of text, you can select the corresponding entry in the **Search Results** list to find the new text in the VI.



NOTE

A strikethrough icon  appears next to an entry in the **Search Results** list if LabVIEW cannot replace that instance. For example, you cannot replace the text in a Property Node or an Invoke Node. You also cannot replace text in running VIs. If you click the **Replace All** or **Replace Selected** buttons, LabVIEW replaces all instances except those marked with a strikethrough icon.

Finding and Replacing Block Diagram Objects

You can [search VIs in an application instance](#) for VIs, Express VIs, functions, terminals, type definitions, global variables, control references, or breakpoints and replace them with other block diagram objects.

(Statechart) You also can find block diagram objects in the guard and action code of a statechart.

Complete the following steps to find and replace VIs or other block diagram objects in the VIs in an application instance.

1. Select **Edit»Find and Replace** to display the Find dialog box. You also can select an object in the current VI and select **Edit»Find and Replace** to display the **Find** dialog box with the object preselected for the search.
2. Click the **Objects** button.
3. From the **Application Instance** selection menu, select the [application instance](#) in which you want to search for the object. The default application instance is the application instance from which you opened the **Find** dialog box.
4. Click the **Select Object** button. From the menu that appears, select the object you want to find.
5. Configure the scope of the search from the **Search Scope** pull-down menu.
6. Click the **Find** button. The Search Results window lists all instances of the object that LabVIEW found. If only one instance is found, LabVIEW highlights the result on

the front panel window or block diagram and the Search Results window is not displayed.



NOTE

If you search for a local variable or a global variable, the **Search Results** list specifies whether the variable is set to read or write.

1. Select an entry in the **Search Results** list and click the **Go To** button to find the object on the block diagram. You also can double-click the entry in the **Search Results** list to find the object on the block diagram.
2. Click the **Replace with** button. From the menu that appears, select the object with which you want to replace the found objects.
3. To replace all instances of the original object with the new object, click the **Replace All** button. To replace only selected instances of the original object with the new object, select the instances you want to replace from the **Search Results** list and then click the **Replace Selected** button. LabVIEW does not replace objects if the new object is an incompatible replacement.

LabVIEW Misc

Selecting a Tool

If automatic tool selection is enabled and you move the cursor over front panel or block diagram objects, LabVIEW automatically selects the corresponding tool from the Tools palette. You can disable automatic tool selection and select a tool manually by clicking the tool you want on the **Tools** palette.

If you want to use the <Tab> key to cycle through the most commonly used tools in the sequence they appear on the palette, click the **Automatic Tool Selection** button on the **Tools** palette to disable automatic tool selection. Press the <Shift-Tab> keys or click the **Automatic Tool Selection** button to enable automatic tool selection again. You also can manually select a tool on the **Tools** palette to disable automatic tool selection. If automatic tool selection is disabled, you can press the spacebar to switch to the next most commonly used tool.



TIP

You can configure LabVIEW to disable automatic tool selection and move through the most commonly used tools whenever you press the <Tab> key and to toggle automatic tool selection only when you press the <Shift-Tab> keys or click the **Automatic Tool Selection** button. Select **Tools»Options** to display the Options dialog box, select Environment from the **Category** list, and remove the checkmark from the **Lock automatic tool selection** checkbox.

If automatic tool selection is enabled, you can select specific tools in the following ways:

- Press the <Shift> key to temporarily switch to the Positioning tool.

- Move the cursor over any open front panel or block diagram space and press the <Shift-Ctrl> keys to temporarily switch to the Scrolling tool. (macOS) Press the <Shift-Option> keys. (Linux) Press the <Shift-Alt> keys.

Navigating Large Front Panels and Block Diagrams

The **Navigation** window displays an overview of the active front panel in edit mode or the active block diagram. Use the **Navigation** window to navigate large front panels or block diagrams.



NOTE

LabVIEW does not have a zoom feature. Use the **Navigation** window to more easily navigate large front panels or block diagrams instead.

Complete the following steps to practice using the **Navigation** window.

1. Open the Exporting VIs project in the labview\examples\Application Control\VI Server\Exporting VIs directory.
2. Select **View»Navigation Window** to display the Navigation window or press the <Ctrl-Shift-N> keys. (macOS) Press the <Command-Shift-N> keys. (Linux) Press the <Alt-Shift-N> keys.
3. Complete the following steps to practice viewing and navigating a large front panel or block diagram using the **Navigation** window.
 1. Click an area of the image in the **Navigation** window to display that area in the front panel or block diagram window.
 2. Click and drag the image in the **Navigation** window to scroll through the front panel or block diagram. Portions of the front panel or block diagram that are not visible appear dimmed in the **Navigation** window.
 3. Resize the **Navigation** window to resize the image it displays.



NOTE

If the front panel contains multiple panes, the **Navigation** window displays an image of the front panel. However, you cannot click or drag the image in the **Navigation** window to navigate the front panel.

Changing Controls to Indicators and Indicators to Controls

Complete the following steps to change a control to an indicator or vice versa.

1. Right-click either the control or the indicator.
2. Select **Change to Indicator** or **Change to Control** from the shortcut menu.

Clearing Front Panel Indicators

Complete the following steps to clear all front panel indicators automatically each time you run a top-level VI or call a subVI.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Execution from the **Category** pull-down menu at the top of the **VI Properties** dialog box.
3. Place a checkmark in the **Clear indicators when called** checkbox.
4. Click the **OK** button.

Shortcut menu plug-ins

Understanding Shortcut Menu Plug-Ins

You can add custom items to the shortcut menu of front panel and block diagram objects by creating shortcut menu plug-ins. Shortcut menu plug-ins call VIs that you customize using VI scripting. You can create custom items that appear when you right-click the following objects:

- Edit-time front panel objects
- Edit-time block diagram objects
- Run-time block diagram objects

Refer to ni.com/lvmenus for more information about shortcut menu-plugins and for code examples from the NI Community.

Shortcut Menu Plug-In Architecture

A shortcut menu plug-in must contain the following files, where x is the name of the plug-in.

- `x .ctl` —The type definition that specifies the objects that the plug-in affects.
- `x .vi` —The build VI that builds the shortcut menu.
- `Execute x .vi` —The plug-in execution VI that executes when you select the custom item from the shortcut menu.

You save these files in an LLB with the plug-in name `x .llb`. LabVIEW searches and loads plug-ins from LLBs in one of the following directories only. LabVIEW ignores contents of subdirectories when searching for plug-ins.

- <LabVIEW Data>\PopupMenu\edit time panel and diagram—Contains LLBs of plug-ins that affect edit-time front panel and block diagram objects used by all versions of LabVIEW.
- <LabVIEW Data>\PopupMenu\run time diagram—Contains LLBs of plug-ins that affect run-time block diagram objects used by all versions of LabVIEW.
- labview\resource\plugins\PopupMenu\edit time panel and diagram—Contains LLBs of plug-ins that affect edit-time front panel and block diagram objects used by this version of LabVIEW only.
- labview\resource\plugins\PopupMenu\run time diagram—Contains LLBs of plug-ins that affect run-time block diagram objects used by this version of LabVIEW only.

If two LLBs have the same name, the LLB in the labview\resource\plugins\PopupMenu directory takes precedence.

Creating Shortcut Menu Plug-Ins

Creating a shortcut menu plug-in consists of the following parts:

- **Part 1**: Specifying the objects that the plug-in affects.
- **Part 2**: Determining how the custom item appears in the shortcut menu.
- **Part 3**: Editing how the plug-in executes when you select the custom item.

To begin creating a shortcut menu plug-in, run the following utility VI: labview\resource\plugins\PopupMenu\Create Shortcut Menu Plug-In From Template.vi

The utility VI generates the files you need for your plug-in. You customize the files to build the shortcut menu and execute the plug-in.

Part 1: Specifying the Objects that Shortcut Menu Plug-Ins Affect

Related Information

[Menus and Toolbars](#)

Part 1: Specifying the Objects that Shortcut Menu Plug-Ins Affect

You must specify which front panel or block diagram objects to apply a shortcut menu plug-in. Complete the following steps to specify the objects that the shortcut menu plug-in affects.



NOTE

Generate the files you need to create a shortcut menu plug-in by running the following utility VI: `labview\resource\plugins\PopupMenu\Create Shortcut Menu Plug-In From Template.vi`

1. Open the type definition `x.ct1`, where `x` is the name of the plug-in) that the utility VI generates.
2. Edit the **Affected Items** control, which contains the objects or refnum types that the shortcut menu plug-in affects. Initially, the **Affected Items** control contains an array of the `GObject` refnum type, which means the plug-in appears in the shortcut menu of any `GObject` refnum. You can edit the **Affected Items** control in the following ways:
 - Add an object by adding an array of a refnum type.
 - Remove an object by deleting an array.
 - Choose a different object by right-clicking an array and selecting **Select VI Server Class** and selecting a refnum type under the **Generic** hierarchy.

Note To *identify the refnum types* of an object, you can use the **Lookup All Shortcut Menu Tags and Types** plug-in that LabVIEW provides.

3. Select **File»Apply Changes** to apply changes and propagate the changes to the plug-in VIs that use the type definition.
4. Select **File»Save** to save changes to the type definition.



Understanding Shortcut Menu Plug-Ins



Part 2: Building the Shortcut Menu Using Plug-Ins

Part 2: Building the Shortcut Menu Using Plug-Ins

You can specify how and when your custom item appears in the shortcut menu of front panel or block diagram objects. You also can create plug-ins that build the shortcut menu in the following ways:

- Add custom items, conditionally, to object types.
- Toggle transaction behavior of plug-ins.
- Add multiple custom items.

- Remove items.

The following figure illustrates the block diagram of the build VI `x.vi`, where `x` is the name of the plug-in) that the Create Shortcut Menu Plug-In From Template VI generates. You modify the block diagram code to build the shortcut menu. Refer to the block diagram comments for tips on modifying the block diagram.

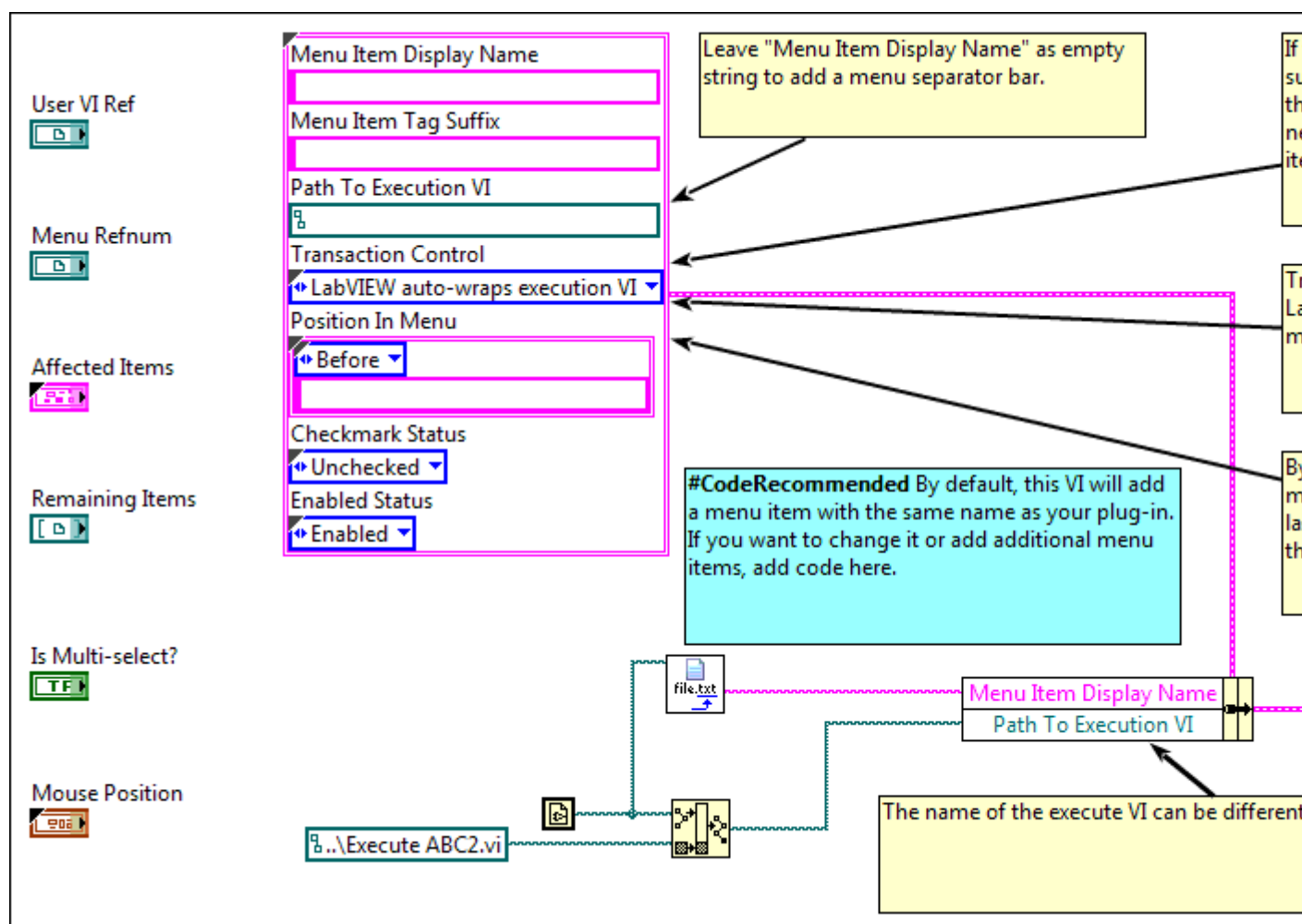


NOTE

Generate the files you need to create a shortcut menu plug-in by running the following utility VI: `labview\resource\plugins\PopupMenu\Create Shortcut Menu Plug-In From Template.vi`



Caution Do not modify the controls that are assigned to connector pane terminals. To load and execute plug-ins, LabVIEW requires these predefined control names, types, and terminal positions.



The default configuration of this VI builds the shortcut menu by adding your custom item before the **Properties** item. The display name of the custom item is the name of this VI without the file extension.

← *Part 1: Specifying the Objects that Shortcut Menu Plug-Ins Affect*

→ *Part 3: Executing Shortcut Menu Plug-Ins*

Part 3: Executing Shortcut Menu Plug-Ins

Use VI scripting to define actions that LabVIEW performs when you select a custom item from the shortcut menu.



NOTE

Generate the files you need to create a shortcut menu plug-in by running the following utility VI: `labview\resource\plugins\PopupMenu\Create Shortcut Menu Plug-In From Template.vi`

1. Open the plug-in execution VI `Execute x.vi`, where *x* is the name of the plug-in that the utility VI generates.
2. On the block diagram, use VI scripting to modify the behavior of objects the plug-in affects. Add a Property Node or Invoke Node to access and modify the properties and methods of objects using a VI Server reference.
3. (Optional) Add additional VIs to the LLB of the shortcut menu plug-in as needed to modify the behavior of the objects.

Note To avoid name conflicts, National Instruments recommends prefixing the filename of any additional VIs and subVIs with the name of the plug-in or using a project library. For example, name subVIs as `x_subVI.vi` or save additional VIs and subVIs in the project library `x.lvlib`, where *x* is the name of the plug-in.

4. Select **File»Save** to save changes. Restart LabVIEW for changes to take effect.

Tip To reload plug-ins without restarting LabVIEW, create an empty VI, add the Invoke Node to the block diagram, configure the Invoke Node for the `Menus:Refresh` method, and run the VI. This method is also useful if you write an installer for your plug-ins. You can run the method as part of the installation process. If you use the JKI VI Package Manager (VIPM) software to distribute your plug-ins, add the `Menus:Refresh` method to the post-installation VI.

Protecting Transactions When LabVIEW Executes Shortcut Menu Plug-Ins

To protect VI scripting operations when the plug-in execution VI runs, LabVIEW calls the `Set Busy VI` to disable the mouse. After the plug-in execution VI runs, LabVIEW calls the `Unset Busy VI` to enable the mouse.

You can instruct LabVIEW to automatically wrap your scripting operations in an undoable transaction by using the **Transaction Control?** of the build VI `x.vi`, where *x* is the name of the plug-in. Alternatively, you can instruct LabVIEW not to control transactions in the build VI so that you manage your own transactions in the plug-in execution VI. If LabVIEW automatically creates the transaction, configure the **Fail Transaction?** output of the plug-in execution VI to return `TRUE` if this VI does not run successfully.

Identifying Refnum Types and Menu Item Tags of Objects

You must specify which front panel or block diagram objects to apply a shortcut menu plug-in. In the type definition of the plug-in, you specify the refnum types of the objects. A single right-click on an object may generate references to more than one refnum type. Use the **Lookup All Shortcut Menu Tags And Types** plug-in to help you identify the affected refnum types. When you right-click an object and select the **Lookup All Shortcut Menu Tags And Types** plug-in, LabVIEW displays a list of refnum types affected by the right-click. Use the refnum types to specify the object your plug-in affects.

The **Lookup All Shortcut Menu Tags And Types** plug-in also lists the menu item tags for every shortcut menu item. Use the menu item tags to customize where to display your plug-in item in the shortcut menu.

To use the **Lookup All Shortcut Menu Tags And Types** plug-in, copy the following LLB:

```
labview\resource\plugins\PopupMenu\support\Lookup All
Shortcut Menu Tags And Types.llb
```

Paste the LLB into the following directory:

```
labview\resource\plugins\PopupMenu\edit time panel and
diagram
```

To remove the **Lookup All Shortcut Menu Tags And Types** plug-in from the shortcut menu, delete the `Lookup All Shortcut Menu Tags And Types.llb` and restart LabVIEW.

Related Information

[Understanding Shortcut Menu Plug-Ins](#)

[Specifying the Objects that Shortcut Menu Plug-Ins Affect](#)

[Building Shortcut Menus Using Plug-Ins](#)

Troubleshooting Shortcut Menu Plug-Ins

The following guidelines help to ensure your LabVIEW shortcut menu plug-in is valid. A malformed plug-in might not appear when you right-click an object or fail to execute if you select the plug-in. LabVIEW does not return errors or warnings if a plug-in is invalid.

- **VIs are not broken** —Load VIs in LabVIEW and ensure VIs do not report errors.
- **Connector panes adhere to required pattern** —The Create Shortcut Menu Plug-in From Template VI generates the build VI `x.vi` and plug-in execution VI `Execute x.vi` to adhere to a specific connector pane pattern and terminal name syntax. LabVIEW cannot use these VIs as plug-ins if you change the pattern or syntax. You should only change the type definition `x.ct1`, where `x` is the name of the plug-in).
- **Plug-in LLB and build VI names match** —LabVIEW saves the plug-in files and type definition in an LLB with the plug-in name `x.llb`. The build VI must have the same name as the LLB except for the file extension. This naming convention allows LabVIEW to simultaneously manage multiple plug-ins in memory.
- **Debugging tools are disabled** —You must disable debugging tools for the build VI. LabVIEW cannot stop for breakpoints or probes when building the shortcut menu.
- **Length of the menu item tag does not exceed 255 characters** —The maximum length of a menu item tag is 255 characters. LabVIEW constructs the tag name using the name of the plug-in and the menu item tag suffix, returned from the build VI. The complete tag name has the following syntax: `APP_SC_PLUGIN_<x.vi>/<menu_item_tag_suffix>`, where `x` is the name of the plug-in.
- **Refnum type matches the objects the plug-in affects** —Check that the refnum types in the type definition match the object type that the plug-in affects. To [identify the refnum types](#) of an object, you can use the **Lookup All Shortcut Menu Tags and Types** plug-in that LabVIEW provides.

Related Information

[Creating Shortcut Menu Plug-Ins](#)

[Identifying Refnum Types and Menu Item Tags of Objects](#)

[Tips and Tricks for Creating Shortcut Menu Plug-Ins](#)

Tips and Tricks for Creating Shortcut Menu Plug-Ins

The following list provides additional information about creating and using shortcut menu plug-ins.

- When you select and right-click multiple front panel or block diagram objects, plug-ins retrieve refnums for all the selected objects. Refnums pass to the **Affected Items** array of plug-ins when they match the **Affected Items** array. Refnums for all other objects pass to the **Remaining Items** array of plug-ins. This functionality allows you to enable plug-ins when multiple selected objects must match **Affected Items** or to enable plug-ins when only some selected objects match **Affected Items**.
- When you right-click without a selection list, you may still get multiple refnums. For example, right-clicking a terminal generates refnums for the terminal, its node, and the diagram of the node. As with the multiple selection case, you can determine how each plug-in decides to apply its menu items in these cases.

- Many plug-ins perform similar functionality. You may find subVIs useful to your plug-ins by browsing the plug-ins that LabVIEW provides. These plug-ins are in the `labview\resource\plugins\PopupMenu` directories.
- To share subVIs among multiple plug-ins, you can store the subVIs in a subdirectory that is in the same folder as your LLB.
- To add a menu separator bar to the shortcut menu, leave the Menu Item Display Name as an empty string in the build VI.
- To reload plug-ins without restarting LabVIEW, create an empty VI, add the Invoke Node to the block diagram, configure the Invoke Node for the `Menus:Refresh` method, and run the VI. This method is also useful if you write an installer for your plug-ins. You can run the method as part of the installation process. If you use the JKI VI Package Manager (VIPM) software to distribute your plug-ins, add the `Menus:Refresh` method to the post-installation VI.

Related Information

[Understanding Shortcut Menu Plug-Ins](#)

[Identifying Refnum Types and Menu Item Tags of Objects](#)

[Troubleshooting Shortcut Menu Plug-Ins](#)

Accessing LabVIEW Add-ons with the VI Package Manager (VIPM) Software

Menus:Refresh Method

Adding or Reducing Space on the Front Panel or Block Diagram

Complete the following steps to add space to the front panel or block diagram without resizing the window.

1. Press <Ctrl> and use the Positioning tool to click the front panel or block diagram. (macOS) Press <Option>. The Positioning tool cursor changes from an arrow to a plus sign.
2. While holding the key combination, drag in the direction you want to add space. The objects move in real time as you drag the mouse. If the direction you drag is primarily vertical or horizontal, the operation snaps to the dominant direction.
3. Release the mouse button to set the position of the objects.

To reduce the space between scattered objects, press <Ctrl-Alt> and drag in the direction you want to reduce the space. (macOS) Press <Option-Ctrl>.

Reading from One Front Panel Object and Writing to Two Block Diagram Locations

You can use a local variable to control multiple objects on the block diagram. If you have a VI with two While Loops, you can use a single front panel control, such as a button, to control the two parallel While Loops. These loops execute simultaneously.

Place the front panel control terminal inside of the first While Loop on the block diagram. Then, place a local variable inside the second While Loop. This procedure is similar to the procedure you use to write to a front panel object from two block diagram locations.



RELATED INFORMATION

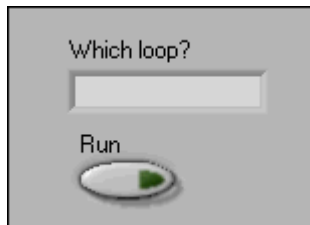
[Local Variables](#) on page 605

[While Loops: Repeating Operations Until a Condition Occurs](#) on page 414

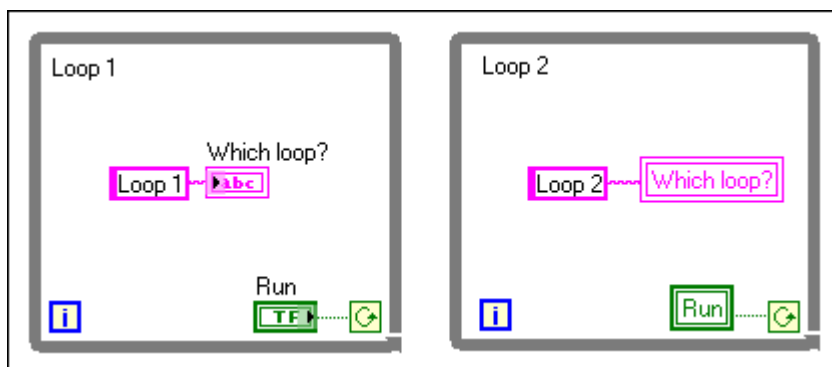
[Writing to One Front Panel Object from Two Block Diagram Locations](#) on page 918

Writing to One Front Panel Object from Two Block Diagram Locations

You can use a local variable to update a single front panel indicator from more than one location on the block diagram. For example, if you have a VI with two While Loops, you can update a front panel indicator to display which loop is currently executing. The following illustration shows the front panel for such a VI.



On the block diagram, place the indicator terminal in Loop 1 and a local variable instance of the indicator in Loop 2, as shown in the following illustration. Notice that this example also uses a local variable instance of the **Run** control to control the execution of both While Loops. You also can use a local variable to allow a single front panel button to control two parallel While Loops in a single VI. A wire does not connect the loops, and they execute simultaneously.

**NOTE**

You cannot associate a local variable with a Boolean control or indicator that has its *mechanical action* set to latch when pressed or released.

VIs

Dragging and Dropping VIs, Pictures, and Text

Click the Select a VI icon or text on the **Functions** palette, navigate to a VI, double-click the VI or drag the VI to the block diagram to create a subVI call to that VI. Use the same technique for *custom controls*, type definitions, and *global variables*. You also can drag text and pictures from the Windows Explorer and other applications to copy them to front panels and block diagrams.

**NOTE**

You cannot release the object until LabVIEW highlights the destination. You can press the <Esc> key before you release the mouse button to cancel a drag and drop operation.

You can perform the following tasks within LabVIEW using the drag and drop capabilities:

- Drag front panel controls and indicators to block diagrams to *create block diagram constants* and vice versa.
- Drag a subVI to a path control or constant to place the full pathname of the subVI in the control or constant.
- Drag a VI icon from the upper-right corner of a front panel or block diagram window to a block diagram to place that VI as a subVI. Press the <Shift> key while dragging to automatically wire the non-default values of the controls as constants for the subVI. If the subVI already appears on the block diagram, press the <Shift> key while dragging onto the existing subVI to update the wired constants. If a control has its default value, LabVIEW discards the constant wired to the subVI. Any inputs not wired to constants are unaffected.
- Drag and drop global variables and custom controls.
- Drag a VI icon into a VI refnum on a front panel control to load VIs into memory dynamically, which is part of the *VI Server* functionality.

- Drag an object near the boundary of a front panel or block diagram to scroll the window. Press the <Shift> key to increase the speed at which the window scrolls. If you release the mouse button immediately outside the boundary of the window while it scrolls, LabVIEW continues to scroll the window to the position where you released the mouse button and places the object in that position.

You can perform the following tasks between LabVIEW and another application using the drag and drop capabilities:

- Drag a file to a path control or constant to place the full pathname in the control or constant.
- Drag a datalog file to a datalog file refnum to create a cluster that contains the data structure in the datalog file.
- **Drag a graphics file** to a VI icon, picture ring control, front panel, or block diagram to place the picture in the control, on the front panel or block diagram, or on the VI icon. LabVIEW can import most graphic files that your computer supports.
- Drag a custom control file to the front panel to place the custom control on the front panel.
- (Windows) You can drag pictures and text from an application into the front panel or block diagram, provided the application supports ActiveX technology. For example, you can drag a bitmap or a section of text from a Microsoft Word document and add it to the front panel or block diagram.
- (macOS) Drag a text clipping or text selected from any application to place that text in a string control or constant, front panel, block diagram, or label.
- (macOS) Drag an image clipping or an image selected from another application to place it in a picture ring control, front panel, or block diagram.



Note (macOS) Your system must have the Drag Manager to drag and drop text and pictures from other applications. Systems 7.5 and later include the Drag Manager. It is available for System 7.0 up to System 7.5 as an extension from Apple Computer.

Refer to your operating system documentation for more information about drag and drop capabilities.

Customizing Styles for your Current VI

Setting the Control and Indicator Style for the Current VI

The Editor Options page of the VI Properties dialog box and the Front Panel page of the Options dialog box contain similar options. Use the **VI Properties** dialog box to set the style of controls and indicators in the current VI. Use the **Front Panel** page of the **Options** dialog box to set the style of controls and indicators in all new VIs.

Complete the following steps to change the style of the control or indicator LabVIEW creates when you right-click a terminal and select **Create»Control** or **Create»Indicator** from the shortcut menu in the current VI.

1. Select **File»VI Properties** to display the **VI Properties** dialog box.
2. Select **Editor Options** from the **Category** pull-down menu.
3. Select a style from the **Control Style for Create Control/Indicator** options.
4. Click the **OK** button.



NOTE

When you right-click a subVI terminal and select **Create»Control** or **Create»Indicator** from the shortcut menu, LabVIEW creates a control or indicator in the style of the subVI front panel object that corresponds to the subVI terminal. When you right-click a control or indicator and select **Create»Control** or **Create»Indicator** from the shortcut menu, LabVIEW creates a control or indicator in the style of the control or indicator you right-clicked.

Setting the Alignment Grid Size for the Current VI

Use the **VI Properties** dialog box to set the size of the alignment grid for the current VI. Use the grid options on the Front Panel page and the Block Diagram page of the **Options** dialog box to set the size of the grid for all new VIs.

Complete the following steps to set the size of the grid for the current VI.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Editor Options from the **Category** pull-down menu.
3. Enter the size to use for the front panel and block diagram grid in pixels in the **Front panel** and **Block diagram** fields, respectively.

Setting Printing Options for the Current VI

The **Print** dialog box, the **Print Options** page of the **VI Properties** dialog box, and the **Printing** page of the **Options** dialog box contain similar options. Use the **Print** dialog box and the **VI Properties** dialog box to set printing options for the current VI. Use the Printing page of the **Options** dialog box to set printing options for all VIs.

Complete the following steps to set printing options for the current VI.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Print Options from the **Category** pull-down menu.
3. Select the appropriate printing options.
4. Click the **OK** button.

When you print VI documentation using the Print dialog box, LabVIEW uses the options in the **Print** dialog box to set the printing options for the current VI. For example, if you place a checkmark in the **Surround panel with border** checkbox on the **Printer** page of the **Print** dialog box and you print the VI documentation, a checkmark also appears in the **Surround front panel with border** checkbox on the **Print Options** page of the **VI Properties** dialog box.

If the operating system has a print dialog box, certain options you set in that dialog box also set printing options for the current VI. For example, if you change the page orientation (portrait or landscape) in the operating system print dialog box, LabVIEW also sets the page orientation for the VI.

Setting Revision History Options for the Current VI

The Revision History page of the VI Properties dialog box and the Revision History page of the Options dialog box contain similar options. Use the **VI Properties** dialog box to set options for the current VI. Use the Revision History page of the **Options** dialog box to set options for all new VIs.

Complete the following steps to revise the information that appears in the History window for the current VI.

1. Select **File»VI Properties** to display the **VI Properties** dialog box.
2. Select **Revision History** from the **Category** pull-down menu.
3. Remove the checkmark in the **Use the default history settings from the Options dialog box** checkbox to override the options on the **Revision History** page of the **Options** dialog box.
4. Select the appropriate history options.

You also can use the History properties to customize the VI history programmatically.

Creating a Recursive VI

You can configure a VI to be *recursive*, meaning the VI can call itself from its own block diagram or from the block diagram of subVIs. Recursive VIs are useful if you want to operate multiple times on the output of the same process.

Complete the following steps to make a VI recursive.

1. Open the VI you want to make recursive.
2. Select **Shared clone reentrant execution** on the Execution page of the VI Properties dialog box.

3. Call the recursive VI from its own block diagram or from the block diagram of a subVI.

**NOTE**

LabVIEW allows up to 15,000 recursive calls on 32-bit platforms and up to 35,000 recursive calls on 64-bit platforms. To avoid too many recursive calls, consider calling the recursive VI in a Case structure.

Searching in the VI Hierarchy

Complete the following steps to find a hierarchy node by name in the VI Hierarchy window.

1. Open a VI or project and select **View»VI Hierarchy** to display the **VI Hierarchy** window.
2. Initiate a search by typing the name of the item you want to find anywhere in the window. As you type the text, the search string appears, displaying the text as you type. LabVIEW highlights one item at a time whose name begins with the search string.
3. If there is more than one item with a name that begins with the search string, press the <Enter> key to search for the next item that matches the search string. Press the <Shift-Enter> keys to find the previous item that matches the search string.

You also can find an item in the **VI Hierarchy** window by *selecting Edit»Find* to display the Find dialog box. From the **Search Scope** pull-down menu, select **VI Hierarchy** to search only the **VI Hierarchy** window.

**NOTE**

You can perform the same search in the LabVIEW Class Hierarchy window. Select **View»LabVIEW Class Hierarchy** to display the window.

Viewing the Hierarchy of VIs

The VI Hierarchy window displays a graphical representation of all open LabVIEW projects and targets, as well as the calling hierarchy for all VIs in memory. Use this window to view the subVIs and other nodes that make up the VIs in memory and to search the VI hierarchy.

The **VI Hierarchy** window displays the following items:

- Projects and targets
- Static and dynamic subVI calls
- Type definitions
- Global variables
- Shared variables
- LabVIEW classes

- XControls
- Statecharts
- Project libraries `.lvlib`
- Express VIs
- Static VI references

The **VI Hierarchy** window displays a top-level icon to represent the main LabVIEW *application instance*, under which appear all the objects that are not part of a project or are not part of the application instance for a project. If you add a project, the **VI Hierarchy** window displays another top-level icon to represent the project. Each target you add appears under the project.

Complete the following steps to use the **VI Hierarchy** window.

1. Select **View»VI Hierarchy** to display the **VI Hierarchy** window.
2. Select from the following options:
 - Move the cursor over nodes in the **VI Hierarchy** window to display the name of each node in a tip strip.
 - Use the Positioning tool to select a VI or subVI node from the **VI Hierarchy** window and drag it to the block diagram. You also can select and copy one or more nodes to the clipboard and paste them on block diagrams.
 - Double-click a VI or subVI node to display the front panel of that VI.
 - Hold the <Ctrl> key and double-click a VI or subVI node to display the front panel and block diagram of that VI.
 - *Search for a node* by name.
 - Right-click a blank area of the window and select **Show All VIs** from the shortcut menu to show the entire hierarchy.
 - Right-click a node and select **Highlight Connections** from the shortcut menu to highlight other items in the hierarchy that connect with the current node.
 - Right-click a project or a VI and select **Hide All SubVIs** or **Show All SubVIs** from the shortcut menu to alternately hide and show the subVIs in the hierarchy.
 - Right-click a project or a VI and select **Show Immediate SubVIs** from the shortcut menu to show the subVIs that immediately connect with a single node. This item is available only if all subVIs are hidden.
 - Right-click a node and select **Show VI Hierarchy** from the shortcut menu to display the hierarchy for a single node.
 - Right-click a node and select **Show All Callers** from the shortcut menu to highlight all callers of the node you select.
 - Right-click a paused VI and select **Show Call Chain** from the shortcut menu to highlight the call chain of the VI. The call chain is the chain of callers from the current VI to the top-level VI. This option is only available while debugging a running VI.

- Use the buttons on the **VI Hierarchy** window toolbar to select more ways to view the VI hierarchy.



NOTE

The **VI Hierarchy** window is a dynamic view of what is in memory. If you rearrange items in the hierarchy, LabVIEW does not save the new position of items. When you close and reopen the **VI Hierarchy** window, the positions regenerate.

To display instances of polymorphic VIs in the VI hierarchy, click the red arrow at the bottom of each polymorphic VI icon. The top-level VI indicates that it is linked to the polymorphic VI and to an instance of the polymorphic VI.

To switch to the Scrolling tool when *automatic tool selection is enabled*, press the <Ctrl-Shift> keys when the cursor is not over an icon in the window. (macOS) Press the <Option-Shift> keys. (Linux) Press the <Alt-Shift> keys.

You also can use the Application:All VIs in Memory property to retrieve a list of VIs in memory programmatically.

Polymorphic VIs

Building Polymorphic VIs

Polymorphic VIs adapt to different data types. A polymorphic VI is a collection of VIs with the same connector pane patterns. Each VI in the collection is an instance of the polymorphic VI, and each instance has at least one input or output terminal that accepts a different data type than in the other instances.

For example, the Read Key VI is polymorphic. Its **default value** terminal accepts Boolean; double-precision, floating-point numeric; 32-bit signed integer numeric; path; string; or 32-bit unsigned integer numeric data.

Build polymorphic VIs when you perform the same operation on different data types and the operation must be implemented differently across data types. If the operation can be implemented in similar ways for different data types without significant customization for a specific data type, *build malleable VIs* instead.

Complete the following steps to build a polymorphic VI.

1. Create two or more VIs with the same *connector pane pattern*. Make sure the connector panes have corresponding input and output terminals—if a terminal on the connector pane of one VI is an input, the corresponding terminal of the connector pane of the other VI(s) also must be an input or it must be unused. The same applies for output terminals. The VIs do not have to have similar data types or similar subVIs and functions.
2. Select **File»New** to display the New dialog box.
3. From the **Create New** list, select **Polymorphic VI**.

4. Click the **OK** button to display the Polymorphic VI window.
5. Click the **Add** button to display a file dialog box.
6. Select one of the VIs you created to include as an instance in the polymorphic VI, and click the **OK** button. The VI appears at the top of the **Instance VIs** list. The instance at the top of the list is the default instance of the polymorphic VI and determines the default connector pane for the polymorphic VI.
7. Repeat steps 5 and 6 to include other VIs. Any VIs that do not conform to the default connector pane pattern appear with a broken icon . You still can save the polymorphic VI. To remove an instance from the polymorphic VI, select the instance and click the **Remove** button.
8. (Optional) To change the order of the instances, select an instance in the list and click the **Move Up** arrow button or the **Move Down** arrow button. You also can select an instance in the list and drag it to a new position in the list.



NOTE

Double-clicking a VI in the **Instance VIs** list opens the VI.

1. (Optional) To specify how the polymorphic VI appears on the block diagram, configure the **Block Diagram Appearance** options.
2. Click the **Edit Icon** button to [create an icon](#) for the polymorphic VI.
3. [Create a description](#) for the polymorphic VI by selecting **File»VI Properties** and selecting **Documentation** from the **Category** pull-down menu. The polymorphic VI description is not associated with descriptions of the instance VIs that compose the polymorphic VI. If you select **Draw Instance VI Icon** in the **Polymorphic VI** window, create descriptions for each instance VI because LabVIEW displays the connector pane and description of the instance VI in the **Context Help** window when you move the cursor over the polymorphic VI. The **Context Help** window also displays a description of each instance as you move the cursor through the shortcut menus of the polymorphic VI or its selector.
4. Select **File»Save** to save the VI.
5. Select **File»Close** to close the **Polymorphic VI** window.

You also can [add or remove an instance](#) in an existing polymorphic VI, [edit the shortcut menus](#) of a polymorphic VI and its selector, and [manually select the instance](#) of a polymorphic VI.

Related Information

[Polymorphic VIs](#)

[Choosing between Malleable VIs and Polymorphic VIs](#)

Editing the Shortcut Menus of a Polymorphic VI

Complete the following steps to edit the instance names to display in the shortcut menus of a polymorphic VI you created and its selector.



NOTE

Use caution when editing polymorphic *VIs that ship with LabVIEW*.

1. Right-click a polymorphic VI on the block diagram and select **Open Polymorphic VI** from the shortcut menu to display the Polymorphic VI window. You also can select **File»Open** and open a polymorphic VI to display the window.
2. Select the instance whose name you want to edit in the **Instance VIs** list.
3. Click the **Edit Name** button to display the Edit Instance Name dialog box.
4. In the **Menu Name** text box, enter the name to use in the polymorphic VI shortcut menu. To create a hierarchical shortcut menu, use a colon : to separate menu items.
5. In the **Polymorphic VI Selector Name** text box, enter the name to use in the *polymorphic VI selector* shortcut menu. To create a multiline selector name, press the <Enter> key on the keyboard to start a new line.
6. Click the **OK** button to return to the Polymorphic VI window. The shortcut menu names appear next to the instance VI name in the **Instance VIs** list.
7. Select **File»Save** to save the VI.
8. Select **File»Close** to close the Polymorphic VI window.

Adding or Removing an Instance in a Polymorphic VI

Complete the following steps to add an instance to a polymorphic VI you created.



NOTE

Use caution when editing polymorphic *VIs that ship with LabVIEW*.

1. Right-click a polymorphic VI on the block diagram and select **Open Polymorphic VI** from the shortcut menu to display the Polymorphic VI window. You also can select **File»Open** and open a polymorphic VI to display the window.
2. Click the **Add** button to display a file dialog box.
3. Select a VI to include as an instance in the polymorphic VI and click the **OK** button. The VI you add must have the same *connector pane pattern* as all other instances of the polymorphic VI.
4. (Optional) To change the order of the instances, select an instance in the list and click the up arrow or down arrow button. You also can select an instance in the list and drag it to a new position in the list.
5. Select **File»Save** to save the VI.
6. Select **File»Close** to close the Polymorphic VI window.

Complete the following steps to remove an instance from a polymorphic VI you created.

1. Right-click a polymorphic VI on the block diagram and select **Open Polymorphic VI** from the shortcut menu to display the Polymorphic VI window. You also can select **File»Open** and open a polymorphic VI to display the window.
2. Select the instance you want to remove from the **Instance VIs** list.
3. Click the **Remove** button.
4. Select **File»Save** to save the VI.
5. Select **File»Close** to close the Polymorphic VI window.

Building Malleable VIs

Build *malleable VIs* to perform the same operation on any acceptable data type instead of saving a separate copy of the VI for each data type. Malleable VIs are useful if an operation can be implemented in similar ways for different data types without significant customization for a specific data type. If the operation must be implemented differently across data types or if the sets of terminals differ across data types, *build a polymorphic VI* instead.

You can create a malleable VI in the following ways:

- To create a new, blank malleable VI, select **File»New** and select **Malleable VI** from the **New** dialog box.
- To convert an existing VI into a malleable VI, set the file extension to `.vim`. If the existing VI is not configured to be *inlined* into its calling VIs, the malleable VI appears broken. You must configure the malleable VI to be inlined by selecting **File»VI Properties»Execution**, enabling the **Inline subVI into calling VIs** and **Preallocated clone reentrant execution** options, and disabling the **Allow debugging** and **Enable automatic error handling** options.



NOTE

You can convert only standard VIs into malleable VIs. You cannot convert polymorphic VIs, global VIs, or XControl abilities into malleable VIs.

When you place a malleable VI on the block diagram, the inputs and outputs display the default data types. If you wire a control, indicator, or constant to an input or output, the input or output adapts to the wired data type.

For example, in the Sort 2D Array malleable VI, the **2D array** input and the **sorted 2D array** output accept 2D arrays of any data type except refnums, with 2D array of variant as the default data type. If you wire a 2D array of string to **2D array**, the data types of **2D array** and **sorted 2D array** both change to 2D array of string. However, if you wire a 1D array to **2D array**, the VI appears broken because this malleable VI is not implemented to accept 1D arrays.

You can build specialized malleable VIs for the following purposes using the Type Specialization structure and the Assert Type VIs and functions:

- Customizing sections of code in a malleable VI for specific data types

- Forcing a malleable VI to accept only a subset of the acceptable data types
- Forcing a malleable VI to decline specific data types

For example, the Sort 2D Array malleable VI uses the Type Specialization structure to behave differently when **index** is an integer versus an array of integers so that the malleable VI works in an expected way for both data types. The **index** input coerces integer types other than signed 32-bit integers to signed 32-bit integers. You can force **index** to accept signed 32-bit integers only using the Assert Integer Type VI.

Because malleable VIs are inherently polymorphic, they accept any data type that does not cause syntax errors. NI recommends that you allow coercion in your malleable VIs for better code reusability. Restrict the data types a malleable VI can accept only if your application meets one of the following conditions:

- Coercion causes significant performance burden.
- Specific data types in the context of the application are meaningless.

Related Information

Debugging Malleable VIs

Choosing between Malleable VIs and Polymorphic VIs

Debugging Malleable VIs

After you place a *malleable VI* on the block diagram and wire specific data types to the inputs and outputs, the malleable VI adapts to the wired data types. You cannot debug the instance VI with the wired data types directly because the instance VI is inlined and debugging is disabled. However, you can debug this instance VI by converting it into a standard VI.

Right-click the instance VI and select **Convert Instance VI to Standard VI** to convert the instance VI into a standard VI. The standard VI then replaces the instance VI on the block diagram and the front panel of the standard VI opens. The standard VI has debugging disabled initially, but you can enable debugging on it by selecting **File»VI Properties»Execution**, disabling the **Inline subVI into calling VIs** option, and enabling the **Allow debugging** option and one of the reentrant execution options.

Saving VIs

You can save VIs as individual files or you can group several VIs together and save them in an LLB. LLB files end with the extension `.llb`. National Instruments recommends that you

save VIs as individual files, organized in directories, especially if multiple developers are working on the same project.



NOTE

Make backup copies of VIs before you upgrade to a new version of LabVIEW or when you prepare to make major revisions to the VIs.

You also can use the `Save:Instrument` method to save a VI programmatically.



RELATED INFORMATION

[Saving Individual VI Files](#) on page 941

[Saving VIs in Existing LLBs](#) on page 942

Creating Password-Protected VIs

When you distribute VIs, you can assign password protection to the block diagrams. The block diagram is still available, but users must enter a password to view or edit the block diagram.



CAUTION

VI password protection does not encrypt block diagrams. For maximum security, [remove the block diagrams](#) from your VIs. Refer to the KnowledgeBase at ni.com for more information about the security differences between password protection and removing block diagrams.

You also can [lock a VI](#) so users can see but not edit a block diagram. However, users can unlock a locked VI that is not password-protected.

Complete the following steps to set password protection for a VI.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Protection from the **Category** pull-down menu in the **VI Properties** dialog box.
3. Select the **Password-protected** option to display the **Enter Password** dialog box.
4. Enter the password in the **Enter New Password** field. Re-enter the same password in the **Enter New Password Again to Verify** field.
5. Click the **OK** button.



NOTE

Applying a password for a VI automatically stores the password in the LabVIEW password cache, which gives you access to make changes to the VI as if you had entered the password. You can remove the password from the cache by restarting LabVIEW or by clearing the password cache from the Environment page of the Options dialog box.

You also can add a password using the Source Distribution Properties dialog box.

You also can use the Lock State:Set method to set the password of a VI programmatically.

Locking and Unlocking VIs

You can lock a VI so users can see but not edit the front panel or block diagram. You also can *assign password protection to a VI* so users cannot see or edit the block diagram without a password. You can use the Lock State:Set method to lock or unlock a VI programmatically.

Locking VIs

Complete the following steps to lock a VI.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Protection from the **Category** pull-down menu.
3. Select **Locked (no password)**.
4. Click the **OK** button.

Unlocking VIs

Complete the following steps to unlock a VI.

1. Select **File»VI Properties** to display the **VI Properties** dialog box.
2. Select **Protection** from the **Category** pull-down menu.
3. If the VI is locked without a password, select **Unlocked (no password)** to unlock it. If the VI is locked with a password, select **Unlocked (no password)** and enter the appropriate password in the **Authentication** dialog box. Click the **Yes** button in the prompt to remove protection.

You also can select **Operate»Change to Edit Mode** to unlock a VI. Select **Unlock** in the file dialog that appears.

Common Mass Compile Status Messages

You may receive one or more of the following common mass compile messages under the **Status** tab of the Mass Compile dialog box.



NOTE

If you receive a message not listed in the following table, open the VI listed. If the VI has an error, fix the error in the VI before trying to mass compile again.

Message	Description
### Bad VI : " filename of VI " Path =" path to VI "	The specified VI is a non-executable VI that LabVIEW cannot mass compile. Open the VI and check for errors.
Failed to load path to VI	LabVIEW cannot find the VI in the specified directory or search directories because the VI is not in memory. Find the specified VI and make sure it is not corrupt or does not have errors.
Search failed to find " filename of VI " previously from " path to VI "	LabVIEW cannot find the VI in the specified directory or search directories at the specified location. Find the specified item and move it into either a search directory or the expected location.
Could not load path to VI because path to VI in memory	LabVIEW found the VI in the specified directory but cannot load the VI because a VI of the same name is already loaded in memory. Compare the two items, and confirm that the correct version is in the calling VI.
The filename of VI expected to be at " path to VI " was loaded from " path to VI "	The VI was not found at the path specified in the calling VI, so LabVIEW loaded an item with the same name from the search directories. Check that the VI specifies the caller and the directory paths correctly and check that the loaded VI is the same as the caller specified VI. If the VIs are not the same, add the correct subVI to the calling VI.
The project at path to project could not be compiled because it is already open. Close the project, then try compiling again.	LabVIEW cannot compile the project because it is already open. Close the project before you compile.
Skipping project already in memory : path to project	LabVIEW found a VI with the same path that has already been loaded in memory. As a result, LabVIEW is not mass-compiling the VI.
CompileFile: error number at path to VI	This error returns a specified error code that you can search for more information.
CompileFile: skipping path to VI	LabVIEW cannot not compile the VI for one of a few reasons. The VI could be a template VI or have template controls. The file at the path could have a file type different from what the extension of the path suggests. The VI also might not compile because LabVIEW cannot find or open the VI or the VI is corrupt.
CompileFile: user cancelled at path to VI	The VI did not compile because you cancelled loading the VI.

LLBs

Creating LLBs

Complete the following steps to create a new [LLB](#).

1. Select **File»Save** or **Save As** from the block diagram or front panel of a VI to display a file dialog box. (macOS) If you use native file dialog boxes, click the **Use LLBs** button in the file dialog box.



NOTE

You can create a new LLB with a VI you already saved by selecting **Tools»LLB Manager** from the block diagram or front panel to display the LLB Manager window.

1. Click the **New LLB** button to display the New LLB dialog box.
2. Enter the name of the new LLB and click the **Create** button.
3. In the file dialog box that appears, enter the name of the VI to save it in the new LLB. LabVIEW automatically adds a `.llb` extension to the name of the LLB.
4. Click the **OK** button.

You also can set the destination for the source distribution to output to an LLB on the Destinations page of the Source Distribution Properties dialog box.

You also can [save VIs in existing LLBs](#).

Converting Directories to LLBs

Convert directories to [LLBs](#) when you want to [group all the VIs](#) in a directory in an LLB.

Complete the following steps to convert directories to LLBs.

1. Select **Tools»LLB Manager** to display the LLB Manager window.
2. Navigate to the directory where the VIs reside by clicking the down arrow in the pull-down menu. LLBs, other files, and subdirectories appear in the list below the pull-down menu.
3. Select the directory where the VIs reside.
4. Click the **Convert Directories and LLBs** button to display the Directories to Convert dialog box.
5. Click the **Convert** button.
6. Select **File»Close** to exit the **LLB Manager** windows.

You also can [convert LLBs to directories](#).

Converting LLBs to Directories

Complete the following steps to convert [LLBs](#) to directories when you want to remove VIs from an LLB and have them appear as separate VIs.

1. Select **Tools»LLB Manager** to display the LLB Manager window.
2. Navigate to the directory where the LLB resides by clicking the down arrow in the pull-down menu. LLBs, other files, and subdirectories appear in the list below the pull-down menu.
3. Select the LLB to convert.
4. Click the **Convert Directories and LLBs** button to display the Conversion Options dialog box.
5. (Optional) Configure the **Directory name** and **Original LLB** options.
6. Click the **Convert** button.
7. Select **File»Close** to exit the **LLB Manager** window.

You also can [convert directories to LLBs](#).

Merging VIs and LLBs

You can use source control to share software projects between multiple developers. If you check out a file, make changes, and check in the updated file, the updated file replaces the master copy and becomes available to the other developers. However, if you check out a file that is currently checked out by another developer, any change you make to the file will have to be merged with the changes the other developer makes before an updated master copy is saved.

Use the Select LLBs to Merge dialog box, available in the LabVIEW Professional Development System, to automatically or interactively resolve differences between an LLB and two revisions of that original LLB.

You also can merge individual VIs using the Select VIs to Merge dialog box, also available in the LabVIEW Professional Development System.



RELATED INFORMATION

[Using Source Control in LabVIEW](#) on page 384

[Merging LLBs](#) on page 972

[Merging VIs](#) on page 971

Marking VIs as Top-Level VIs in an LLB

Mark a VI as a top-level VI in an **LLB** so when you launch LabVIEW with an LLB name *specified on a command line*, LabVIEW opens all the top-level VIs in that LLB. (macOS and Linux) Mark a VI as top level so when you open the LLB from the operating system, LabVIEW opens all the top-level VIs in that LLB.

Top-level VIs are separated by a dividing line from all other VIs in the LLB when you view the LLB contents in the File dialog box. The top-level VIs appear above the dividing line.

Complete the following steps to mark a VI as a top-level VI.

1. Select **Tools»LLB Manager** to display the LLB Manager window.
2. Double-click an LLB to display the VIs in the LLB.
3. Select a VI and select **Edit»Top Level**. You also can right-click a VI and select **Top Level** from the shortcut menu.
4. Select **File»Close** to exit the **LLB Manager** window.

(Windows) You also can double-click a .llb file in Windows Explorer to display the **LLB Manager** window. Right-click a VI in the LLB and select **Top Level** from the shortcut menu to mark the VI as top level.

Mass Compiling VIs

When you mass compile VIs, LabVIEW verifies the existence of any subVIs and relinks the subVIs to the main VI. Relinking subVIs to the main VI decreases loading time because LabVIEW does not have to search for those VIs. LabVIEW also updates VIs to the version of LabVIEW you use. LabVIEW detects and reports corrupted VIs, which LabVIEW cannot load. If LabVIEW locates corrupted VIs, you can restore them from a backup.

Certain circumstances, such as upgrading a VI to a newer version of LabVIEW, require LabVIEW to recompile a VI when the VI first loads. LabVIEW continues to recompile the VI each time the VI loads until you save the VI. If you expect LabVIEW to recompile many VIs on load, you might want to compile and save all the VIs at the same time to avoid these tasks at a later time. When you simultaneously compile and save a collection of VIs, you are mass compiling the VIs.

Complete the following steps to mass compile a directory of VIs or an LLB.

1. Select **Tools»Advanced»Mass Compile** to display the Mass Compile dialog box.
2. Navigate to the directory of VIs or LLB you want to compile.
 - Navigate into a folder and click the **Current Folder** button to update all the VIs and LLBs in that folder.
 - Select a .llb file to update all the VIs in that LLB. Select the open folder icon labeled VI with a single period to the right and click the **Select** button.

3. (Optional) Place a checkmark in the **Log Results** checkbox and select the desired location to store a log of the mass compile operation.
4. (Optional) Place a checkmark in the **Cache VIs** checkbox and enter the number of VIs you want LabVIEW to store in memory.

**NOTE**

Caching VIs can significantly decrease the execution time of a mass compile because LabVIEW does not have to load cached VIs when you compile a top-level VI. However, VI caching consumes more system memory. If you mass compile a folder that contains VIs with the same name in different subfolders and the VIs do not belong to a library, VI caching may cause [cross-linking](#).

5. Click the **Mass Compile** button to mass compile the VIs.

Note If you [receive messages](#) under the **Status** tab, open the VI listed. If the VI has an error, fix the error in the VI before trying to mass compile again.

6. Click the **Done** button to return to LabVIEW.

When you mass compile VIs, the **Status** tab of the Mass Compile dialog box displays a log of [warning messages](#) for the mass compile operation. For example, the log can report corrupted VIs and VIs that were loaded from the search directories instead of the path specified in the calling VI. You can mass compile to detect unexpected behavior in VIs you compile, such as VIs that break or load from unexpected locations.

You also can use the Mass Compile method or the [MassCompile](#) command line operation to load and compile VIs in a directory programmatically.

Mass Compile Operation Order

LabVIEW mass compiles only VIs last saved in a previous version of LabVIEW and VIs with unsaved changes that would result in a recompile when you open them. When LabVIEW mass compiles a directory of VIs, it processes files in the following order at each level of the directory hierarchy:

- LabVIEW project files—When LabVIEW encounters a project, LabVIEW processes the project file, then all files owned by the project in the order they appear in the project tree of the **Project Explorer** window. Project- and target-specific settings defined by the project such as [conditional disable symbols](#) can affect how the files are compiled, such as by specifying the platforms or processor LabVIEW uses to execute the VI.
- Files within subdirectories—LabVIEW processes subdirectories one by one in alphabetical order. For example, if a directory contains folders B and F, LabVIEW processes the files in B in alphabetical order, then the files in F in alphabetical order.
- Any remaining files at that level of the directory, in alphabetical order

Removing Block Diagrams from VIs

You can remove the block diagram source code from a VI so other users cannot edit the VI if you distribute VIs to other computers or other users. You also can [assign password protection to a block diagram](#) so the block diagram is still available if users enter a password, but a password provides less security than removing the block diagram.



NOTE

If you save a VI without the block diagram, users cannot edit the VI, [move the VI to another platform](#), or upgrade the VI to a future version of LabVIEW.

Complete the following steps to save VIs without the block diagrams to reduce the file size and prevent users from changing the source code.

1. [Create a source distribution](#).
2. On the Source File Settings page of the Source Distribution Properties dialog box, remove the checkmark from the **Use default save settings** checkbox and place a checkmark in the **Remove block diagram** checkbox to ensure that LabVIEW removes the block diagram.

Note You cannot remove a block diagram from a [VI without compiled code](#). If you place a checkmark in the **Remove compiled code** checkbox on the Additional Exclusions page, LabVIEW dims the **Use default save settings** checkbox on the **Source File Settings** page.

3. Build the source distribution to create a copy of the VI without its block diagram.



Caution If you save VIs without block diagrams, do not overwrite the original versions of the VIs. Save the VIs in different directories or use different names.

Removing Files from LLBs

Complete the following steps to delete a file from an [LLB](#).



NOTE

Deleting a file from a .llb file in Windows Explorer permanently deletes the file and does not store the file in the Recycle Bin.

1. (Optional) Save a backup copy of the file to another location.
2. Select **Tools»LLB Manager** to display the LLB Manager window.
3. Navigate to the LLB you want to modify and double-click the LLB to view files in the LLB.
4. Select the file you want to remove from the LLB.
5. Click the **Delete** button or select **File»Delete** to remove the file from the LLB. You also can right-click the file and select **Delete** from the shortcut menu to remove the file.



Note You cannot cancel changes you make in the **LLB Manager** window.

1. Select **File»Close** to exit the **LLB Manager** window.

(Windows) You also can double-click a .llb file in Windows Explorer to display the **LLB Manager** window and delete files in the LLB. Right-click a file in the LLB and select **Delete** from the shortcut menu to delete the file.

Reverting to the Last Saved Version of a VI

Select **File»Revert** to return to the last saved version of the VI you are editing. A dialog box confirms whether to discard all changes made to the VI.

VIs modified since they were last saved have an asterisk in their title bars, the list of open VIs displayed in the **Window** menu, and the All Windows dialog box. When you save a VI, the asterisk disappears until you make a new change.

You also can use the Revert VI method to return to the last saved version of the VI programmatically.

Saving Changes to VIs That Ship with LabVIEW

The following list describes some of the caveats and recommendations when modifying VIs and other files that ship with LabVIEW:

- Make a copy of the shipping file and save it in another directory before making changes.
- Do **not** save your own VIs or other files in .llb files installed with LabVIEW in the labview\vi.lib directory because LabVIEW uninstalls the .llb files, including any VIs and other files you saved in the .llb files, when you uninstall or reinstall.

Saving for a Previous Version

You can save VIs, LabVIEW projects and project libraries for a previous version of LabVIEW.

- To save all VIs in your project for a previous version of LabVIEW by default, you can define the **Save version** in the project and project library.
- To save a copy of a VI or VI hierarchy for a previous version of LabVIEW, as separate from the current project, you can use the **Save for Previous Version** option in the **File** menu.

Defining the Save Version for all VIs

You can configure a project to save all project VIs for a previous version of LabVIEW by specifying a save version in your LabVIEW project and project library.

1. Enable **Separate compiled code from source file** in both the VI and library settings:
 - In **VI Properties** dialog box, select **General** from the **Category** list, and select the **Separate compiled code from source file** checkbox.
 - In **Project Library Properties** dialog box, select the **Separate compiled code from source file** checkbox on the **General Settings** page.
2. In your project, select **Project»Properties** to open the **Project Properties** dialog box.
3. Select a version of LabVIEW in the **Save version** option.



NOTE

You can save VIs, projects, and libraries for as early as LabVIEW 2017. Select **Editor Version** to save to the current editor version.

4. Click **OK**.

When you save your VIs, LabVIEW saves them for the selected version by default.

You can also define the save version for the library in the **Project Library Properties** dialog box. If you select **Default**, the save version in the library inherits that of the project. If you select a specific version, the save version defined in the library supersedes that of the project.



NOTE

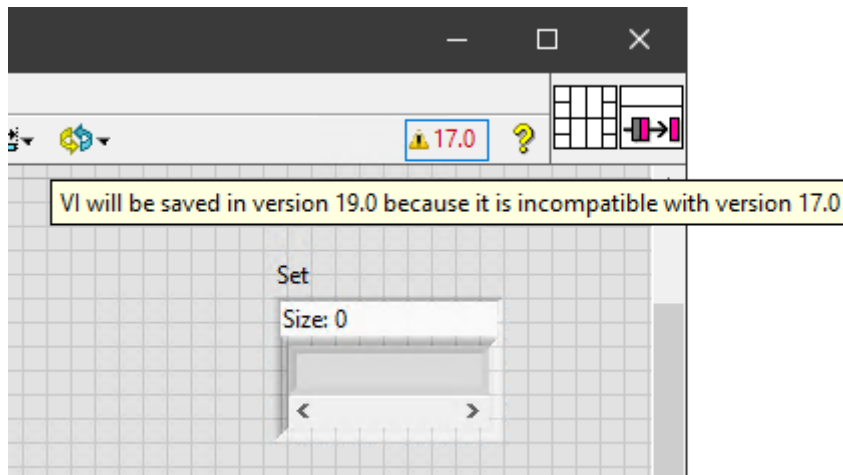
You can also set the save version for an entire directory by creating a file named `.lvversion` in the directory. The file should contain a single line specifying the save version, such as `21.0`. This file-specified save version affects all VIs loaded from the directory or any subdirectories.

If the file-specified save version differs from the save version of the project, LabVIEW saves the VI in the earlier of the two versions. If a specific save version is defined in the library, the save version in the library supersedes both the file-specified and project save versions.

Save Version Indicator

When you save a VI for a version of LabVIEW different from the current editor version, a **Save version** indicator appears to the left of the VI icon on the toolbar.

If LabVIEW cannot save a VI for the selected version due to compatibility issues, the indicator appears in red text with a warning glyph as shown in the following figure. You can hover over the indicator to see which compatible version is recommended.

Figure 4 : Save Version Indicator

You can click the indicator to open the **Error List** window and identify the code that causes compatibility issues. You can then update the code as needed.

Files that Cannot Save for a Previous Version

- LabVIEW-built applications, shared libraries, and PPLs can save for the current editor version only. For source distributions, in order for the defined save version of the project to apply, you need to configure the source distributions to save VIs as source-only. To do this, go to **Source Distribution Properties»Additional Exclusions** and select the **Separate compiled code from source file** checkbox.
- LabVIEW Real Time targets must match your current editor version for deployment.
- VIs on FPGA targets or VIs using FPGA nodes can save for the current editor version only, regardless of the defined save version of the project.

Using the Save for Previous Version Option

To save a particular VI or VI hierarchy for a previous version of LabVIEW, complete the following steps to use the **Save for Previous Version** Option.



NOTE

To save *packed project libraries* for a previous version of LabVIEW, save the files in the packed library for a previous version and *rebuild the packed library* with the desired version of LabVIEW.

1. For the top-level VI in a hierarchy of VIs, select **File»Save for Previous Version** to display the Save for Previous Version dialog box. For a project, select **File»Save for Previous Version** in the *Project Explorer* window. For a project library, right-click the library file in the **Project Explorer** window and select **Save for Previous**

Version from the shortcut menu, or open the project library and select **File»Save for Previous Version**.

2. Select the version of LabVIEW for which to save the VIs, project, or project library from the **LabVIEW Version** pull-down menu.



NOTE

When you save a VI hierarchy, project, or project library that uses a LabVIEW toolkit that has provided LabVIEW with a list of version information, you can select which version of the toolkit to save for in the **Toolkit Version** field.

3. Click the **Save** button to display a file dialog box.
4. Select the directory where you want to save the VI hierarchy, project, or project library.
5. Click the **Save** button.



NOTE

If you save a password-protected VI for a previous LabVIEW version, you must enter the password.

If a VI, project, or project library uses functionality that is not available in the previous version, LabVIEW saves as much as it can and produces a report of what it cannot convert. The report appears immediately in the Save for Previous Warning Summary dialog box.

Click the **Show Details** button in the **Save for Previous Warning Summary** dialog box to open the Load and Save Warning List dialog box and review the details of each warning. You also can click the **Ignore** button to acknowledge these warnings and close the summary dialog box. Click the **Save to File** button in the **Load and Save Warning List** dialog box to save the warnings to a text file you can review later.

You also can use the Save:For Previous method to save a VI for a previous version of LabVIEW programmatically, the Save For Previous method (Project) to save a LabVIEW project for a previous version of LabVIEW programmatically, and the Save:For Previous method (Library) to save a project library for a previous version of LabVIEW programmatically.

Saving Individual VI Files

Use one of the following ways to save an individual VI:

- Select **File»Save** to save changes to an existing VI in its current disk location or to save a new VI. If the VI is new, specify the name of the VI and its location in the file dialog box.
- Select **File»Save As** on a previously saved VI to display the Save As dialog box, in which you can save a copy of the VI in memory to disk with a new name you choose, rename the VI, or copy the VI hierarchy to a new location.
- Select **File»Save for Previous Version** to display the Save for Previous Version dialog box, in which you can save a VI for use in a previous version of LabVIEW.

Saving VIs in Existing LLBs

You can save a new VI in an existing **LLB** just as if it were a directory.

Complete the following steps to save a VI in an existing LLB.

1. Select **File»Save** to display a file dialog box. (macOS) If you use native file dialog boxes, click the **Use LLBs** button before you select an LLB.
2. Navigate to the LLB where you want to save the VI. The name of the LLB with the **.llb** extension appears in the file dialog box.



CAUTION

Do **not** save your own VIs and other files in **.llb files installed with LabVIEW**. *Save your controls and VIs in the user.lib directory* to add them to the **Controls** and **Functions** palettes.

1. Double-click the name of the LLB file or click the **OK** button to display the **Name the VI** dialog box.
2. Enter the VI name and click the **OK** button.

Customizing VIs

LabVIEW provides several options for customizing VIs. Select **File»VI Properties** or complete the following steps to access the options for customizing VIs.

1. Right-click the icon in the upper-right corner of the front panel or block diagram.
2. Select VI Properties from the shortcut menu.

If you want to change subVI settings during a specific call to that subVI, use the SubVI Node Setup dialog box.

You also can set VI properties programmatically.

Changing a VI Window Title

You can customize the VI window title to make it more descriptive than the VI filename. This is useful for **localized VIs** because you can translate the VI window title to the local language.

Complete the following steps to change the VI window title.

1. Select **File»VI Properties** to display the VI Properties dialog box.

2. Select Window Appearance from the **Category** pull-down menu at the top of the **VI Properties** dialog box.
3. Remove the checkmark from the **Same as VI name** checkbox.
4. Type a title for the VI in the **Window title** text box.
5. Click **OK** to exit the dialog box.

You also can use the Front Panel Window:Title property to change the VI window title programmatically.

Copying Browse Paths and Keywords

Complete the following steps to copy the task-based browse paths and keywords from one example VI to another.

1. Select **Tools»Prepare Example VIs for NI Example Finder** to display the Prepare Example VIs for NI Example Finder dialog box.
2. **Add** the task-based browse paths and keywords to an example VI.
3. Click the **Copy** button in the **Prepare Example VIs for NI Example Finder** dialog box to copy the task-based browse paths and keywords for the example VI from the **Documentation** table. You cannot copy only the task-based browse paths or only the keywords.
4. In the **Double-click the example to document** list, navigate to and double-click the example VI into which you want to copy the task-based browse paths and keywords.
5. (Optional) Click the **View** button to view the task-based browse paths and keywords before you paste them into the example VI. Click the **OK** button to close the **Example Browse Path(s) and Keyword(s) to Copy** window.
6. Click the **Paste** button to add the task-based browse paths and keywords to the example VI into which you want to copy the documentation.
7. Click the **Save Documentation** button to save the task-based browse paths and keywords to the example VI into which you copied the documentation.

Preparing Example VIs to Appear in the NI Example Finder

Complete the following steps to prepare example VIs you create to appear in the NI Example Finder and build the data `.bin3` file that stores the task-based browse paths, keywords, and other information about the example VIs you document. LabVIEW uses the `.bin3` file to display example VIs in the NI Example Finder.



NOTE

Make sure the example VIs you document are not write protected.

1. Select **Tools»Prepare Example VIs for NI Example Finder** to display the Prepare Example VIs for NI Example Finder dialog box.

- Click the browse button next to the **Start directory for examples** text box, navigate to the directory that contains the example VIs or projects you want to appear in the NI Example Finder, and click the **Select Folder** button.



Note The example VIs you created and the `.bin3` file must reside in the following subdirectories of the `labview` directory:

- example VIs— `examples`, `instr.lib`, or `user.lib`. For example, `labview\user.lib\my_examples` is an acceptable directory in which to include the example VIs. The example VIs also must reside in a directory structure identical to the directory structure in which they will reside on the user computer.
- `.bin3` file— `examples\exbins`, `instr.lib`, or `user.lib`

- In the **Double-click the example to document** list, navigate to and double-click the example VI you want to document. The VI name appears in the **Documentation** table. The ADE (application development environment) and version of the ADE in which you created the example VI or project appear in the **Documentation** table when you scroll up. You cannot change the **ADE** and **Version** values.
- Add task-based browse paths to the example VI for users to navigate to the example VI by task on the **Browse** page of the NI Example Finder.
 - In the **Double-click to add a browse path** list, choose an appropriate task-based browse path for the example VI.
 - Navigate to the lowest level of the category, then double-click the gray icon to add the category.

The task-based browse path appears in the **Documentation** table. You also can [copy browse paths](#) from one example VI to another. Double-click the browse path in the **Documentation** table to remove the browse path from the example VI.

- (Optional) Repeat step 4 to add more task-based browse paths to the example VI.
- Add keywords to the example VI for users to enter on the **Search** page of the NI Example Finder.
 - In the **Double-click to add a word** listbox, scroll through the list to find a keyword and double-click the word to add it to the example VI. The keyword appears in the **Documentation** table.
 - Enter a word into the **New keyword** text box and click the **Add New Keyword** button to add a new keyword to the **Double-click to add a word** listbox.
 - Double-click the new keyword in the **Double-click to add a word** listbox to add the keyword to the example VI. The keyword appears in the **Documentation** table.

You also can [copy keywords](#) from one example VI to another. Double-click the keyword in the **Documentation** table to remove the keyword from the example VI. Select the keyword in the **Double-click to add a word** listbox and click the **Remove Keyword** button to remove a keyword from the list of keywords.

5. (Optional) Repeat step 6 to add more keywords to the example VI.

Tip Click the **Save Documentation** button to save the example VI documentation you have added. After you save the example VI documentation, you can proceed through step 13 or you can close this dialog box and finish documenting the example VI later. Relaunch the Preparing Examples for NI Example Finder dialog box and navigate to the example VI in the **Double-click the example to document** list to edit the example VI documentation you saved.

6. If you have not *entered a description* for the example VI, enter a description in the **Example description** text box. The description appears in the **Description** text box of the NI Example Finder when a user selects the VI. If you already *entered* a description for the example VI on the Documentation page of the **VI Properties** dialog box, the description appears in the **Example description** text box.
7. Click the **Save Documentation** button to save the documentation to the example VI. Clicking this button overrides any existing task-based browse paths, keywords, and description you entered in the **Example description** text box or in the example VI on the **Documentation** page of the **VI Properties** dialog box.
8. (Optional) You also can *copy the description* from one example VI to another.
9. Repeat steps 1 through 10 for each VI you want to appear in the NI Example Finder.

Note If an example VI is part of a project-based example and that example VI can run independently of the LabVIEW project, you can document both the project and example VI. To document a project, navigate to and select the project file `.lvproj` in the **Double-click the example to document** list. Complete the steps listed above.

10. Build the `.bin3` file that stores the task-based browse paths, keywords, and other information about example VIs you document.

Note You must add a task-based browse path or a keyword for example VIs you created to appear in the NI Example Finder. If you add only a task-based browse path, the example VIs you created appear only on the **Browse** page of the NI Example Finder. If you add only a keyword, the example VIs you created appear only on the **Search** page of the NI Example Finder.

1. Click the **Build Data File** tab.
2. Click the browse button next to the **Data file to create (bin3)** text box, and navigate to the directory where you want to save the `.bin3` file. In order for LabVIEW to locate the `.bin3` file, you must save it in one of the following locations within the `labview` directory:
 - `examples\exbins`
 - `instr.lib`
 - `user.lib`
3. Enter a meaningful name for the `.bin3` file and include the file extension. For example, enter `myexamples.bin3`, where `myexamples` is the name of the `.lib` that contains the example VIs you created. After you enter a name, click the **OK** button.

Note LabVIEW reserves the following prefixes and you cannot use them when you name the .bin3 file: ms, test, lv, daq, cvi, tree, fr, de, ja, vi, im, dsc, rt, _, ko, zh, it, and es.

4. Click the browse button next to the **Directory that contains the examples** text box, navigate to the directory that contains the example VIs you want to appear in the NI Example Finder, and click the **Select Folder** button.
5. Click the **Build File** button to build the data file.
11. Click the **Done** button to exit the **Prepare Example VIs for NI Example Finder** dialog box.

Setting Execution Properties

LabVIEW includes several execution properties you can use to customize VIs.

Complete the following steps to determine the execution properties for the VI.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Execution from the **Category** pull-down menu at the top of the **VI Properties** dialog box.
3. Select the appropriate execution properties.

You also can use the Execution properties to set execution properties programmatically.

Setting Window Appearance and Size

LabVIEW includes several options you can use to customize the window appearance, window position, and window size for VIs. These options are useful when distributing VIs as stand-alone applications.

Complete the following steps to customize window appearance and window size.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Window Appearance, Window Size, or Window Run-Time Position from the **Category** pull-down menu.
3. Select the appropriate window options.
4. In the **Window Appearance** page, click the **Customize** button to display the Customize Window Appearance dialog box. In this dialog box, you can customize options such as whether the title bar or menu bar appears, whether the user can

close or minimize the window, whether shortcut menus appear at run time, and more.



NOTE

If you remove the checkmarks from the **Show toolbar when running**, **Show Abort button**, and **Allow user to close window** checkboxes, consider adding a stop button to the front panel so the user can stop the VI.

You also can use the Front Panel Window and Tool Bar properties and Front Panel methods to customize the window appearance and window size programmatically. You also can use the Front Panel:Run-Time Position methods to customize the run-time front panel window position and size programmatically.

Running VIs

Running a VI executes the operation for which you designed the VI. You can run a VI if the **Run** button on the toolbar appears as a solid white arrow. The solid white arrow also indicates you can use the VI as a subVI if you [create a connector pane](#) for the VI.

You can configure how a VI runs by selecting **File»VI Properties** and selecting Execution from the **Category** pull-down menu.

Complete the following steps to run a VI on the block diagram:

1. Wire all the subVIs, functions, and structures with the correct data types for the terminal.
2. Click the **Run** or **Run Continuously** buttons or the single-stepping buttons on the block diagram toolbar. Clicking the **Run** button runs the VI once. The VI stops when the VI completes its data flow. Clicking the **Run Continuously** button runs the VI continuously until you stop it manually. Clicking the [single-stepping](#) buttons runs the VI in incremental steps.



NOTE

The **Abort Execution** button stops the VI immediately, before the VI finishes the current iteration. Aborting a VI that uses external resources, such as external hardware, might leave the resources in an unknown state by not resetting or releasing them properly. Design VIs with a stop button to avoid this problem.

You also can run a VI outside of the block diagram in the following ways:

- Run a VI from a [LabVIEW Project](#) by right-clicking a VI in the [Project Explorer](#) window and selecting **Run** from the shortcut menu. The VI front panel opens in run mode.
- Run a VI programmatically using the Run VI method.
- Run a VI programmatically using the [RunVI](#) command line operation.

Launching a VI from the Command Line

To launch a VI from the command line (such as by selecting **Start»Run** on Windows), use the following syntax:

```
c:\labview\labview.exe <path to VI relative to the labview directory>
```



NOTE

You must close LabVIEW before launching a VI from the command line.

For example, to launch `readme.vi` in `c:\labview\examples`, use the following command:

```
c:\labview\labview.exe examples\readme.vi
```

If the VI is located in a directory other than the `labview` directory, you must specify the full path to the VI, as shown in the following example:

```
c:\labview\labview.exe c:\coolapp\mycool.vi
```

If a directory in the path contains spaces, you must enclose the path in quotation marks, as shown in the following example:

```
c:\labview\labview.exe "c:\cool application\mycool.vi"
```

If the VI is in an LLB, use one of the following ways:

- Place the full path in quotation marks, as shown in the following example:

```
c:\labview\labview.exe "c:\coolapp\eagle.llb\mycool.vi"
```
- Select **Tools»LLB Manager** and *mark the VI as top level*. In this case, you need to specify only the path to the LLB. LabVIEW opens the LLB Manager window, as shown in the following example:

```
c:\labview\labview.exe "c:\coolapp\eagle.llb"
```

You also can pass *LabVIEW-defined* or user-defined arguments when you launch LabVIEW from the command line. To pass user-defined arguments in the command line, enter two hyphens `--` surrounded by spaces before the set of user-defined arguments. LabVIEW does not use any arguments after the two hyphens to launch `labview.exe`. LabVIEW passes the arguments after the two hyphens to the block diagram of the VI you launch. Use the Application:Command Line Arguments property to read the user-defined command-line arguments passed when LabVIEW launches.

For example, to launch `mycool.vi` and pass user-defined arguments, use the following command:

```
c:\labview\labview.exe c:\coolapp\mycool.vi -- 1000 sine
```

On the block diagram of the `mycool.vi`, use the `Application:Command Line Arguments` property to read the `1000` and `sine` arguments you passed and handle the values. For example, you can wire the argument values to the case selector on a [Case structure](#).

LabVIEW-Defined Command-Line Arguments

When you [launch a VI from the command line](#), you can use the following LabVIEW-defined arguments. LabVIEW-defined arguments apply to the executable, which is either `labview.exe` or a stand-alone application. If the command line includes any other string before the two hyphens `--` that indicate the start of [user-defined arguments](#), the executable interprets the string as the filename of a VI to open.

Platforms	Argument	Description
All	<code>-pref path</code>	If a preferences file does not reside in the same directory as the executable you want to launch, use this argument to specify the full path of the preferences file.
Windows	<code>/Automation</code> or <code>-Automation</code>	Launches an executable in response to an ActiveX client requesting an instance of an object in the executable.
Windows	<code>/RegServer</code> or <code>-RegServer</code>	Registers the ActiveX server of an executable with the operating system.
Windows	<code>/UnregServer</code> or <code>-UnregServer</code>	Unregisters the ActiveX server of an executable from the operating system.
Linux	<code>-display display</code>	Runs LabVIEW on the given X Windows display. For example, <code>labview -display myhost:0</code> .
Linux	<code>-launch</code>	LabVIEW launches and checks for an existing application instance already running on the same display. If an existing instance is running, the existing instance opens the VIs passed in the command line, and the new application instance quits silently. If an existing instance is not running, LabVIEW opens the VIs passed in the command line in the new instance.

Adding Comments to the Revision History

Complete the following steps to add a comment to the **History** window.

1. Select **Edit»VI Revision History** to display the History window.
2. Enter a description of the changes you made to the VI in the **Comment** text box near the top of the window.
3. Click the **Add** button to manually add the comment to the history for the VI. If the **Comment** text box contains a comment when you save the VI, LabVIEW

automatically adds the comments to the history. The header of the comment includes the next revision number. After you add a comment, it is a permanent part of the history. You cannot edit comments in the VI history.

Leave the **History** window open while you edit a VI so you can enter information in the **Comment** text box every time you make a change. To include an entry in the revision history each time you save the VI, even if you do not enter a comment, select **History** in the **History** window and make sure there is a checkmark next to **Show Empty Entries**.



NOTE

By default, LabVIEW does not create history comments automatically. Anyone who makes changes to the VI must enter a description of the changes in the **History** window to keep the history up to date. You can configure LabVIEW to record comments generated by LabVIEW in the *history for the current VI* by placing a checkmark in the **Record comments generated by LabVIEW** checkbox on the Revision History page of the **VI Properties** dialog box. You also can configure LabVIEW to record comments generated by LabVIEW in the history for all new VIs by placing a checkmark in the **Record comments generated by LabVIEW** checkbox on the Revision History page of the **Options** dialog box.

Resetting the VI Revision History and Revision Number

Complete the following steps to reset the VI revision history and revision number.

1. Select **Edit»VI Revision History** to display the History window.
2. Click the **Reset** button. The **Reset** button is dimmed if the VI has no revision history.
3. In the dialog box that appears, click the **OK** button to erase the VI revision history and reset the revision number.



NOTE

To erase the VI revision history without resetting the revision number, remove the checkmark from the **Also reset revision number** checkbox.

You also can use the Clear History method to clear the revision history of a VI programmatically.

Viewing the VI Revision History

Select **Edit»VI Revision History** to display the History window. The **History** window shows a header for each comment in the history. The header includes the revision number for the change, the revision date and time, and the user name.

To reduce the size of the **History** window, click the **History** arrow—the small black triangle. To expand the size of the **History** window, click the same arrow.

Resize the **History** window to change the size of the **History** and **Comment** text boxes.

Debugging VIs

Disabling Debugging Tools

You can disable the debugging tools for a VI to reduce memory requirements and to improve performance slightly.

Complete the following steps to disable debugging for a VI.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Execution from the **Category** pull-down menu.
3. Remove the checkmark from the **Allow debugging** checkbox.
4. Remove the checkmark from the **Enable automatic error handling** checkbox.

You also can use the Allow Debugging property to disable the debugging tools programmatically and the Automatic Error Handling property to disable automatic error handling programmatically.

Complete the following steps to disable automatic error handling for any new, blank VIs you create.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Block Diagram from the **Category** list.
3. Remove the checkmark from the **Enable automatic error handling in new VIs** checkbox.

Disabling Sections of Block Diagrams

You can run a VI with a section of the block diagram disabled, similar to commenting out a section of code in a text-based programming language.

Complete the following steps to run a VI with a section of the block diagram disabled.

1. Place a Diagram Disable structure around the nodes you want to disable. The default subdiagram is **Disabled**.
2. In the **Enabled** subdiagram, include any code that you want to enable in place of the code in the **Disabled** subdiagram.
3. Run the VI. Notice that only the code in the **Enabled** subdiagram is run.

To execute the section of the block diagram you disabled, select the **Disabled** subdiagram, right-click the structure border, and select **Remove Diagram Disable Structure** from the shortcut menu.



NOTE

When you select **Remove Diagram Disable Structure**, LabVIEW deletes objects in all subdiagrams except for the visible subdiagram.

Refer to the Diagram Disable Structure VI in the labview\examples\Structures\Disable Structures directory for an example of using the Diagram Disable structure.

Displaying Warnings

Warnings do not prevent you from running a VI. They are designed to help you avoid potential problems in VIs.

Complete the following steps to display warnings in the Error list window.

1. Select **View»Error List** to display the **Error list** window.
2. Place a checkmark in the **Show Warnings** checkbox to display all warnings for each VI in the **Items with errors** section.
3. Click the **Warning** button on the toolbar to access the **Error list** window. The **Warning** button appears only if the VI has a warning. After you configure the **Error list** window to show warnings, you can access the **Error list** window any time the **Warning** button appears on the toolbar.

Complete the following steps to configure LabVIEW to always show warnings in the **Error list** window.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Environment from the **Category** list.
3. Place a checkmark in the **Show warnings in Error List dialog by default** checkbox in the **Debugging** section of this page and click the **OK** button. You can make this change with the **Error list** window open and see the change immediately.

Execution Highlighting

Use execution highlighting to view an animation of the execution of the block diagram. Execution highlighting shows the movement of data on the block diagram from one node to another using bubbles that move along the wires. Use execution highlighting in conjunction with [single-stepping](#) to see how data moves from node to node through a VI.



NOTE

Execution highlighting greatly reduces the speed at which the VI runs.

Complete the following steps to use execution highlighting.

1. Open the block diagram of any VI.
2. Click the **Highlight Execution** button on the block diagram toolbar to enable execution highlighting.
3. Run the VI and view the block diagram as the VI runs. Notice that with execution highlighting, bubbles move along the wires to mark the movement of data from one node to another and the values at each terminal are displayed.
4. Stop the VI.
5. Click the **Highlight Execution** button again at any time to disable execution highlighting.

Single-Stepping through a VI

Single-step through a VI to view each action of the VI on the block diagram as the VI runs. The single-stepping buttons affect execution only in a VI or subVI in single-step mode. You can single-step through subVIs or run them normally.

Complete the following steps to single-step through a VI.

1. Click the **Start Single Stepping** button on the block diagram toolbar.
2. Click one of the following buttons:



Step Into —Opens a node and pauses. When you click the **Step Into** button again, it executes the first action and pauses at the next action of the subVI or structure. You also can press the <Ctrl> key and the down arrow key. (macOS) Press the <Command>-down arrow keys. (Linux) Press the <Alt>-down arrow keys.



Step Over —Executes a node and pauses at the next node. You also can press the <Ctrl> key and the right arrow key. (macOS) Press the <Command>-right arrow keys. (Linux) Press the <Alt>-right arrow keys.



Step Out —Finishes executing the current node and pauses. When the VI finishes executing, the **Step Out** button becomes dimmed. You also can press the <Ctrl> key and the up arrow key. (macOS) Press the <Command>-up arrow keys. (Linux) Press the <Alt>-up arrow keys.

When you single-step through a VI, nodes blink to indicate they are ready to execute.

Move the cursor over the **Step Over**, **Step Into**, or **Step Out** button to view a tip strip that describes the next step if you click that button. If you single-step through a VI with *execution highlighting* on, an execution glyph appears on the icons of the subVIs that are currently running.

3. End single-stepping by clicking the **Step Out** button when a blinking border surrounds the entire block diagram. The stepping buttons are dimmed when single-stepping is no longer active.

Suspending Execution

Suspend execution of a subVI to edit values of controls and indicators, to control the number of times the subVI runs before returning to the caller, or to go back to the beginning of the execution of the subVI. You can cause all calls to a subVI to start with execution suspended, or you can suspend a specific call to a subVI.

Complete the following steps to suspend all calls to a subVI.

1. Open the subVI.
2. Select **Operate»Suspend when Called**.

The subVI automatically suspends when another VI calls it. If you select this menu item when *single-stepping*, the subVI does not suspend immediately. The subVI suspends when it is called.

When you select **View»VI Hierarchy**, a VI configured to suspend when called appears with a green exclamation point !. A suspended VI appears with a red exclamation point.

Complete the following steps to suspend execution at a specific subVI call.

1. Right-click the subVI node and select **SubVI Node Setup** from the shortcut menu. The SubVI Node Setup dialog box appears.
2. Place a checkmark in the **Suspend when called** checkbox and click the **OK** button.

You also can use the Suspend On Call property to suspend a VI programmatically.

After you suspend execution of a subVI, you can perform the following tasks:

- Click the **Return to Caller** button at the top left of the subVI front panel to return to the caller VI. The **Return to Caller** button appears when a suspended subVI is not running. Notice that you can return to a caller VI without running the current VI. If you want to run the current VI, click the Run button before you return to the caller.
- Run the subVI as many times as you want before returning to the caller VI. Click the **Run** button or select **Operate»Run**. To go back to the beginning of the execution of the subVI when the subVI is running, click the **Skip to Beginning** button at the top left of the subVI block diagram.
- Change the control or indicator values if you want to return different values to the caller VI. You can set values of indicators only when a suspended VI is not running. Click the **Return to Caller** button at the top left of the subVI front panel to send the indicator values of the subVI to the caller VI.

Memory Usage

Profiling VI Execution Time and Memory Usage

The Profile Performance and Memory window is a powerful tool for *analyzing* how your application uses execution time and memory. With this information, you can identify the specific VIs or parts of VIs you need to optimize. For example, if you notice that a particular subVI takes a long time to execute, you can focus your attention on *improving the performance of that VI*.

Complete the following steps to profile a VI.

1. Stop all running VIs. If you do not stop these VIs, the results that the **Profile Performance and Memory** window displays can be misleading or inconsistent.
2. Select **Tools»Profile»Performance and Memory** to display the **Profile Performance and Memory** window.
3. If you want to collect memory usage information, place a checkmark in the **Profile memory usage** checkbox. You cannot place a checkmark in this checkbox after starting the profiling session. Collecting information about VI memory use adds a significant amount of overhead to VI execution, which affects the accuracy of any timing statistics you gather during the profiling session. Therefore, you should perform memory profiling separate from time profiling.
4. Click the **Start** button in the **Profile Performance and Memory** window to begin the collection of performance data.
5. Run the VI you want to profile.
6. Run the VI to completion, or click the front panel stop button to stop the VI if it is running continuously. If there is no stop button, click the **Abort Execution** button.
7. Click the **Stop** button in the **Profile Performance and Memory** window to end the profiling session. The tabular display includes the time it took the top-level VI to run, the time it took all subVIs to run, and the total time it took the VI and all subVIs to run within this profiling session.



NOTE

The **Profile Performance and Memory** window measures only CPU usage time. During wait functions, the CPU is free to process other tasks in the operating system, but the **Profile Performance and Memory** window still measures the small amount of time the CPU takes to initiate and return from the wait. The **Profile Performance and Memory** window does not measure the CPU usage time for calling by reference or for using control references in Property Nodes separately from the CPU usage time for other nodes.

1. If you place a checkmark in the **Timing statistics** checkbox, the tabular display includes more statistics about the VI run times. If you place a checkmark in the **Timing details** checkbox, the tabular display includes several timing categories that can help you determine what operations take the most time. If you place a checkmark in the **Memory usage** checkbox, which is only available if you place a

checkmark in the **Profile memory usage** checkbox before you begin the profiling session, you can view information about how your VIs are using memory.

In addition to the **Profile Performance and Memory** window, the LabVIEW VI Analyzer Toolkit provides tests that check VIs interactively or programmatically for performance, efficiency, and other aspects of LabVIEW programming.

You also can use the LabVIEW Desktop Execution Trace Toolkit to monitor code analysis, memory leaks, and other aspects of LabVIEW programming.

Identifying Buffer Allocations in LabVIEW Applications

Use the Profile Buffer Allocations window to identify and analyze buffer allocations in a LabVIEW application.

Identifying Large Buffers

To identify the large buffers that an application allocates, click **Start** to start profiling, run a VI or multiple VIs that you want to profile, and click **Stop** to end the profiling session. The **Profile Data** table displays the largest buffer allocations. The number of buffer allocations in the table is no greater than the **Maximum buffers to display**.

Analyzing Buffer Allocations

To analyze the cause of a buffer allocation, double-click a row in the **Profile Data** table to open the block diagram of the VI that causes the buffer allocation. LabVIEW highlights the node, wire, or tunnel which causes the buffer allocation.

Saving and Loading Profiles

To save the profiles for analysis, select **File»Save Profile** to save the profile data to a binary file on the disk. To load the saved profile back into this window, select **File»Load Profile**. You can also export the profile data to a CSV file by selecting **File»Export Table**. You can load the CSV file in either Microsoft Excel or Notepad.

Optimizing Execution Speed for Built Applications

After you finish developing and testing a VI or built application, you can configure settings in the LabVIEW environment and the build specification to maximize the execution speed of the application. You must wait until you finish development because these configurations can slow or prevent editing and debugging.

The following procedure improves execution speed by ensuring that LabVIEW uses full compiler optimizations to build the application.

1. **Open the build specification.** Right-click the build specification to display the Properties dialog box.
2. **Ensure the Enable Debugging option for the build specification does not contain a checkmark.** On the **Advanced** page of the dialog box, remove the checkmark from the **Enable debugging** checkbox. This global debugging setting allows LabVIEW to evaluate the debugging setting on individual VIs in the build specification. Therefore, in order to later disable debugging on the individual VIs in the build specification, you must not enable debugging in this global setting.
3. **Disable debugging in the build specification for each VI.** Disabling debugging for each VI improves the execution speed of the application. Additionally, the debugging setting sometimes causes LabVIEW to recompile the VI, which is necessary to apply updated compiler settings that affect execution speed. The rest of this procedure ensures that you confirm these settings and force recompilation to occur.
 1. On the **Source File Settings** page of the dialog box, select each VI and click the **Customize VI Properties** button.
 2. Remove the checkmark from the **Use VI Property** checkbox next to **Allow debugging**.
 3. Remove the checkmark from the **Allow debugging** checkbox.
4. **Disable debugging in the build specification for dependencies.** Disabling debugging for dependencies ensures that dependency VIs benefit from the same execution speed and recompilation effects as the other VIs in the build specification.
 1. On the **Source File Settings** page of the dialog box, select **Dependencies** and click the **Customize VI Properties** button. If the **Customize VI Properties** button is disabled, place a checkmark in the **Set VI properties for all contained items** checkbox to enable it.
 2. Remove the checkmark from the **Use VI Property** checkbox next to **Allow debugging**.
 3. Remove the checkmark from the **Allow debugging** checkbox.
5. **Ensure all source VIs are configured to allow debugging.** By allowing debugging in the source VIs, you create an intentional discrepancy between the VI settings and the build specification settings from steps 3 and 4. This discrepancy ensures that LabVIEW recompiles the VIs when you build the application, thereby applying changes you make to the complexity threshold in later steps.



NOTE

Complete the following steps only if you changed the debugging settings on any individual VIs. LabVIEW allows debugging on VIs by default.

1. Open each VI in the build specification and dependencies list.
2. Place a checkmark in the **Allow debugging** checkbox on the Execution page of the VI Properties dialog box.
6. **Configure the build specification to use all additional exclusions.** The additional exclusions improve both the load time and execution speed of the application. On the **Additional Exclusions** page of the **Properties** dialog box for the build specification, place a checkmark in every checkbox.
7. **Use a pre-build VI to set the compiler optimizations threshold before building the application.** Programmatically setting the compiler optimizations threshold in

a pre-build VI ensures that the change is present when LabVIEW recompiles the VI during the build process.

1. On the **Pre/Post Build Actions** page of the **Properties** dialog box for the build specification, click the **Generate VI** button to generate a new pre-build VI with the appropriate connector pane.
 2. Close the **Properties** dialog box for the build specification and open the pre-build VI.
 3. In the pre-build VI, programmatically set the Application:Compiler Threshold property to 10. You must *enable VI Scripting* to use the Application:Compiler Threshold property.
8. **Use a post-build VI to restore the compiler optimizations threshold to 5 after building the application.** Programmatically restoring the compiler optimizations threshold in a post-build VI ensures that all other VIs that you develop correctly prioritize between editor responsiveness and VI execution speed.
1. On the **Pre/Post Build Actions** page of the **Properties** dialog box for the build specification, click the **Generate VI** button to generate a new post-build VI with the appropriate connector pane.
 2. Close the **Properties** dialog box for the build specification and open the post-build VI.
 3. In the post-build VI, programmatically set the Application:Compiler Threshold property to 5.
9. **Build the application.** Right-click the build specification and select **Build** from the shortcut menu.

Note If LabVIEW runs out of memory while building the application, configure the pre-build VI to use a compiler optimizations threshold of 9 instead of 10. Continue lowering the threshold until LabVIEW does not run out of memory.

Choosing between Editor Responsiveness and VI Execution Speed

To allow you to edit large VIs without significant delays, LabVIEW automatically *limits compiler optimizations* when the *complexity of a VI* exceeds a certain threshold. As a tradeoff for limiting these optimizations, these large compiled VIs may not run at maximum execution speed. To give you control over when LabVIEW prioritizes editor responsiveness over execution speed, you can adjust the complexity threshold at which LabVIEW begins limiting optimizations.



TIP

You can avoid this tradeoff completely by breaking your large VIs into subVIs. This approach reduces the complexity of the VI, thereby optimizing editor responsiveness, execution speed, and code readability.

If you cannot divide a large VI further into subVIs, consider adjusting the complexity threshold in the following situations:

- A large VI causes delays *when LabVIEW compiles the VI*.

- LabVIEW runs out of memory when compiling a VI.
- You finish developing a large or complex VI and want to optimize it for execution speed.

What to Do

Complete the following steps to choose between editor responsiveness and VI execution speed:

1. **Identify a VI for which you want to change either the editor responsiveness or execution speed.** Although the complexity threshold is an environmental setting that applies to all VIs, you can use the complexity of an individual VI to decide whether the current threshold accomplishes your goals for that VI.
2. **Identify the complexity of the VI.** Refer to the **Compiled Code Complexity** value on the Memory Usage page of the VI Properties dialog box.
3. **Compare the complexity of the VI to the threshold for compiler optimizations.** Refer to the **Compiler optimizations** slider on the Environment page of the Options dialog box. The value of the slider indicates the VI complexity above which the compiler limits optimizations to improve editor responsiveness. If the complexity of a VI is lower than the value of the slider, the compiler applies optimizations to the VI.
4. **Adjust the complexity threshold if doing so accomplishes your goals.**
 - To prioritize editor responsiveness or prevent out of memory errors, lower the slider below your VI complexity.
 - To prioritize execution speed, raise the slider above your VI complexity.
5. **Recompile the VI to apply changes.** (Windows) Press <Ctrl> while you click the **Run** button. (macOS) Press <Command> while you click the **Run** button. (Linux) Press <Alt> while you click the **Run** button.
6. **Change the complexity threshold to 5 after you finish developing or deploying the VI.** Because the complexity threshold applies to all VIs, National Instruments recommends that you generally use the default value of 5 for the threshold. This value ensures that LabVIEW correctly prioritizes editor responsiveness and execution speed for the vast majority of VIs.

Tips and Troubleshooting Information

- To avoid the tradeoff between editor responsiveness and execution speed, reduce the complexity of large VIs by moving sections of code into subVIs. This approach simultaneously optimizes editor responsiveness, execution speed, and code readability.
- If you cannot identify which VI in a project is causing a long build time or an out of memory error, decrease the **Compiler optimizations** slider by one increment at a time until the problem improves.
- The complexity threshold applies to all VIs. Therefore, if you leave the threshold set at a non-default value, other VIs might exhibit unexpected changes in editor responsiveness or execution speed.
- The compiler ignores the complexity threshold when building an application for targets running on PowerPC or ARM architecture (VxWorks and NI Linux Real-Time

targets, respectively). These targets require VIs to use certain compiler optimizations. Therefore, lowering the complexity threshold below the VI complexity might not allow LabVIEW to build the application much faster, and raising the threshold above the VI complexity might not improve execution speed.

- You can programmatically interact with VI complexity and the complexity threshold by using the Compiled:Code Complexity property, the Compiled:Last Compiled With property, and the Application:Compiler Threshold property.

Resolving Out of Memory Errors When Compiling a VI

The LabVIEW compiler can run out of memory when it tries to compile large or complex VIs. Choose one of the following solutions to correct this problem.

Possible Cause	Solution
The complexity of a VI falls below the threshold at which the compiler begins limiting compiler optimizations to preserve memory and facilitate editor responsiveness.	Lower the complexity threshold to <i>prioritize editor responsiveness over execution speed</i> for the relevant VIs.
A VI is too complex to compile even though the compiler is limiting optimizations.	(Recommended) Reduce the complexity of the VI by dividing the VI into subVIs.

VI Strings

Exporting and Importing VI Strings

Complete the following steps to export and import the strings to *localize* a VI.

1. Select **Tools»Advanced»Export Strings** to export the VI strings to a text file.
2. In the file dialog that appears, navigate to and select a location for the text file. The default name for the text file is the same as the VI name. Click the **OK** button.
3. Click the **Yes** button in the message box that appears to localize captions names.
4. Click the **Yes** button in the message box that appears to localize the block diagram strings.
5. Open the text file to localize the VI strings and save your changes.
6. Select **Tools»Advanced»Import Strings** to import the VI strings.
7. Navigate to the text file in the **Select a VI string file to read** dialog box. Click the **OK** button.
8. LabVIEW creates a log file that contains a list of any errors that occurred during the operation and prompts you to save. Click the **OK** button.

LabVIEW does not import a VI string file if the file contains unknown or missing tags. If you incorrectly localize the VI string, it may contain unknown or missing tags. If you modified the VI between the time you export the strings and the time you import your translation, the VI string file may contain unknown or missing tags. Refer to the [Example of Exported VI Strings](#) topic for an example text file containing exported VI strings.

Example of Exported VI Strings

The following example text demonstrates the contents of a text file that contains exported VI strings. Refer to [exporting the VI strings](#) for more information about exporting VI strings.

```
<VI syntaxVersion=11 LVversion=8508002 revision=13
name="Demo Recursion.vi">
  <TITLE><NO_TITLE name="Demo Recursion.vi"></TITLE>
  <HELP_PATH></HELP_PATH>
  <HELP_TAG></HELP_TAG>
  <RTM_PATH type="default"></RTM_PATH>
  <DESC>This demo shows how you can create a recursive VI
in LabVIEW. Recursion is when a VI uses itself on its own
block diagram. Such functionality is useful when
implementing VIs to solve problems of the form  $F(n) = F(n-1) + G()$ , where the functions may be arbitrarily
complex and more direct iterative methods may not be easy
to implement. </DESC>
  <CONTENT>
    <GROUPER>
      <PARTS></PARTS>
    </GROUPER>
    <CONTROL ID=80 type="Numeric" name="How many times to
recurse?">
      <DESC></DESC>
      <TIP></TIP>
      <PARTS>
        <PART ID=82 order=0 type="Caption">
          <LABEL>
            <STEXT>How many times to recurse?</STEXT>
          </LABEL>
        </PART>
      </PARTS>
    </CONTROL>
    <CONTROL ID=81 type="String" name="Resulting String">
      <DESC></DESC>
      <TIP></TIP>
      <PARTS>
        <PART ID=11 order=0 type="Text">
          <LABEL>
            <STEXT></STEXT>
          </LABEL>
        </PART>
        <PART ID=82 order=0 type="Caption">
          <LABEL>
            <STEXT>Resulting String</STEXT>
          </LABEL>
        </PART>
      </PARTS>
    </CONTROL>
  </CONTENT>
</VI>
```

```

    </PARTS>
    <DEFAULT><SAME_AS_LABEL></DEFAULT>
  </CONTROL>
  <LABEL>
    <STEXT>This demo uses recursion -- that is, a VI
that calls itself on its own block diagram -- to build a
string of numbers. The task itself is trivial and could be
easily achieved without using recursion, but the demo will
let you see the details of recursion which may be useful
for solving harder problems.
    <LF>
    <LF>
    <FONT style='B'>Instructions:<FONT
predef=APPFONT>
    <LF> 1. Select the number of times that the
demo should recurse.
    <LF> 2. Run VI.
    <LF> 3. The output will be a string of numbers
starting at zero and going up to your input number.
    <LF>
    <LF> Details on how the recursion works may be
found on the block diagrams of this VI and its subVIs.
  </STEXT>
</LABEL>
</CONTENT>
<BDCONTENT>
  <LABEL>
    <STEXT>Look in here.</STEXT>
  </LABEL>
  <NODE ID=49 type="SubVI" subVIName="Read String.vi">
    <DESC></DESC>
  </NODE>
  <DESC></DESC>
  <TIP></TIP>
  <PARTS></PARTS>
  <NODE ID=260 type="Object Method SubVI"
subVIName="Recurse.vi">
    <DESC></DESC>
  </NODE>
</BDCONTENT>
</VI>

```

Comparing and Merging VIs

Comparing VIs and VI Hierarchies in LabVIEW

Consult the following topics for information on how to compare VIs and VI hierarchies using the tools included in LabVIEW.

Comparing VIs

The **Compare VIs** dialog box helps you track changes in an application by comparing multiple versions of a VI using graphical differencing. Version control becomes more important as a project grows and involves more developers.

Before you begin

LabVIEW cannot load two VIs with the same name. You must rename one of your VIs to compare them. Renaming the VIs does not affect the name of the subVIs. If you compare two hierarchies of VIs, LabVIEW temporarily renames the top-level VI.

1. Open a LabVIEW project.
2. Select **Tools»Compare»Compare VIs** from the front panel or block diagram menu of a VI to display the **Compare VIs** dialog box.
3. Click **Select** to open the **Select a VI** dialog box and select a VI by name. You can only select VIs that are already loaded into memory.
4. Select comparison options in the **Differences** dialog box.
5. Click **Compare**.



NOTE

The progress bar indicates the steps in the comparison algorithm, not the number of differences left to find.



NOTE

Use the same steps to compare private data controls of LabVIEW classes. You must place each version of the class in a different project and enter the project paths into the **VIs to Compare** text fields using the following syntax:

```
ProjectName.lvproj\My Computer
\ClassName.lvclass:ClassName.ct1.
```

Comparing very large VIs can be time consuming. You can cancel the comparison of two VIs in the **Comparison Progress** dialog box.

When the comparison is complete, LabVIEW displays the front panels and block diagrams of the two VIs along with the **Differences** dialog box. You can specify the types of differences you want to detect and view in the **Differences** dialog box. For example, you can filter out cosmetic changes such as objects being moved or resized. If you select an item from the summary, LabVIEW displays and highlights the differences between the two VIs.

**NOTE**

LVCompare.exe also uses the **Compare VIs** dialog box to compare VIs using a third-party source control provider or from the command line.

After the comparison is complete, you can create a VI comparison report by clicking **More Commands»Create Report** in the **Differences** dialog box. In the **Create Report** dialog box, select a report type and specify a filepath to save the report as:

- A web page (.html)
- An XML document (.xml)
- A text file (.txt)
- A Microsoft Word document (.docx (Windows)).

**NOTE**

The file path for saving the comparison report must be a local directory instead of a network path.

You also can create a comparison report when you compare VI hierarchies.

Comparing VI Hierarchies

Use the **Compare VI Hierarchies** dialog box to compare two hierarchies of VIs. LabVIEW compares any file with the same name in both hierarchies and categorizes them as being the same or different. VIs and subVIs with the same name and path are considered shared VIs. All other VIs are categorized as added or deleted in the second VI hierarchy.

After the comparison completes, the **Compare VI Hierarchies** dialog box displays a summary of the differences. The **VIs in Hierarchies** listbox displays the VIs and a symbol that indicates how they are categorized.

You can select a set of VIs that have differences and visually compare them using the **Compare VIs** dialog box.

Because LabVIEW cannot compare two files with the same name, LabVIEW renames one of the VIs for comparison. The first VI is left as is. The second VI is moved to the temporary directory and prepended with a cmp. prefix. The renamed VI links to the subVIs loaded by the first VI.

1. Open a project in LabVIEW.
2. Select **Tools»Compare»Compare VI Hierarchies** from the front panel or block diagram menu of a VI to display the **Compare VI Hierarchies** dialog box.
3. Click **Browse Files** to select VIs from the file system to compare.
4. Click **Browse Memory** to display the **Select a VI** dialog box and select a VI from the VIs in memory to compare.
5. In the **Compare** section of the **Compare VI Hierarchies** dialog box, enable the comparison options you want to include.

6. Click **Compare Hierarchies** to compare VIs in the hierarchies. The results of the comparison appear in the **VIs in Hierarchies** and **Description** listboxes.

**NOTE**

Press the <Ctrl-.> keys to abort comparisons of large VI hierarchies.

7. (Optional) Click **Create Report** to generate a report of the differences between the VI hierarchies.
8. In the **Create Report** dialog box, select a report type and specify a filepath to save the report as:
 - A web page .html
 - An XML document .xml
 - A text file .txt
 - A Microsoft Word document .doc (Windows)

**NOTE**

The file path for saving the comparison report must be a local directory instead of a network path.

9. Click a VI in the **VIs in Hierarchies** listbox to display a description of the hierarchical differences in the **Description** listbox.
10. Double-click a VI in the **VIs in Hierarchies** listbox to open the VIs in the two hierarchies and view the differences.
11. (Optional) You can also click **Show Differences**.
12. Click **Done** to close the **Compare VI Hierarchies** dialog box.
13. (Optional) Click **Read Hierarchies** to categorize the VIs in the hierarchy. VIs that need to be compared are categorized as not compared. You can selectively compare individual VIs by double-clicking an item or clicking **Show Differences**.

Showing Differences between Two Compared VIs

Use the Differences dialog box to show the differences of two *compared VIs*. After you compare two VIs, click **Compare** in the **Compare VIs** dialog box to open the **Differences** dialog box.

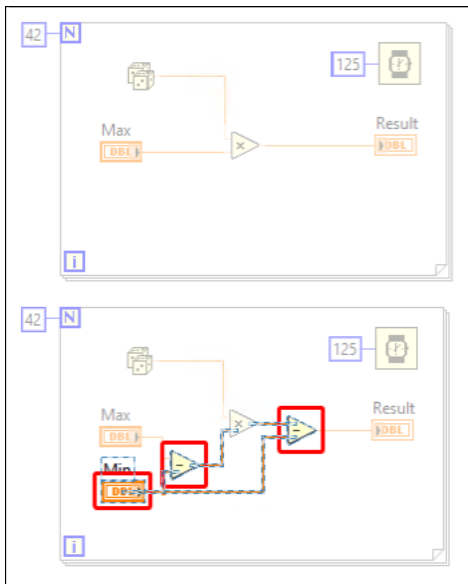
You also can access the **Differences** dialog box by clicking the **Show Differences** button in the Compare VI Hierarchies dialog box.

The **Differences** dialog box lists out the differences and details of each selected difference. When you open the dialog box, the first difference in the list is selected and highlighted by default. Use the left and right arrow keys to move between the differences list and the details list. Use the up and down arrow keys to move through the differences and details. If a difference or detail involves only front panel or block diagram cosmetic changes, it appears italicized in the list.

Complete the following steps to selectively find differences between two compared VIs.

1. Click an item in the differences list or the details list to highlight it. A checkmark indicates that you have viewed an item.
2. Use the **Include:** checkboxes to filter VI attribute, front panel, front panel position/size, block diagram, and block diagram cosmetic differences.
3. Click **More commands** and select **Tile Left and Right** or **Tile Up and Down** to tile the windows of the two VIs.
4. To swap the positions of the two VIs, select **Swap VI Positions** in the **More commands** menu.
5. If you make edits after comparing your VIs, some of the differences might no longer exist. Click **Update** to compare the two VIs again and update the differences list.

Click **More commands** and select **Highlight Differences** to display a red rectangle around the object or objects that have changed. When you highlight a difference, objects that are part of the difference are selected. The following illustration shows an example of a block diagram difference. In the illustration, the highlighted objects have been inserted into the second VI.



VI Comparison Options

You can selectively find front panel and block diagram differences when comparing VIs or VI hierarchies. For the block diagram, you can ignore cosmetic differences. A cosmetic change is any change that does not affect the execution of the block diagram. For the front panel, you can ignore position and size changes in objects. Position and size changes include movement of an object from front to back and vice versa.

You also can identify differences in VI attributes, such as settings in the VI Properties dialog box.



RELATED INFORMATION

[Showing Differences between Two Compared VIs](#) on page 965

Comparing VIs using the LabVIEW Command Line Interface

You can use the LabVIEW Command Line Interface (CLI) to programmatically compare two VIs and generate a comparison report listing the differences between the files.

The LabVIEW CLI `CreateComparisonReport` command compares two specified VIs and outputs an HTML, XML, Word, or text report containing the results of the comparison operation. Use this command to compare VIs without including VI dependencies, preventing missing dependencies from causing errors during comparison.

The following table lists the available arguments of this operation. Required arguments are in bold.

Argument	Description	Default
-vi1	Name or path of the first VI you are comparing.	—
-vi2	Name or path of the second VI you are comparing.	—
-reportType	Desired file type of the output report. This argument supports the following values: - HTML – produces an HTML format report with external images and style file. - HTMLSingleFile – produces an HTML form report with embedded images and style definition. - MicrosoftWord – produces a report in Microsoft Word format.  NOTE Microsoft Word must be installed on your machine for this option to function. - PlainText – produces a report in plain text. - XML – produces a report in XML format with external images. If this argument is not specified, it is inferred from the value provided for <code>-reportPath</code>.	HTMLSingleFile
-reportPath	Name or path of the output report.	—
-o	Overwrites existing reports and supporting files in the specified report directory if they exist.	—

Argument	Description	Default
-c	Creates a new directory for the specified report path if it does not exist.	—
-nofp	Excludes front panel differences from the comparison report.	—
-nofppos	Excludes front panel position differences from the comparison report.	—
-nobd	Excludes block diagram differences from the comparison report.	—
-nobdcosm	Excludes block diagram cosmetic differences from the comparison report.	—
-noattr	Excludes VI attribute differences from the comparison report.	—
-d	Compares the specified VIs without including dependencies.	—

To run this operation in the CLI for LabVIEW, execute a command with the following syntax:

```
LabVIEWCLI -OperationName CreateComparisonReport -vi1 < name
or path of first VI > -vi2 < name or path of second VI > -
reportType < file type of report > -reportPath < name or path
of output report >
```

```
For example, LabVIEWCLI -OperationName CreateComparisonReport -
vi1 "firstVI.vi" -vi2 "secondVI.vi" -reportType
"HTMLSingleFile" -reportPath "report.html"
```

Configuring Source Control with LVCompare.exe

You can configure a third-party [source control](#) provider to use LVCompare.exe as the default comparison tool. LVCompare.exe is equivalent to the Compare VIs dialog box. Use LVCompare.exe to compare differences between two VIs interactively.



NOTE

LVCompare.exe is available only in the LabVIEW Professional Development System.

Complete the following steps to configure a third-party source control provider to use LVCompare.exe as the default comparison tool.

1. Make sure the computer on which you are running LabVIEW also includes a third-party source control provider. Consult the source control administrator for additional assistance.

2. Direct the source control provider to LVCompare . exe. LabVIEW installs LVCompare . exe in different directories depending on the platform, and directories might vary depending on the configuration of the computer:
 - (Windows) C:\Program Files\National Instruments\Shared\LabVIEW Compare\LVCompare.exe
 - (macOS) /Library/Application Support/National Instruments/LabVIEW Compare/LVCompare.app/Contents/MacOS/LVCompare
 - (Linux) /usr/local/bin/LVCompare
3. Use the following syntax to configure any arguments that LVCompare . exe needs.


```
lvcompare . exe < path to VI 1 > < path to VI 2 > [-lvpath < path to LabVIEW >][-noattr][-nofp][-nofppos][-nobd][-nobdcosm][-nobdpos]
```

Note The path specified for each VI can be absolute or relative to where LVCompare . exe launches from.

The arguments have the following meanings:

- `lvpath` is the path to the version of LabVIEW you want to use for the comparison operation.

Note If you do not specify which version of LabVIEW to use for the comparison operation, LabVIEW uses the currently registered version of LabVIEW. (macOS) LabVIEW uses the latest version of LabVIEW on the computer. (Linux) LabVIEW uses the last installed version of LabVIEW.

- `-noattr` means do not compare VI attributes.
- `-nofp` means do not compare the front panels.
- `-nofppos` means do not compare the size or position of front panel objects.
- `-nobd` means do not compare the block diagrams.
- `-nobdcosm` means do not compare the appearance of block diagram objects, including changes in position or size.
- `-nobdpos` means do not compare the size or position of block diagram objects.

Note This argument was deprecated in LabVIEW 2021.

For example, to configure the third-party source control provider to use LabVIEW 2009 to compare VIs but not compare cosmetic, size, or position differences, use the following command line. "C:\Program Files\National Instruments\Shared\LabVIEW Compare\LVCompare.exe" \$1 \$2 -lvpath "C:\Program Files\National Instruments\LabVIEW 2009\LabVIEW.exe" -

`nobdcosm -nofppos` where \$1 and \$2 are variables the third-party source control provider defines for the paths to the VIs you want to compare.



Note You also can enter this command directly in the command line to launch `LVCompare.exe`.

1. (Optional) Complete any remaining steps to configure the third-party source control provider. For example, some source control providers allow you to specify comparison tools for particular file extensions. If the source control provider allows you to specify a comparison tool, specify `LVCompare.exe` for the `.vi` file extension. Consult the documentation for the source control provider for more information.

Configuring Source Control with LVMerge.exe

You can configure a third-party [source control](#) provider to use `LVMerge.exe` as the default merge application. `LVMerge.exe` is equivalent to the Merge VIs dialog box. Use `LVMerge.exe` to automatically or interactively resolve differences between an original VI and two revisions of the original VI.



NOTE

`LVMerge.exe` is available only with the LabVIEW Professional Development System.

Complete the following steps to configure a third-party source control provider to use `LVMerge.exe` as the default merge application.

1. Make sure the computer on which you are running LabVIEW also includes a third-party source control provider, which you should install and configure to meet the requirements of the current software project. Consult your source control administrator for additional assistance.
2. Direct the source control provider to `LVMerge.exe`. LabVIEW installs `LVMerge.exe` in different locations depending on the platform. The following directories are default installation locations for `LVMerge.exe`, but may vary depending on the configuration of the local machine:
 - (Windows) `C:\Program Files\National Instruments\Shared\LabVIEW Merge\LVMerge.exe`
 - (macOS) `/Library/Application Support/National Instruments/LabVIEW Merge/LVMerge.app/Contents/MacOS/LVMerge`
 - (Linux) `/usr/local/bin/LVMerge`
3. (Optional) Configure any arguments that `LVMerge.exe` needs in the third-party source control provider. The following table lists available arguments.

Specifies path to the version of LabVIEW to use for the merge operation. If you do not supply this argument, the **Merge VIs** dialog box uses the following versions of LabVIEW by default:

- (Windows) Currently registered LabVIEW
- (macOS) Latest installed version of LabVIEW
- (Linux) Last installed LabVIEW

Absolute path to a copy of the VI from where you checked it out **Base VI**.

Absolute path to a copy of the latest revision of the VI in source control **Their VI**.

Absolute path to the VI you edited **Your VI**.

Absolute path to which to save the merged VI.

For example, to specify the version of LabVIEW for the merge operation is LabVIEW 8.6, use the following command line: `LVMerge.exe "C:\Program Files\National Instruments\LabVIEW 8.6\LabVIEW.exe" c:\files\base.vi c:\files\theirs.vi c:\files\yours.vi c:\files\merged.vi`. In the command line, the first path is optional and leads to the location of the LabVIEW executable.

1. (Optional) Complete any remaining steps to configure the third-party source control provider. Consult the documentation for your third-party source control provider for more information.

Merging VIs

Use the **Select VIs to Merge** dialog box to automatically or interactively resolve differences between an original VI and two revisions of the original VI. Given the original VI and two revisions based on the original VI, LabVIEW can resolve and merge changes into a separate merged VI. If differences between the two VIs conflict, you can interactively resolve the changes using the **Merge VIs** dialog box.



NOTE

The **Merge VIs** dialog box is available only with the LabVIEW Professional Development System.

Complete the following steps to automatically or interactively merge two VIs.

1. Because LabVIEW cannot load two VIs with the same name, you must rename the VIs you want to merge.
2. Select **Tools»Merge»Merge VIs** to display the Select VIs to Merge dialog box.
3. Click the **Browse** button next to the **Base VI** text box, navigate to the original VI, and click **OK**.
4. Click the **Browse** button next to the **Their VI** text box, navigate to the first VI you want to merge, and click **OK**.

5. Click the **Browse** button next to the **Your VI** text box, navigate to the second VI you want to merge, and click **OK**.
6. (Optional) Select options from the Merge options section to configure how LabVIEW merges elements of the VIs.
7. Click the **Merge** button. The **Merge front panels** window, **Merge block diagrams** window, a merged VI, and the Merge VIs dialog box appear.
8. By default, LabVIEW collapses each change in the **Unresolved conflicting change(s)** field. Click the expand/contract symbol next to each change to view the details of that change.
9. For each change, select the detail of either **Your VI** or **Their VI** by clicking the description of the change under the **Yours** or **Theirs** subcategories. LabVIEW puts an X next to each unresolved change.



Note Most changes are circled in red on either the **Front panel changes** window, the **Block diagram changes** window, or in the untitled merged VI.

1. Click the **Choose Base**, **Choose Theirs**, or **Choose Yours** button to select the change you want to accept and add to the merged VI. As you resolve each change, a checkmark appears next to the change.
2. After you resolve all changes, click the **Close** button to close the **Merge VIs** dialog box and click the **Save** button to save the merged VI.



Note To merge changes from **Their VI** with changes from **Your VI**, there must be a common intersection in **Base VI**. For example, a tunnel must exist in **Base VI** so that **Merge VIs** knows that the matching tunnels in **Their VI** and **Your VI** also match each other.

You also can configure a third-party [source control](#) provider to [use LVMerge.exe](#) as the default merge application. LVMerge . exe is equivalent to the **Select VIs to Merge** dialog box.

To merge LLBs, you can use the Select LLBs to Merge dialog box to automatically or interactively resolve differences between different versions of an LLB.

Merging LLBs

Use the Select LLBs to Merge dialog box to automatically or interactively resolve differences between an [LLB](#) and two revisions of that original LLB. Given the LLB and two revisions based on the original LLB, LabVIEW can resolve and merge changes into a separate merged LLB automatically. If differences between the two LLBs conflict, you can interactively resolve the changes using the Merge LLBs dialog box.



NOTE

The **Merge LLBs** dialog box is available only with the LabVIEW Professional Development System.

Complete the following steps to automatically or interactively merge two LLBs.

1. Select **Tools»Merge»Merge LLBs** to display the **Select LLBs to Merge** dialog box.
2. Click the **Browse** button next to the **Base LLB** text box, navigate to the original LLB, and click **OK**.
3. Click the **Browse** button next to the **Their LLB** text box, navigate to the first LLB you want to merge with the **Base LLB**, and click **OK**.
4. Click the **Browse** button next to the **Your LLB** text box, navigate to the second LLB you want to merge with the **Base LLB**, and click **OK**.
5. (Optional) Select options from the Merge options section to configure how LabVIEW merges elements of the LLBs.
6. Click the **Merge** button. The **Merge LLBs** dialog box appears. LabVIEW lists all the files found in each LLB in the **File(s)** column. The symbols next to each file under the **Theirs**, **Yours**, and **Merged** columns specify the type of changes, if any, that LabVIEW found in each LLB.



Note If both **Your LLB** and **Their LLB** have changes to the same project library file `.lvlib`, LabVIEW cannot resolve the changes automatically. The recommended way of resolving the changes is to open one revision of the `.lvlib` file and manually edit it to match the other revision.

1. Select a row of changes and click the **Choose Theirs**, **Choose Base**, or **Choose Yours** button to select the change you want to accept and add to the merged LLB. As you resolve each change, a checkmark appears next to the change you selected.
2. If any two LLBs contain different revisions of the same VI, use the **Merge VIs** button to merge the revisions. Click the row containing the VI and click the **Merge VIs** button to open the **Select VIs to Merge** dialog box.
3. *Resolve the differences between the revisions to the VI.*
 - If you merge files, you can highlight the row containing the file and click **View Merged File** to see the file you created.
4. After you resolve all changes, click the **Close** button to close the **Merge LLBs** dialog box and click the **Save** button to save the merged LLB.

You also can configure a third-party [source control](#) provider to [use LVMerge.exe](#) as the default merge application. LVMerge .exe is equivalent to the **Select LLBs to Merge** dialog box.

To merge individual VIs, you can use the **Select VIs to Merge** dialog box to automatically or interactively resolve differences between different versions of a VI.

VI Metrics

Using the VI Metrics Window

The VI Metrics window, available in the LabVIEW Professional Development System, provides a way to measure the complexity of an application similar to the widely used source lines of code (SLOC) metrics for text-based programming languages. With the **VI Metrics** window you can view metrics about VIs, which can be useful in finding areas of a VI that are too complex. You also can use those metrics to establish baselines for estimating future projects.

Remember that any metric, including SLOC, is a crude measurement of complexity. The **VI Metrics** window gives you access to many metrics because you might find that some are more valuable than others in certain cases. For example, you might decide that for user interface VIs you can combine certain metrics to get a better idea of the complexity of a VI. In that case, you can make your own metric by saving the information about a VI and writing VIs to parse the results, combining fields to produce a new measurement of the complexity of a VI.

National Instruments is interested in hearing about combined or alternative metrics you find useful in analyzing VIs. Visit the National Instruments Web site to submit suggestions.

Analyzing VIs with the VI Metrics Window

Complete the following steps to analyze VIs.

1. Open the VI or VIs you want to analyze.
2. Select **Tools»Profile»VI Metrics** to display the VI Metrics window.



NOTE

VI Metrics are available only in the LabVIEW Professional Development System.

1. Use the **Select a VI** pull-down menu to select from the list of VIs whose front panels are open. In the **statistics** table at the bottom of the window, LabVIEW displays the names of the VI and its subVIs and metrics for each VI. For each VI in the selected hierarchy, the window lists the number of nodes that VI contains. This number is a rough metric comparable to the SLOC metric commonly used with text-based programming languages. The number of nodes includes functions, subVIs, structures, front panel object terminals, constants, global and local variables, and Property Nodes. However, LabVIEW counts reading multiple properties with the same Property Node as one node. Place a checkmark in the **User interface** checkbox to determine the total number of properties a VI reads or writes. The number of nodes does not include wires, tunnels, or objects that are subcomponents of structures, such as the iteration terminal of a For Loop or the sequence local terminal of a Stacked Sequence structure.

2. (Optional) Place checkmarks in the **Show statistics for** checkboxes to display additional information related to the complexity of each VI, including *block diagram*, *user interface*, *global and local variable*, *shared library*, and *subVI interface* metrics.
3. (Optional) Click the **Save** button to save the metric information for the VI hierarchy to a text file. The file contains the metrics from visible columns in the **statistics** table. The file also contains the number of user VIs and the number of `labview\vi.lib` VIs. LabVIEW saves the information in tab-delimited format so you can open it in a spreadsheet application or read and parse it using VIs.

Block Diagram Metrics

Block diagram metrics, available in the LabVIEW Professional Development System, appear in the VI Metrics window when you place a checkmark in the **Diagram** checkbox. The following metrics specify the block diagram of a VI and its structures and wires:

- **structures** —Number of For Loops, While Loops, Case structures, Event structures, Flat Sequence structures, Stacked Sequence structures, Conditional Disable structures, and Diagram Disable structures.
- **diagrams** —Number of block diagrams. Each VI has a single top-level block diagram. In addition, it has one subdiagram for each loop or structure.
- **max diag depth** —Deepest nesting level of block diagrams in a VI. If the VI has no structures, it has a depth of 0.
- **diag width (pixels)** —Width of the block diagram in pixels.
- **diag height (pixels)** —Height of the block diagram in pixels.
- **wire sources** —Measures the total number of sources in the VI. Each wire has a single source but can branch to multiple destinations. If a wire enters or exits a structure, LabVIEW considers the tunnel that the wire passes through to be a new source.

Shared Library Metrics

Shared library metrics, available in the LabVIEW Professional Development System, appear in the VI Metrics window when you place a checkmark in the **Shared lib calls** checkbox. The following metric specifies the Call Library Function Nodes on a block diagram:

- **shared lib calls** —Number of Call Library Function Nodes on the block diagram.

Globals/Locals Metrics

Globals/locals metrics, available in the LabVIEW Professional Development System, appear in the VI Metrics window when you place a checkmark in the **Globals/locals** checkbox. The following metrics specify the global and local variables of a VI:

- **Global reads** —Number of reads of global variables on the block diagram.
- **Global writes** —Number of writes to global variables on the block diagram.
- **Local reads** —Number of reads of local variables on the block diagram.
- **Local writes** —Number of writes to local variables on the block diagram.

SubVI Interface Metrics

SubVI interface metrics, available in the LabVIEW Professional Development System, appear in the VI Metrics window when you place a checkmark in the **SubVI interface** checkbox. The following metrics specify the subVIs of a VI and their controls and indicators:

- **Connector inputs** —Number of controls on the VI connector pane.
- **Connector outputs** —Number of indicators on the VI connector pane.

User Interface Metrics

User interface metrics, available in the LabVIEW Professional Development System, appear in the VI Metrics window when you place a checkmark in the **User Interface** checkbox. The following metrics specify the VI user interface:

- **Controls** —Number of top-level front panel controls. A cluster or an array is counted as a single control.
- **Indicators** —Number of top-level front panel indicators. A cluster or an array is counted as a single indicator.
- **Property reads** —Number of property reads by a block diagram. If you read multiple properties with the same Property Node, each property increments this number.
- **Property writes** —Number of property writes by a block diagram. If you write multiple properties with the same Property Node, each property increments this number.

Setting Undo

Complete the following steps to set the number of actions you can undo or redo.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Environment from the **Category** list.
3. Enter a number into the **Maximum undo steps per VI** text box. The maximum number you can enter is 99. The minimum number you can enter is 1.

- Click the **OK** button.

**NOTE**

Each VI records its own undo instances. Therefore, the number you enter for **Maximum undo steps per VI** pertains to each VI. Setting this number low can conserve memory resources.

In some instances, LabVIEW deletes all undo information. VI callers automatically update when the VI changes its front panel. LabVIEW deletes all undo information in caller VIs when you make the following changes:

- *Change a VI connector pane pattern* or change the data type of a control.
- Change the appearance of a strict data type, change the data type or strictness, or change a type definition of a custom control.
- Change the data type or name of a control in a global VI or add or remove a control to a global VI.



Configuring Shortcuts for the Quick Drop Dialog Box

Use the Quick Drop Configuration dialog box to specify a shortcut that represents an object. You can assign a shortcut name to an object so that you do not have to enter the entire name of the object in the Quick Drop dialog box.

Complete the following steps to configure shortcuts for the Quick Drop dialog box.

- Press the <Ctrl-Space> keys or select **View»Quick Drop** to display the **Quick Drop** dialog box. On Chinese keyboards, press the <Ctrl-Shift-Space> keys. (macOS) Press the <Command-Shift-Space> keys. (Linux) Press the <Alt-Space> keys.

**NOTE**

You cannot assign shortcuts to LabVIEW project items using the **Quick Drop Configuration** dialog box.

- Click the **Configure** button in the **Quick Drop** dialog box to display the **Quick Drop Configuration** dialog box. LabVIEW displays the **Front Panel** or **Block Diagram** page of the dialog box according to the window from which you access the dialog box.
- Type a shortcut in the **Panel Shortcut** or **Diagram Shortcut** text box. For example, enter cs in the **Diagram Shortcut** text box if you want to assign that shortcut to an object that you select, such as a Case structure.

Note You cannot use modifier keys, such as the <Ctrl> and <Shift> keys, to create shortcuts for the **Quick Drop** dialog box.

3. Select an object from the **Panel Shortcut Object** or **Diagram Shortcut Object** pull-down menu.
4. Click the **Add** button to assign the shortcut to the object. The **Front Panel Shortcuts** or **Block Diagram Shortcuts** table lists all the shortcuts you assign.
5. (Optional) If you want to remove a shortcut, select a shortcut from the table and click the **Remove Selected Shortcut(s)** button.
6. Click the **OK** button to close the dialog box.

To use the shortcuts you assigned, press the <Ctrl–Space> keys or select **View»Quick Drop** to display the **Quick Drop** dialog box. On Chinese keyboards, press the <Ctrl–Shift–Space> keys. (macOS) Press the <Command–Shift–Space> keys. (Linux) Press the <Alt–Space> keys. Enter a shortcut that you assigned to an object. The shortcut and the name of the object, in parentheses, appear in the text box. For example, if you assigned the shortcut cs to a Case structure, enter cs in the text box. The text box displays cs (Case Structure). Click the block diagram to place the Case structure on the block diagram.



Note If you want to transfer the shortcuts you configured for the **Quick Drop** dialog box to another installation of LabVIEW that supports the **Quick Drop** dialog box, copy QuickDropPanelShortcuts and QuickDropDiagramShortcuts from the existing *LabVIEW configuration file* and place them in the other LabVIEW configuration file.

Managing Unfinished Tasks

To manage a list of tasks, or to-do items, you can use hashtags (#) in block diagram labels and comments only. A hashtag followed by text is called a bookmark. When you add a hashtag to a block diagram comment, LabVIEW detects a bookmark and bolds the bookmark tag to distinguish it from the rest of the text, such as **#rewrite** algorithm. Use bookmarks to identify incomplete code that needs further development. To find all bookmarks in your application, select **View»Bookmark Manager** to display the Bookmark Manager window.



NOTE

You cannot use bookmarks in control or indicator labels.

Creating VIs from Templates and Sample Projects

Select **File»Create Project** to display the Create Project dialog box, which contains the following customizable templates and sample projects:

- **Simple State Machine** —A template that defines sections of code and the order in which they execute.
- **Finite Measurement** —A sample project that uses the Simple State Machine template to acquire a single measurement.
- **Queued Message Handler** —A template that monitors a user interface while allowing you to perform other tasks, such as data acquisition or logging.

- **Continuous Measurement and Logging** —A sample project that uses the Queued Message Handler template to acquire measurements continuously and log them to disk.
- **Actor Framework** —A template for an application that consists of multiple independent tasks that communicate with one another.
- **Feedback Evaporative Cooler** —A sample project that uses the Actor Framework template to cool a water tank, where the cooler is represented by either simulated or real hardware, controlled either manually or automatically, and viewed with one of two user interfaces.
- **Touch Panel Project** —A template that defines an application for a touch panel target running the Windows Embedded Standard operating system. This template includes a touch panel VI that runs on a touch panel target and a VI that runs on the host computer. The VI running on the host computer uses the Queued Message Handler template.



NOTE

LabVIEW add-ons may install additional templates and sample projects.

Templates show how to implement commonly-used design patterns in LabVIEW code, whereas sample projects show how to use a template for a specific application. Each template and sample project contains the project libraries, functions, structures, and user interfaces necessary to get started developing a scalable, robust application.

Refer to ni.com for more information about application design patterns.

Creating Template VIs

You can create custom template VIs to avoid adding the same components to the block diagram or the front panel each time you want to perform a similar operation.

Complete the following steps to create a new template based on an existing VI template.

1. Select **File»New** to display the New dialog box.
2. From the **Create New** list, select a template VI from the From Template directory similar to the one you want to create.
3. Click the **OK** button. The front panel and block diagram of the template VI appear.



NOTE

When you create a VI from a template, LabVIEW does not apply the options you select for new VIs in the Options dialog box.

4. Edit the VI.

Tip You can add templates as subVIs to a template VI. To add a template as a subVI, open the VI template and drag the VI icon in the upper right corner of the front panel or block diagram window to the block diagram of the top-level template VI. When you create a VI from a template that contains template subVIs and save the VI, LabVIEW prompts you to save each template subVI as a VI.

1. Save the VI as a template.

1. Select **File»Save As** to display a file dialog box.

Note If you have already saved the template VI, selecting **File»Save As** displays the Save As dialog box instead. Select the **Copy** and **Substitute copy for original** options and click the **Continue** button to display the file dialog box.

2. (Optional) Navigate to the `labview\templates` directory if you want the template VI to appear in the New dialog box with the template VIs LabVIEW installs. To set the block diagram preview, save an image of the block diagram as a `.png` file in the `labview\templates` directory. Append `d` to the name of the VI to create the `.png` filename. For example, if you want to create the block diagram preview of `Test.vi`, save an image of the block diagram as `Testd.png` in the `labview\templates` directory. This block diagram preview and the *VI description* appear in the **Description** section when the VI template is selected in the **Create New** list.

Caution Do **not** save your own VIs and other files in *.llb files installed with LabVIEW*.

3. (Windows) Select **Template VIs** from the **Save as type** pull-down menu. (macOS and Linux) Include the template VI file extension `.vit`.
4. Enter the name of the template VI and click the **OK** button.

You also can change the `.vi` extension to a `.vit` extension in the file system to convert an existing VI to a template VI.

Separating Compiled Code from VIs and Other File Types

Separate compiled code from VIs, thereby creating source-only VIs, for the following reasons:

- **To simplify source control** —If you separate compiled code from all files in a VI hierarchy or LabVIEW project, changes to one file do not produce unsaved changes elsewhere in the hierarchy or project. This behavior allows you to check out only the VIs you want to modify instead of all VIs that call the changed VI.
- **To prepare VIs in source control to be upgraded to a new version of LabVIEW** — You can upgrade source-only VIs to a new version of LabVIEW without checking out the VIs from source control.
- **To improve load time for VIs** —LabVIEW can load source-only VIs more quickly than regular VIs. To maximize this benefit, separate compiled code from all the files in a VI hierarchy or project.

Do not separate compiled code from VIs that you intend to load or run using the LabVIEW Run-Time Engine. The Run-Time Engine cannot run these VIs because it has no access to the compiled object cache that stores the separate compiled code. If you intend for VI hierarchies that you check into source control to run on the Run-Time Engine, build a source distribution with the VIs.

From Which Files Do You Want to Separate Compiled Code?

- **All new files by default** Place a checkmark in the **Separate compiled code from new files** checkbox on the Environment page of the Options dialog box.
- **All new project items in a single project by default** Place a checkmark in the **Separate compiled code from new project items** checkbox on the Project page of the Project Properties dialog box.



NOTE

This option does not separate compiled code from pre-existing files, even if you add the files to the project after you select this option.

- **Existing project items in a single project** Click the **Mark Existing Items** button on the Project page of the Project Properties dialog box. You can then select which items in the project you want to have separate compiled code.
- **A single VI** Place a checkmark in the **Separate compiled code from source file** checkbox on the General page of the VI Properties dialog box. You can also use the Contains Compiled Code property of a VI to separate compiled code from a single VI programmatically.
- **A single project library, LabVIEW class, or XControl** Place a checkmark in the **Separate compiled code from source file** checkbox on the properties page of the file. You can also use the Contains Compiled Code property of a project library to programmatically store the code of the project library in the object cache.

Note LabVIEW does not actually separate compiled code from these file types. Instead, when you select this option, LabVIEW stores the code for each of these source-only files in the object cache, obtaining the same benefits as source-only VIs.



RELATED INFORMATION

[Facilitating Source Control by Separating Compiled Code from VIs and Other File Types](#) on page 388

[Using the LabVIEW Run-Time Engine](#) on page 397

[Compiled Object Cache](#) on page 146

[Developing and Distributing an Application](#) on page 1497

Asynchronously Calling VIs

Instructions and considerations for asynchronously calling VIs, including the following common usage patterns:

- **Call-and-collect:** Asynchronously calling a VI and collecting the results
- **Call-and-forget:** Asynchronously calling a VI without collecting the results

- Asynchronously calling multiple instances of a VI for parallel execution



RELATED INFORMATION

- [Asynchronously Calling VIs](#) on page 981
- [Asynchronously Calling a VI and Collecting the Results](#) on page 982
- [Asynchronously Calling a VI without Collecting the Results](#) on page 984
- [Asynchronously Calling Multiple Instances of a VI for Parallel Execution](#) on page 985

Asynchronously Calling a VI and Collecting the Results

When you open a reference for a VI that you want to call asynchronously, you must decide whether you want to collect the results of the VI later. This topic describes how to start an asynchronous call-and-collect VI call and collect the results later. Other programming languages describe this usage model as fork-and-join. Alternatively, you can [call a VI without collecting its results](#).

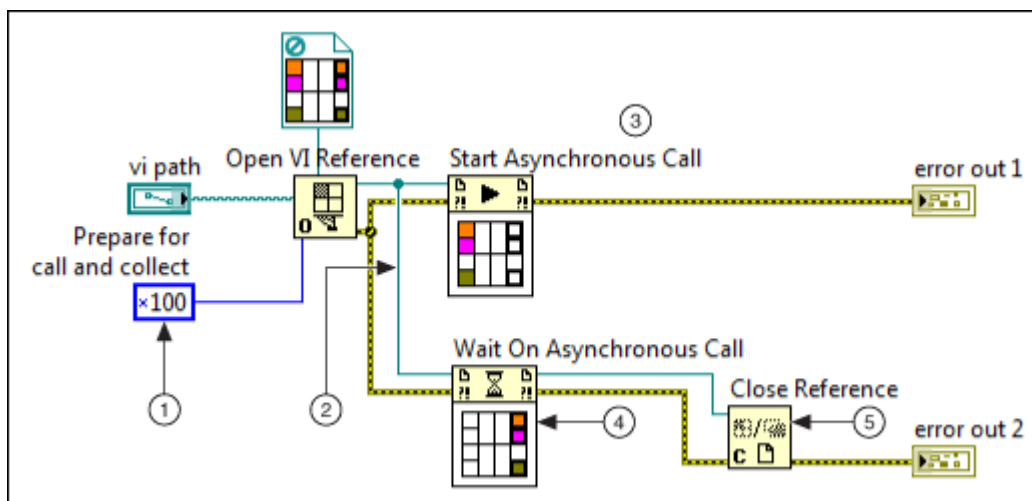
What to Use

Use the following block diagram objects to dynamically start an asynchronous call to a VI and then collect the outputs of the VI at a later time:

<i>Strictly Typed VI Reference</i>	Open VI Reference	Start Asynchronous Call	Wait On Asynchronous Call	Close Reference
 <i>Create</i>				

What to Do

Create the following block diagram to asynchronously call a VI and then collect the results at a later time.



The following list describes important details about the previous diagram.



NOTE

The *LabVIEW Help* displays each of the option flags for the Open VI Reference function in hexadecimal representation. Therefore, always *change the display format* of the Numeric Constant input to hexadecimal to ensure that you are passing the correct value.

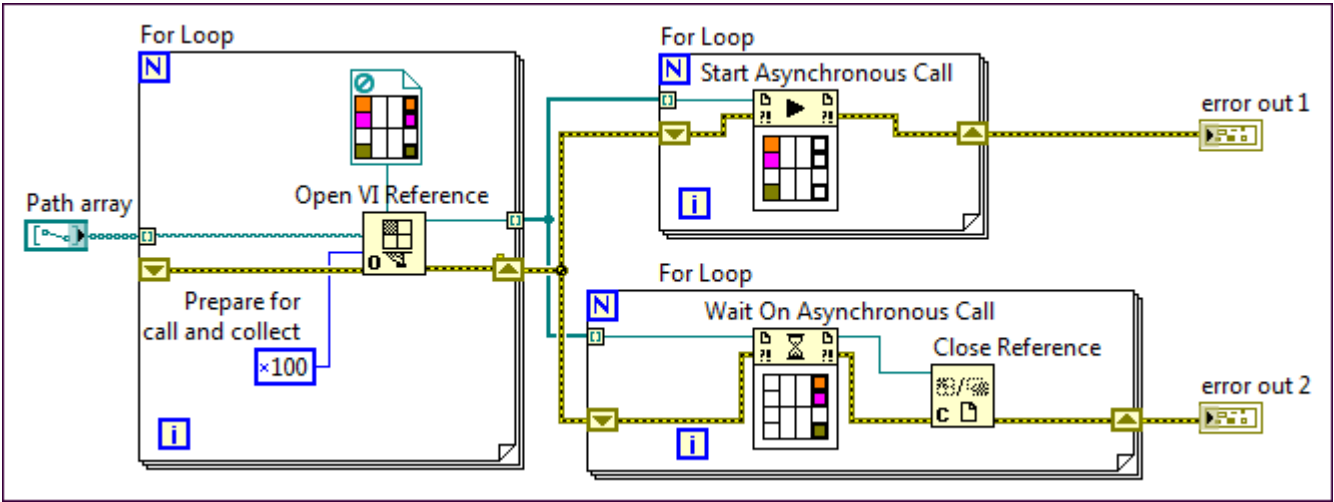
Caveats and Recommendations

- **Aborting VIs** —Consider the following ramifications of aborting the VIs in this usage pattern:
 - If you abort the VI that opened the call-and-collect VI reference, LabVIEW automatically closes the reference and aborts all running instances of the VI.
 - If you abort a specific instance of a call-and-collect VI reference, exactly one Wait On Asynchronous Call node returns an error indicating that the called VI aborted. All other Wait On Asynchronous Call nodes that are waiting on calls to the same VI reference continue waiting.
 - If you call the Abort method on a call-and-collect VI reference, LabVIEW aborts all currently running instances of the VI. A corresponding number of Wait On Asynchronous Call nodes return errors.
- When the VI that opened the call-and-collect VI reference finishes executing, LabVIEW automatically closes the reference and aborts all running instances of the VI.
- If you start multiple asynchronous calls to the same VI reference, you cannot directly identify which VI call returns its outputs to a specific Wait On Asynchronous Call node. *Assign an identifier to a VI call* if you need to know which instance of the VI returned.

Variation: Collecting Outputs from Multiple Calls to Unique VI References

The following block diagram provides a template for calling multiple VIs asynchronously and then collecting their outputs. Because each asynchronous call is associated with a

unique VI reference, each Wait On Asynchronous Call node waits for a specific VI call to finish. This template is particularly useful for creating user interfaces with subpanels whose contents represent the front panels of separate subVIs.



Examples

Refer to the Asynchronous Call and Collect VI in the labview\examples \Application Control\VI Server\Asynchronous Call By Reference directory for an example of asynchronously calling a VI and collecting its results later.

Asynchronously Calling a VI without Collecting the Results

When you open a reference for a VI that you want to call asynchronously, you must decide whether you want to collect the results of the VI later. This topic describes how to start an asynchronous call-and-forget VI call without tracking when or what values the VI returns. Other programming languages describe this usage model as fork-and-forget. Alternatively, you can *call a VI and collect its results later*.

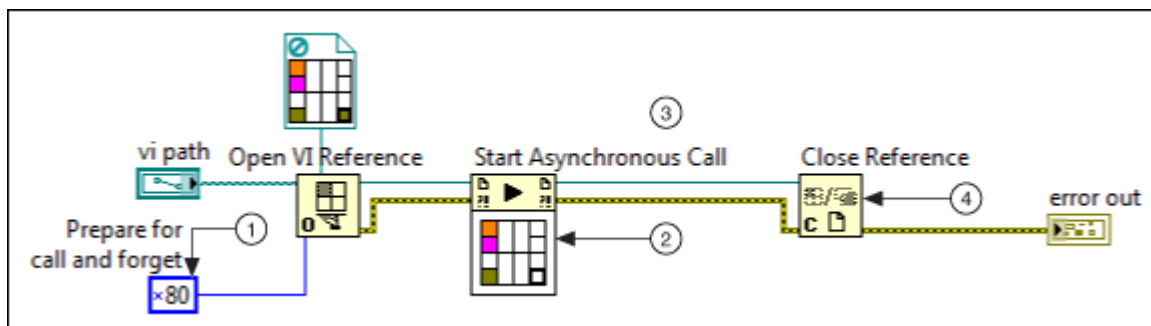
What to Use

Use the following block diagram objects to asynchronously call a VI without collecting its results:

<i>Strictly Typed VI Reference</i>	Open VI Reference	Start Asynchronous Call	Close Reference
 <i>Create</i>			

What to Do

Create the following block diagram to dynamically start an asynchronous call to a VI without collecting the results.



The following list describes important details about the previous diagram.



NOTE

The *LabVIEW Help* displays each of the option flags for the Open VI Reference function in hexadecimal representation. Therefore, always *change the display format* of the Numeric Constant input to hexadecimal to ensure that you are passing the correct value.

Caveats and Recommendations

- Do not call VIs that never terminate. After you start a call with a call-and-forget VI reference, LabVIEW does not abort the target VI when the calling VI finishes executing. Instead, the target VI executes to completion regardless of whether the calling VI is still running. Therefore, if the target VI contains an infinite loop or otherwise cannot finish executing, it remains in memory until you close LabVIEW and may negatively impact performance.
- If you call the Abort method on a call-and-forget VI reference, LabVIEW aborts all currently running instances of the VI.

Examples

Refer to the Asynchronous Call and Forget VI in the `labview\examples\Application Control\VI Server\Asynchronous Call By Reference` directory for an example of asynchronously calling a VI without collecting its results.

Asynchronously Calling Multiple Instances of a VI for Parallel Execution

By default, when you start multiple asynchronous calls to the same VI, LabVIEW serializes the calls, forcing them to run one at a time and leading to a longer cumulative execution

time. Use the following procedure to execute the calls in parallel, preventing any one call from delaying the execution of another call.

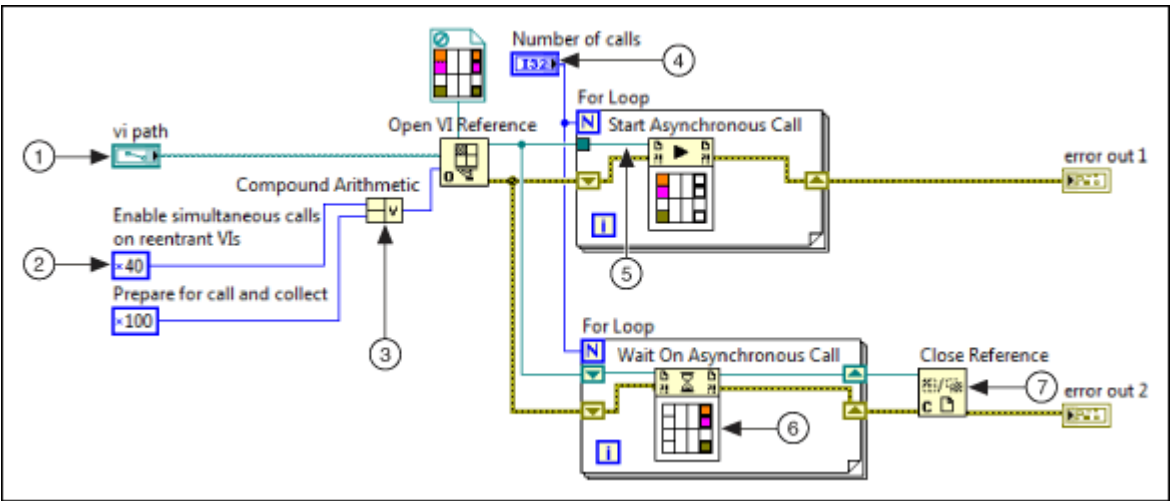
What to Use

Use the following block diagram objects to dynamically start an asynchronous call to a *reentrant VI* and then collect the outputs of the VI at a later time:

Strictly Typed VI Reference	Open VI Reference	Start Asynchronous Call	Wait On Asynchronous Call	Close Reference
				
Create				

What to Do

Create the following block diagram to start multiple asynchronous calls to a reentrant VI and then collect the results.



The following list describes important details about the previous diagram.



NOTE
The *LabVIEW Help* displays each of the option flags for the Open VI Reference function in hexadecimal representation. Therefore, always *change the display format* of the Numeric Constant input to hexadecimal to ensure that you are passing the correct value.

**NOTE**

If you want to *ignore the results of the asynchronous calls*, omit the Wait On Asynchronous Call node and include the 0x80 option flag instead of the 0x100 option flag.

Caveats and Recommendations

VI Server properties and methods cannot modify parallel instances of an asynchronous VI call. If you call a VI Server property or method on a 0x40 VI reference, the property or method cannot modify the VI clone that the Start Asynchronous Call node actually calls. Instead, the property or method affects the original target VI. To apply VI Server properties or methods to the VI clone that the Start Asynchronous Call node actually calls, call the property or method within the target VI itself.

Examples

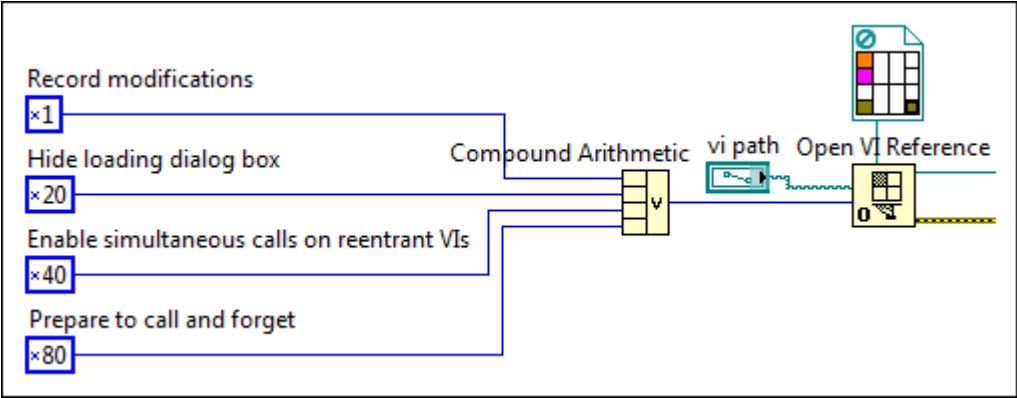
Refer to the Asynchronous Call and Collect (Using Option 0x40) VI in the labview\examples\Application Control\VI Server\Asynchronous Call By Reference directory for an example of asynchronously calling parallel instances of a VI to reduce execution time.

Refer to the Benchmarking Asynchronous Calls VI in the labview\examples\Application Control\VI Server\Asynchronous Call By Reference directory for a comparison of several synchronous and asynchronous approaches to solving the same task.

Combining Option Flags into a Single Option Value

The **options** input of the Open VI Reference function accepts a combination of option flags that each specify a particular characteristic or behavior of the reference output. Each option flag is represented by a unique hexadecimal number. Because the option flags have a numeric representation, you can combine several option flags into one number to specify multiple characteristics for the returned VI reference.

To combine multiple option flags into one value for the **options** input of the Open VI Reference function, create the following block diagram using your desired option flags. Refer to the procedure for details.



Complete the following steps to combine multiple option flags into one value for the options input.

1. Determine which characteristics you want to assign to the reference output of the Open VI Reference function.

Description
Record modifications
Open templates for editing
Prompt user to save changes
Prepare for reentrant run
Prompt to find missing subVIs
Hide loading dialog box
Enable simultaneous calls on reentrant VIs
Prepare for call-and-forget
Prepare for call-and-collect

Refer to the complete options table for more details about specific option flags.

2. Add one Numeric Constant to the block diagram for each desired characteristic.

 Find on the palette.

3. Right-click each Numeric Constant and select **Visible Items»Radix** from the shortcut menu.
4. Click the radix and select **Hex** from the shortcut menu.
5. Set the value of each Numeric Constant to the option flag that corresponds to a desired characteristic.
6. Add a Compound Arithmetic function to the block diagram.

 Find on the palette.

7. Set the arithmetic mode to **OR** by right-clicking the node and selecting **Change Mode»OR** from the shortcut menu.
8. Expand the Compound Arithmetic function to display enough input terminals for each option flag by clicking and dragging one of the resizing handles.
9. Wire each Numeric Constant to an input terminal of the Compound Arithmetic function.
10. Wire the output of the Compound Arithmetic function to the **options** input of the Open VI Reference function.

Configuring the VI Server

Complete the following steps to configure the VI Server.

1. Select **Tools»Options** to display the Options dialog box.
2. (Optional) The default VI Server configuration is suitable for most applications. Select VI Server from the **Category** list if you need to change this default configuration. The default configuration allows ActiveX to call LabVIEW, but does not allow TCP/IP as a protocol. To use TCP/IP with other LabVIEW applications, place a checkmark in the **TCP/IP** checkbox and configure the **Machine Access** settings.
3. If you want to control browser access to the VIs or specify which VIs remote applications can control, configure the **Machine Access** and **Exported VIs** options.
4. Click the **OK** button to close the **Options** dialog box and save the changes.



NOTE

If a target in a [LabVIEW project](#) supports the VI Server, you can configure the VI Server for the target using the **Properties** dialog box for the target. For example, use the My Computer Properties dialog box to configure the VI Server for the My Computer target.

Configuring VI Calls

You can use the VI Call Configuration dialog box to configure when to load a subVI.

Complete the following steps to configure a VI call.

1. Place a VI on the block diagram.
2. Right-click the VI and select **Call Setup** from the shortcut menu to display the **VI Call Configuration** dialog box.



NOTE

This item is available only for VIs open in an [application instance](#) that supports VI Server calls—not for functions, Express VIs, polymorphic VIs, or VIs open in an application instance that does not support VI Server calls, such as FPGA targets.

1. Select one of the following options to configure when the subVI loads:
 - **Load with callers** —Loads the subVI at the same time the caller VI loads. This behavior is identical to how a subVI loads.
 - **Reload for each call** —Loads the subVI when the caller VI makes the call to the subVI if the subVI is not already in memory. This option does not guarantee that the subVI stays in memory after the call.
 - **Load and retain on first call** —Loads the subVI only the first time the caller VI makes the call to the subVI. If you have a large caller VI, you can save load time and memory by selecting the **Load and retain on first call** option in the dialog box. When you select this option, the subVI does not load until the caller VI needs it. If you select the **Reload for each call** option or the **Load and retain on first call** option, LabVIEW continues to load the subVI if the block diagram of the caller VI is in memory. If you open the caller VI in edit mode, LabVIEW loads the block diagram of the caller VI and therefore loads the subVI as well. To avoid loading the subVI, do not open the block diagram of the caller VI. Call the caller VI as a subVI, and close the front panel and block diagram window of the caller VI.



Note Selecting **Reload for each call** or **Load and retain on first call** replaces the subVI with a Call by Reference node that is configured to call the subVI. Symbols on the **Call by Reference** node correspond to the mode that you select. If you choose either of these options you will no longer be able to find the subVI. These options remove the static link to the subVI.

1. Click the **OK** button to save the calling configuration.

Controlling Front Panel Objects Programmatically from a SubVI

Complete the following steps to pass a control reference to a subVI and use it to modify properties or methods of the referenced control or indicator.

1. Right-click the front panel control or indicator of the main VI to which you want to create a reference and select **Create»Reference** from the shortcut menu. LabVIEW attaches a strictly typed control reference to the mouse cursor. Drop the reference on the block diagram of the main VI. You also can create a control reference by selecting a **control refnum** on the **Controls** palette, adding it to the front panel of the subVI, and adding an object of the same type as the referenced object to the front panel control refnum shell. This creates a non-strictly typed control reference.
2. Open the subVI, or create a new VI, to which you are passing the control reference. From the block diagram of the main VI, drag the control reference to the front panel of the subVI to create a control refnum control for the control reference on the subVI

front panel. A corresponding control reference terminal appears on the subVI block diagram.



NOTE

If you drag the control reference to the block diagram, you only copy the control and control reference rather than create a control refnum control.

1. Complete the following steps to modify properties or methods of the front panel object associated with the control reference by creating a Property Node or Invoke Node.
 1. Right-click the control reference on the block diagram of the VI.
 2. Select **Create»Property for (type name)** or **Create»Method for (type name)** and select a property or method from the shortcut menus.
 3. Click to place the node on the block diagram of the subVI.
 4. Wire the control reference to the **reference** terminal of the Property Node or Invoke Node. The Property Node or Invoke Node references the control in the main VI.

You also can select a Property Node or Invoke Node on the **Functions** palette and place it on the block diagram. Right-click the node and select a property or method from the **Properties** or **Methods** shortcut menus.

2. *Set up the terminals on the connector pane* of the subVI, including the control reference.
3. On the main VI, add the subVI to the block diagram and wire the control reference to the control reference terminal of the subVI.

Refer to the Control References VI in the `labview\examples\Application Control\VI Server\Control References` directory for an example of controlling front panel objects programmatically from a subVI.

Creating a Strictly Typed VI Reference to Call VIs Dynamically

Create a *strictly typed VI reference*, a reference with a *strict type definition*, to call a VI dynamically using either the Call By Reference node or the Start Asynchronous Call node. The strictly typed VI reference identifies both the VI you are calling and its connector pane.

Complete the following steps to create a strictly typed VI reference to use with the Call By Reference node or the Start Asynchronous Call node.

1. Add the Open VI Reference function to the block diagram.
2. Wire the path to the VI you want to use to create the strictly typed reference to the **vi path** input of the Open VI Reference function.
3. Right-click the **type specifier VI Refnum** input of the Open VI Reference function and select **Create»Constant** from the shortcut menu.

4. Right-click the **type specifier VI Refnum** constant, select **Select VI Server Class»Browse**, and select a VI from the file dialog box that appears. You also can select the VI icon in the upper right corner of a front panel window or block diagram window and drag the icon to the **type specifier VI Refnum** constant. After you select a VI, a strikeout icon  indicating that the reference is strictly typed appears in the upper left corner of the reference.

**NOTE**

The connector pane of the VI you select must match the connector pane of the VI whose path you wired to the **vi path** input of the Open VI Reference function. You can use the same VI for both inputs.

Now the **vi reference** output of the Open VI Reference function is a strictly typed VI reference.

Creating a VI Server Application

Use the LabVIEW VI Server to programmatically control front panel objects, VIs, and LabVIEW, and to dynamically load, edit, and run VIs on a computer or remotely across a network. You can control browser access to the VIs and configure which VIs remote applications can control.

Complete the following steps to create a VI Server application.

1. Configure the VI Server to allow the TCP/IP protocol.
2. Use the Open Application Reference function to open a reference to a local or remote application instance.

**NOTE**

If you have multiple application instances open simultaneously, such as if you are working with a LabVIEW project or targets of a LabVIEW project, multiple VI Servers can be listening on different ports. Open an application reference to a specific application instance by stating the machine name and the port or service name.

1. Use the Open VI Reference function to open a reference to a VI on the local or remote computer that already exists in memory for the application instance, or to dynamically load a VI from disk. However, be aware that if the input to **vi path** is a file path, this function waits until the user interface is idle to load the VI from disk.
2. Use the Property Node to get or set properties or the Invoke Node to invoke methods.
 - Property Node
 - Invoke Node

You also can use a Call By Reference node to call a dynamically loaded VI.

3. Use the Close Reference function to close any open references.

Refer to `labview\examples\viserver` for examples of using the VI Server.



Find related examples



RELATED INFORMATION

[Controlling Front Panel Objects Programmatically from a SubVI](#) on page 990

[Allowing and Denying Web Browser Access](#) on page 1570

[Specifying Which VIs You Can Control Remotely](#) on page 994

[Configuring the VI Server](#) on page 989

[Working with Application Instances](#) on page 382

[Managing a Project in LabVIEW](#) on page 365

[Dynamically Loading and Calling VIs](#) on page 245

Creating Control Reference SubVIs from Property Nodes and Invoke Nodes

Complete the following steps to create a subVI that includes a control reference directly from a Property Node or Invoke Node.

1. Right-click the front panel object or its block diagram terminal for which you want a control reference subVI, select **Create»Property Node** or **Create»Invoke Node**, and select a property or method from the shortcut menu to [create a Property Node or Invoke Node](#) implicitly linked to the object.
2. Use the Positioning tool to select the new Property Node or Invoke Node and select **Edit»Create SubVI**. LabVIEW creates a control reference to the control on the block diagram of the main VI, creates a subVI that contains a control refnum wired to a Property Node or Invoke Node, and wires the control reference on the main VI to the **control refnum** input terminal of the new subVI.



NOTE

Select several Property Nodes or Invoke Nodes at once to create a subVI with multiple control refnums and Property Nodes or Invoke Nodes.

1. Double-click the subVI to open it. Modify the Property Node or Invoke Node and other elements of the subVI as you do in any other VI.

Dynamically Calling a VI

Complete the following steps to dynamically call a VI with the Call By Reference node.



NOTE

If you want to call a VI asynchronously, use the Start Asynchronous Call node instead of the Call By Reference node.

1. Add the Open VI Reference function to the block diagram.
2. Use the Open VI Reference function to create a strictly typed VI reference.
3. Add the Call By Reference node to the block diagram.
4. Wire the **vi reference** output of the Open VI Reference function to the **reference** input of the Call By Reference Node.
5. Add the Close Reference function to the block diagram.
6. Wire the **reference out** output of the Call By Reference Node to the **reference** input of the Close Reference function.
7. Wire all error in and error out terminals on the block diagram.



RELATED INFORMATION

[Dynamically Loading and Calling VIs](#) on page 245

[Asynchronously Calling VIs](#) on page 246

[Creating a Strictly Typed VI Reference to Call VIs Dynamically](#) on page 991

[Setting up Error I/O in a VI](#) on page 1436

Specifying Which VIs You Can Control Remotely

You can allow remote applications to control VIs by placing checkmarks in any of the boxes in the **Accessible Server Resources** section in the VI Server page of the Options dialog box. Use the **Exported VIs** section to specify which VIs remote applications can control. Create an **Exported VIs** list that allows and denies access to individual VIs, groups of VIs, or directory paths. You also can use the VI Access List property to allow and deny access to VIs programmatically.

When a remote application attempts to control a VI, the VI Server compares the VI name to the entries in the **Exported VIs** list to determine whether it should grant access. If an entry in the **Exported VIs** list matches the VI name, the VI Server permits or denies access to that VI based on how you set up the entry. By default, remote applications can control all VIs.

When you add new **Exported VIs** list entries, edit existing entries, or remove entries, use the correct syntax, order, and wildcards in the entries.



RELATED INFORMATION

[Syntax for the VIs Lists](#) on page 1580

[Adding New Exported VIs Entries](#) on page 1458

[Editing Existing Exported VIs Entries](#) on page 1459

[Removing VIs from the Exported VIs List](#) on page 1459

[Syntax for the VIs Lists](#) on page 1580

VI Scripting

VI Scripting

Requires: VI Scripting

Learn how to complete many common VI Scripting tasks.

- Instructions for enabling VI Scripting
- Instructions and examples for inspecting, editing, and creating target VIs
- A basic tutorial for using VI Scripting to create a blank VI, add objects to the block diagram, reposition those objects, and wire the objects together



RELATED INFORMATION

[Enabling VI Scripting](#) on page 995

[Common VI Scripting Tasks](#) on page 996

[Enabling VI Scripting](#) on page 995

[Common VI Scripting Tasks](#) on page 996

[VI Scripting Tutorial](#) on page 1026

Enabling VI Scripting

To display the VI Scripting palette and use the associated properties and methods, you must enable [VI Scripting](#). Complete the following steps to enable VI Scripting in LabVIEW.

1. Select **Tools»Options** to display the Options dialog box.
2. Select **VI Server** from the **Category** list.

- Place a checkmark in the **Show VI Scripting functions, properties and methods** checkbox.

**NOTE**

By default, the **Display additional VI Scripting information in Context Help window** checkbox is selected. This option displays *VI Scripting connector pane information* that you can use to navigate and edit VIs.

- Click the **OK** button to apply the changes and close the **Options** dialog box.

Common VI Scripting Tasks

When you use VI Scripting to inspect, edit, or create a target VI, you often combine multiple small VI Scripting tasks to accomplish a larger scripting goal. This topic lists some of the most common VI Scripting tasks and links to information and examples about the functions, properties, and methods that you can use to complete those tasks. Some of the more complex tasks link to separate topics.

Obtaining References to VIs and VI Objects

- Obtaining a reference to a target VI
- Obtaining references to objects in an unknown target VI
- Obtaining references to objects in a known VI
- Navigating from a node to a wire
- Navigating from a wire to a node
- Navigating across a structure
- Navigating between the front panel and the block diagram of a VI

Creating VIs and VI Objects

- Creating a new blank VI
- Creating a new VI from a template
- Adding objects to a front panel or block diagram
- Copying an object
- Creating a constant, control, or indicator from a terminal
- Adding, duplicating, and removing frames of a structure with multiple diagrams
- Wiring objects to other objects

Inspecting and Editing VIs and VI Objects

- Configuring a connector pane
- Moving an object
- Replacing an object

- Deleting an object
- Selecting objects to cut, copy, and paste
- Interacting with the label of a node

Miscellaneous

- Undoing actions
- Block diagram cleanup
- Using object tags
- Showing the front panel or block diagram of a VI
- Closing references

Navigating between the Front Panel and the Block Diagram of a VI

Use the following properties to navigate between the front panel and the block diagram of a VI programmatically:

- `Control:Terminal` —Allows you to navigate from a front panel control to its corresponding terminal on the block diagram
- `ControlTerminal:Control` —Allows you navigate from a block diagram control terminal to its corresponding front panel control

Creating a New Blank VI

Use the New VI function to create a new blank VI programmatically.

Refer to the VI Scripting tutorial for detailed instructions about creating and opening a new blank VI programmatically.

Creating a Constant, Control, or Indicator from a Terminal

Use the following methods to create a constant, control, or indicator from a terminal programmatically:

- `Terminal>Create Constant`
- `Terminal>Create Control`
- `Terminal>Create Indicator`

Adding, Duplicating, and Removing Frames of a Structure with Multiple Diagrams

Use the following methods to interact with the frames of a structure with multiple diagrams programmatically:

- MultiFrameStructure:Add Frame
- MultiFrameStructure:Duplicate Frame
- MultiFrameStructure:Remove Frame
- FlatSequence:Add Frame
- FlatSequence:Remove Frame

Wiring Objects to Other Objects

Use the following methods to wire objects to other objects programmatically:

- Terminal:Connect Wire
- Node:Connect Wires

Refer to the Creating New VI From Scratch VI in the `labview\examples\Application Control\VI Scripting\Creating VIs` directory for an example of using the Terminal:Connect Wire method to wire two objects together.

Refer to the VI Scripting tutorial for an example of using the Node:Connect Wires method to wire two objects together.

Moving or Copying an Object

Use the GObject:Move method to move an object programmatically:

Replacing an Object

Use one of the following methods to replace an object programmatically:

- GObject:Replace
- GObject:Replace No Attributes

Refer to the Creating New VI From Template VI in the `labview\examples\Application Control\VI Scripting\Creating VIs` directory for an example of using the Replace method to replace an object.

Deleting an Object

Use the Generic>Delete method to delete an object programmatically.

Refer to the Closing References VI in the `labview\examples\Application Control\VI Scripting\Managing References` directory for an example of using the Delete method to delete a wire.

Selecting Objects to Cut, Copy, and Paste

Use the following properties to access selected objects programmatically:

- GObject:Selected
- Panel:Selection List[]
- TopLevelDiagram:Selection List[]

Use the following methods to edit selections programmatically:

- Panel Methods
- TopLevelDiagram Methods

Interacting with the Label of a Node

Use the Node:Label property to interact with the label of a node programmatically.

Undoing Actions

Use the following methods to undo VI Scripting actions programmatically:

- VI:Transaction:Undo
- VI:Transaction:Redo
- VI:Transaction.Begin Undo
- VI:Transaction.End Undo
- VI:Transaction.Fail

Refer to the Supporting Undo VI in the labview\examples\Application Control\VI Scripting\Supporting Undo directory for an example of using the Transaction methods to undo actions programmatically.

Block Diagram Cleanup

Use the VI:Block Diagram:Clean Up method to clean up the block diagram of a target VI:

Using Object Tags

Use the following methods to get and set object tags programmatically:

- Generic:Tag.Get Tag
- Generic:Tag.Set Tag

Refer to the Tagging VI in the labview\examples\Application Control\VI Scripting\Finding and Modifying Objects directory for an example of using the Tag methods to add tags to objects and find the tagged objects.

Showing the Front Panel or Block Diagram of a VI

Use the following properties and methods to show or position the front panel or block diagram of a target VI programmatically:

- Front Panel:Open
- Front Panel Window:Window Bounds
- Block Diagram Window:Open
- Block Diagram Window:Window Bounds
- Block Diagram Window:Origin

Refer to the Creating New VI From Scratch VI in the labview\examples\Application Control\VI Scripting\Creating VIs directory for an

example of opening and positioning the front panel and block diagram of a newly created VI.



RELATED INFORMATION

[Obtaining a VI Reference](#) on page 1002

[Obtaining References to Objects in an Unknown Target VI Using VI Scripting](#) on page 1005

[Obtaining References to Objects in a Known Target VI Using VI Scripting](#) on page 1003

[Navigating from a Node to a Wire Using VI Scripting](#) on page 1020

[Navigating from a Wire to a Node Using VI Scripting](#) on page 1022

[Navigating across a Structure Using VI Scripting](#) on page 1024

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Creating a New VI from a Template Using VI Scripting](#) on page 1014

[Adding an Object to a VI Using VI Scripting](#) on page 1008

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Configuring a Connector Pane Using VI Scripting](#) on page 1011

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Common VI Scripting Tasks](#) on page 996

[Closing References to VIs and VI Objects Using VI Scripting](#) on page 1016

[Tutorial: Creating and Opening a VI Using VI Scripting](#) on page 1026

[Tutorial: Wiring Objects Using VI Scripting](#) on page 1031

Obtaining a VI Reference

Requires: [VI Scripting](#)

To use VI Scripting to inspect, modify, or create VIs and VI objects, you must usually begin by obtaining a reference to the target VI with which you want to interact. You then can use the reference to obtain references to objects within the target VI or to add new objects to the target VI.

Refer to the following table to identify the best way to obtain a VI reference for your specific use case.

Type of Target VI	How to Obtain a Reference to the Target VI
Pre-existing target VI whose path you know at edit time	Static VI Reference 
Pre-existing target VI that you want to select at run time Target VI that you intend to call with the Call By Reference node	Open VI Reference 
New VI	New VI 
New VI with contents copied from a template VI	Create a new VI from a template.

Caveats and Recommendations

- [Create complicated new VIs from a template](#) when possible —While you can create a new VI programmatically by creating each individual component object one at a time, it is much easier to create a new VI from a template and then modify small sections of the code.
- **Displaying the target VI is optional** —You do not have to display the front panel or block diagram of a target VI in order to inspect or edit the VI programmatically. However, you can use the following properties and methods to display different parts of the target VI:
 - Front Panel:Open
 - Block Diagram Window:Open
- **Close all references that you open** —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and object references](#) for more performance details.

Obtaining a VI Object Reference Using VI Scripting

Requires: [VI Scripting](#)

When you use VI Scripting to inspect, modify, or create front panel or block diagram objects, you must obtain a reference to the object with which you want to interact. You then can wire this reference to Property Nodes or Invoke Nodes to get or set information about the referenced object.

Refer to the following list to identify the best way to obtain a VI reference for your specific use case. The approaches in the list increase in difficulty.

- **Use the Open VI Object Reference function.** The Open VI Object Reference function allows you to search for a specific object in a target VI by its label. Use this approach when you want to [obtain a reference to an object in a target VI with known contents](#), such as a template VI.
- **Use the Traverse for GObjects VI.** The Traverse for GObjects VI allows you to search for all objects in a target VI that are instances of a specific class. Use this approach when you want to [obtain references to objects in an unknown target VI](#). Also use this approach when you want to interact with multiple objects in a target VI that are instances of the same class.
- **Navigate the connections among objects within the target VI.** [Navigating connections among objects](#) within a target VI allows you to use a reference to one object to obtain references to connected objects. Use this approach to obtain a reference to an object that you cannot access directly with one of the previous approaches.

Caveats and Recommendations

- **Close all references that you open.** An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and object references](#) for more performance details.

Obtaining References to Objects in a Known Target VI Using VI Scripting

Requires: [VI Scripting](#)

When you create a new VI from a template, you usually want to make small modifications to the code that LabVIEW copied from the template into the new VI. To make these modifications, you must obtain references to the objects in the new VI that you want to inspect or modify. After you obtain these object references, you can wire them to Property Nodes or Invoke Nodes to get or set information about the referenced objects. The easiest way to obtain references to objects in the new VI is to use the Open VI Object Reference function. However, to use this function, the desired objects in the new VI must have unique labels. If the desired objects do not have unique labels, you must use a different, more complex procedure to [obtain references for these unlabeled or unknown objects](#).

What to Use

This VI Scripting task requires the following objects:

Open VI Object Reference	Class Specifier Constant	Close Reference
		

What to Do

Before you start: Open the Obtaining Known Object Reference VI in the labview \examples\Application Control\VI Scripting\Finding and Modifying Objects directory for an example that illustrates each step of the following procedure.

1. *Obtain a reference to the target VI* that contains the object with which you want to interact.
2. Determine the identifying information of the object whose reference you want to obtain.
 1. Class type—Every front panel or block diagram object that you can use with VI Scripting is a member of a VI Server class, so the reference for a given object is of a corresponding class. Refer to the *anatomies of VIs* to determine the class name of the object you want to access. Use the Class Specifier Constant to indicate the class of object for the Open VI Object Reference function.

Example Details

In the example, the **Edit Me** control is a member of the Numeric class.

2. Owner—Every object on both the front panel and the block diagram has an owner. In the simplest case, the owner of an object is either the front panel or the block diagram of a VI. However, if an object resides within a substructure such as a loop, a Case structure, a sequence structure, or a cluster, that substructure owns the object.

Example Details

In the example, the **Edit Me** control resides on the front panel of the target VI, so the front panel of the target VI owns the control. Thus, the scripting VI uses the VI:Front Panel property to obtain the reference to the front panel of the target VI.



NOTE

The Control and ControlTerminal classes are often confused. A ControlTerminal object is the block diagram representation for the corresponding front panel Control object. Thus, the front panel of the target VI owns the **Edit Me** Control object, and the block diagram of the target VI owns the associated **Edit Me** ControlTerminal object. You can change the numeric representation of a Control object but not of a ControlTerminal object.

3. Label—This is the name assigned to the desired object in the target VI.

Example Details

In the example, the label of the control to change is **Edit Me**.

3. Use an Open VI Object Reference function to obtain a reference to the desired object by wiring the identifying information from step 2 to the function.
4. Get or set information about the referenced object by wiring the **object refnum** output of the Open VI Object Reference function to a Property Node or an Invoke Node.

Example Details

In the example, the scripting VI uses the Numeric:Representation property to set the numeric representation of the **Edit Me** control.

5. Use the Close Reference function to close each open object reference when you are finished using it.

Caveats and Recommendations

- **Display block diagram labels at least once** —The Open VI Object Reference function cannot locate objects whose labels have never been displayed. Because block diagram objects do not display a label by default, you must manually display the label for any block diagram object that you intend to find using the Open VI Object Reference function. To display the label of an object when the label is invisible, right-click the object and select **Visible Items»Label**. After you display the label at least once, the Open VI Object Reference function can find the associated object even if you make the label invisible again.
- **Create unique labels** —LabVIEW does not enforce unique labels for objects on the block diagram. By default, most objects that are members of the same class have the same label. For example, if you add two For Loops to a block diagram, each one has a label of **For Loop**. If you use the Open VI Object Reference function on an object that shares a label with another object, LabVIEW returns a reference to the first object that it finds. Assigning unique labels to each object ensures that LabVIEW returns the desired reference.
- **Close all references that you open** —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and object references](#) for more performance details.

Additional Examples

Refer to the Creating New VI From Template VI in the `labview\examples\Application Control\VI Scripting\Creating VIs` directory for an example of obtaining references in a known VI.

Obtaining References to Objects in an Unknown Target VI Using VI Scripting

Requires: [VI Scripting](#)

You can use VI Scripting to inspect or modify target VIs whose contents are not known until run time. As in any VI Scripting application, you must obtain references to the objects that you want to manipulate in the target VI. However, because you do not know the contents of the target VI until run time, you cannot use the Open VI Object Reference function to locate the objects by their label. Instead, you can use the Traverse for GObjects VI to identify groups of objects that are members of the same class. Then, you can iterate through the returned array of objects to determine which objects you want to manipulate.

What to Use

This VI Scripting task requires the following objects:

Traverse for GObjects	To More Specific Class	Close Reference
		

What to Do

Before you start: Open the Obtaining Unknown Object References VI in the labview \examples\Application Control\VI Scripting\Finding and Modifying Objects directory for an example that illustrates each step of the following procedure.

1. Obtain a reference to the target VI that contains the objects you want to inspect or edit.
- Example Details
- In the example, the Open VI Reference function provides a reference to the VI specified by the **Path to target VI** control. This implementation enables the user to apply the scripting VI to any target VI just by specifying a different VI path. Furthermore, the scripting VI can be converted to a subVI if the user wants to programmatically apply it to a set of VIs.
2. Determine the class and location of the objects whose references you want to obtain.
1. Containing VI—Wire this reference to the **VI Refnum** input of the Traverse for GObjects VI.
2. Location within VI—Within the target VI, objects can be located on a front panel, on a block diagram, in a cluster, or on the diagram of a structure such as a For Loop or Case structure. You must specify which of these locations you want the Traverse for GObjects VI to search by creating a constant from the **Traverse Target** input.

Example Details

In the example, the scripting VI needs to manipulate control terminals, which are block diagram objects. Therefore, the scripting VI specifies BD as the container for the scripting VI to search.



NOTE

If the objects of interest are in a cluster or a diagram of a structure, select **Other** from the enumerated list of possibilities for the **Traverse Target** input. In both cases, wire the reference for the container cluster or diagram to the **Other Refnum** input of the Traverse for GObjects VI.

3. Class name of the object—Refer to the [anatomies of common VI objects](#) to determine the class name of the objects you want to manipulate.

Example Details

In the example, because the scripting VI is obtaining references to control terminals, the scripting VI uses the class name of control terminals, `ControlTerminal`.

3. Use the Traverse for GObjects VI to obtain references to the desired objects by wiring the information from step 2 to the Traverse for GObjects VI.
4. Iterate through the returned array of references to manipulate individual objects.

Example Details

In the example, the scripting VI uses a For Loop to gain access to each element in the **References** array.

5. Use the To More Specific Class function to cast each returned reference to a more specific class. The Traverse for GObjects VI returns an array of GObject references rather than references that are members of the class specified by the **Class Name** input. However, you can safely use the To More Specific Class function to cast the returned references to the class specified by **Class Name**. When you cast a reference to a more specific class, you can call more specific properties or methods on the reference.

Example Details

In the example, the scripting VI casts each returned reference to a `ControlTerminal`. This cast is necessary because the goal of making the control terminal labels visible can be accomplished only by setting the `ControlTerminal:Label.Visible` property to `TRUE`, and the `Label.Visible` property is not available for GObjects. This cast does not produce an error because the Traverse for GObjects VI had searched specifically for `ControlTerminal` objects.

6. Wire the reference to a Property Node or Invoke Node to get or set information about the referenced object.

Example Details

In the example, the scripting VI uses a Property Node to set the `Label.Visible` property of each `ControlTerminal` reference to `TRUE`, thereby making the label of the corresponding control terminal visible.

7. Use the Close Reference function to close each open object reference when you are finished using it.

Caveats and Recommendations

- **Check the class of each reference when necessary** —If you wire the name of a parent class to the Traverse for GObjects VI, you obtain an array of references that share a parent class but may not be members of the same child class. For example, if you use the Traverse for GObjects VI to search for all objects that are members of the Loop class, the VI returns references to both ForLoops and WhileLoops. You can use one of the following two approaches to check the class of a reference:
 - Use the To More Specific Class function to try to cast the reference to a specific class. If the function does not return an error, the reference matches the specific class.
 - Compare the Class Name property of an individual returned reference with the class name that you want to match.
- **Close all references that you open** —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and object references](#) for more performance details.

Additional Examples

Refer to the Using Traverse VI in the labview\examples\Application Control\VI Scripting\Finding and Modifying Objects directory for an example of using the Traverse for GObjects VI.

Adding an Object to a VI Using VI Scripting

Requires: [VI Scripting](#)

You can use VI Scripting to programmatically add any object to the front panel or block diagram of a VI. These objects include:

- Front panel controls, including user-defined custom controls
- Block diagram nodes, including functions, structures, and calls to subVIs
- Elements of clusters, arrays, and structures

What to Use

This VI Scripting task requires the following objects:

New VI Object	Class Specifier Constant	Close Reference
		

What to Do

Before you start: Open the Adding Objects VI in the `labview\examples\Application Control\VI Scripting\Creating Objects` directory for an example that illustrates each step of the following procedure.

1. *Obtain a reference to the target VI.*
2. Determine the defining information of the new object that you want to create.
 - Owner reference—Every object on the front panel or the block diagram has an owner. In the simplest case, the owner of an object is either the front panel or the block diagram of a VI. However, if an object resides within a substructure such as a loop, a Case structure, a sequence structure, or a cluster, that substructure owns the object. The *Additional Examples* section contains examples of adding an object to a loop, an array, and a cluster.

Example Details

In the example, the goal is to create the new Subtract function on the block diagram of the target VI. The scripting VI uses the reference to the target VI as the owner for the new Subtract function even though the block diagram of the target VI is the more precise owner. However, the New VI Object function recognizes when an object such as a Subtract function can appear only on a block diagram and places the new object accordingly.

- Class name—Every front panel or block diagram object that you can create with VI Scripting is a member of a VI Server class. Refer to the *anatomies of VIs* to determine the class name of the object you want to create. Use a class specifier constant to indicate the class of object for the New VI Object function.

Example Details

In the example, the Subtract function is a member of the **Function** class.

- Style—Every native LabVIEW object has a style that further defines exactly what kind of object it is. You can use the Control:Style ID property or the Node:Style property to help identify the style of an object that you want to create.



NOTE

You do not need to specify a style if you are adding a subVI or a user-defined control to the target VI.

Example Details

In the example, the scripting VI is creating a Subtract function, so the style is `Subtract`. Notice that the style of a node is often different than the class of the node.

- Location—You must specify the location of any object that you create. This location is relative to the origin of the owner of the object.

Example Details

In the example, the scripting VI specifies a location of (100, 200) for the Subtract function.

- (Optional) Path to subVI or user-defined custom controls—SubVIs and user-defined custom controls require you to wire a path to the source file to the **path** input of the New VI Object function. Refer to the [Additional Examples](#) section for an example of creating a call to a subVI.
- (Optional) Size of new object—You can specify a size for any object that you can manually resize.

Example Details

In the example, the scripting VI is creating a Subtract function, which is not resizeable.

3. Use a New VI Object function to add the new object to the target VI by wiring the information from step 2 to the function.
4. (Optional) [Programmatically wire the newly created object to existing objects](#) in the target VI.

Refer to the [Additional Examples](#) section for an example of using VI scripting to wire objects.

5. Use the Close Reference function to close each open object reference when you are finished using it.

Caveats and Recommendations

- [Create complicated new VIs from a template](#) when possible —Although you can create a new VI programmatically by creating each component object one at a time, it is much easier to create a new VI from a template and then modify small sections of the code. The Creating New VI From Scratch VI in the [Additional Examples](#) section demonstrates that creating even a simple target VI one object at a time requires extensive scripting code. You can avoid such complicated scripting code by creating a new VI from a template.
- **Create complex controls from a template when possible** —Although you can programmatically create a complex control one element at a time, it is easier to create a custom control manually, add an instance of the control by following the previous procedure, and then modify individual parts of the control as necessary. If you have a reference to a custom control, you also can use the VI>Create from Reference method to create a copy of the control.
- **Close all references that you open** —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and VI object references](#) for more performance details.

Additional Examples

Refer to the Creating New VI From Scratch VI in the `labview\examples\Application Control\VI Scripting\Creating VIs` directory for an example of adding an array control, a numeric indicator, and a call to a subVI to a target VI.

Refer to the Drop Add Function Inside While Loop VI in the `labview\examples\Application Control\VI Scripting\Creating Objects` directory for an example of adding an Add function inside of another object such as a While Loop.

Refer to the Drop Digital Numeric Inside Cluster VI in the `labview\examples\Application Control\VI Scripting\Creating Objects` directory for an example of adding an object to a cluster.

Configuring a Connector Pane Using VI Scripting

Requires: *VI Scripting*

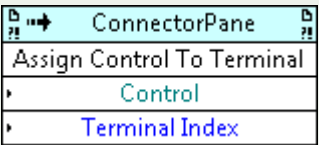
You can use VI Scripting to interact with the connector pane of a VI programmatically to accomplish any of the following tasks:

- Selecting a connector pane pattern for a subVI
- Assigning a control to a terminal on a connector pane
- Navigating across a connector pane
- Defining whether a terminal of a subVI is required, optional, or recommended

This topic highlights the first two tasks. Refer to the Connector Pane methods for more information about interacting with a connector pane programmatically.

What to Use

This VI Scripting task requires the following objects:

VI:Connector Pane:Reference	ConnectorPane:Pattern	ConnectorPane:Assign Control To Terminal	Connector Pane Pattern Reference Example
			

What to Do

Before you start: Open the Add Terminals to Connector Pane VI in the labview \examples\Application Control\VI Scripting\Connector Pane directory for an example that illustrates each step of the following procedure.

1. **Obtain a reference to the VI** whose connector pane you want to inspect or modify.

Example Details

In the example, the scripting VI uses the Open VI Reference function to obtain a reference to a specific VI.

2. Use the VI:Connector Pane:Reference property to obtain a reference to the connector pane of the VI.
3. Determine the connector pane pattern you want the VI to have.

Connector pane patterns are denoted by a pattern index between 4800 and 4835. By default, connector panes use pattern 4815.

You can use one of the following approaches to identify which pattern index to use.

- Refer to the Connector Pane Pattern Reference VI located in the labview \examples\Application Control\VI Scripting\Connector Pane directory for a complete mapping of patterns to their indexes.
- View the Context Help of a VI that has a connector pane pattern that you want to use for the VI that you are modifying. If you **display additional VI scripting information in the Context Help window**, the pattern index appears at the end of the path to the VI.
- Use the ConnectorPane:Pattern property of another VI to programmatically obtain the pattern index of that VI.

Example Details

In the example, the scripting VI enables the user to select the connector pane pattern at run time. Therefore, the scripting VI uses a control to determine the connector pane pattern.



NOTE

Because the **Connector Pane Pattern** control accepts any integer, this example should check the value of that control to ensure that it is within the 4800..4835 range of acceptable pattern indexes. For simplicity, the example omits this step.

4. Use the Connector Pane:Pattern property to set the connector pane to your chosen pattern.
5. **Obtain references to the controls** that you want to assign to connector pane terminals.

Example Details

In the example, the scripting VI uses the Panel:Controls[] property of the front panel of the target VI to obtain references to all front panel controls.

6. Determine how you want to map these controls to the connector pane terminals.
Refer to the [LabVIEW Style Checklist](#) for guidelines about assigning controls and indicators to connector panes that are visible to your users.

Example Details

In the example, the only goal of the scripting VI is to assign all controls to a terminal. Therefore, the scripting VI does not have criteria for how to map the controls to the terminals.

7. Use the ConnectorPane:Assign Control To Terminal method to assign each control to the desired terminal of the connector pane.

To identify the terminal index that corresponds to each terminal in your chosen pattern, refer to the Connector Pane Pattern Reference VI mentioned in step 3.

Example Details

In the example, the scripting VI iterates across the entire Panel:Controls[] array, assigning each control to the next terminal index of the connector pane. This approach does not require knowledge of specific terminal indexes because the scripting goal does not require the controls to be wired to the connector pane in a specific way.



NOTE

Because it is possible in the example for the chosen connector pane pattern to have fewer terminals than there are controls on the front panel, it is also possible for the ConnectorPane:Assign Control To Terminal method to return an error. Refer to the [Caveats and Recommendations](#) section of this topic for information about avoiding this error.

8. Use the Close Reference function to close each open object reference when you are finished using it.

Example Details

In the example, the scripting VI closes the reference to the front panel, the controls, the connector pane, and the target VI.

Caveats and Recommendations

- **Use caution when you change a connector pane pattern that has pre-existing control assignments** —When you replace a connector pane pattern with a different pattern, LabVIEW reassigns the controls that were previously assigned to terminals on the old pattern to terminals on the new pattern. However, the new assignments do not preserve the terminal indexes or relative locations from the old pattern. Therefore, you can rely only on knowledge of terminal assignments that you obtain after the most recent pattern change.
- **Ensure that the pattern of a connector pane has enough terminals to accommodate the terminal indexes that you plan to use** —The Connector Pane:Assign Control to Terminal method returns an error if the pattern of the referenced connector pane does not have enough terminals to accommodate the terminal index that you wire to the method. Use the ConnectorPane:Number of

Connection Terminals property of the relevant connector pane to verify that the connector pane has enough terminals for the terminal index that you want to use.

- **Close all references that you open** —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and VI object references](#) for more performance details.

Creating a New VI from a Template Using VI Scripting

Requires: [VI Scripting](#)

Although you can programmatically create complex VIs one object at a time by using the New VI Object function and the Terminal:Connect Wire method, that approach requires complex and expansive scripting code that is time-consuming to write and difficult to read. It is often easier to write a template VI that closely approximates the VI you want to create. You then can use VI Scripting to create a copy of the template VI and edit small parts of the copied code to achieve your scripting goals.

What to Use

This VI Scripting task requires the following objects:

New VI	Close Reference
	

What to Do

Before you start: Open the Creating New VI From Template VI in the labview \examples\Application Control\VI Scripting\Creating VIs directory for an example that illustrates each step of the following procedure.

1. Design a template VI that approximates the form of the desired target VI as closely as possible.

Include the following components in the template design:

- All objects that you do not want to create programmatically one at a time
- All connections that you do not want to create programmatically from scratch
- Placeholder objects for objects that you want to specify at run time. Placeholder objects allow you data to wire the terminals of the placeholder in the template rather than programmatically creating those connections from scratch in the scripting VI.
- Visible labels on all objects that you want to be able to locate or modify with the scripting VI

Example Details

In the example, the template illustrates a design that includes an array input, a numeric output, a Placeholder subVI, and the wiring required to connect these objects required. At run time, a scripting VI can replace the Placeholder subVI with a subVI that performs the user-specified computation.



NOTE

The placeholder subVI purposely has an identical connector pane pattern to both of the possible replacement subVIs. Matching connector pane patterns enables the scripting VI to easily replace the placeholder subVI without rewiring its terminals. If the placeholder object in a template VI has a different connector pane than the replacement object, LabVIEW might not wire data to the replacement object as desired.

2. Use a New VI function to create a copy of the template VI by wiring the path to the template VI to the function.

You can use a path control, a path constant, or the VI:VI Path property of a reference to your template VI to specify the path of the template VI.

Example Details

In the example, the VI:VI Path property provides the path to the template VI.

3. [Obtain references to the objects that you want to modify](#) in the template VI. The most common way to obtain a reference to an object in a template VI is to use the Open VI Object Reference function.

Example Details

In the example, the scripting VI must replace the Placeholder subVI with a subVI that performs the operation that the user of the scripting VI specifies at run time. Because the Placeholder subVI has a unique label, the Open VI Object Reference function can obtain a reference to that subVI.

4. Inspect or modify the referenced objects to achieve your scripting goal.

Refer to the [list of common scripting tasks](#) for more information about ways to edit a VI using VI Scripting.

Example Details

In the example, the scripting VI uses the Replace method to replace the Placeholder subVI with a subVI capable of performing the user-specified computation task. Based on the value of the **Operation** radio button at run time, a Case structure selects whether to replace the placeholder subVI with an addition subVI or a multiplication subVI.

5. Use the Close Reference function to close each open object reference when you are finished using it.

Example Details

In the example, the scripting VI closes the reference to the replacement subVI as well as the reference to the target VI. However, the scripting VI does not close the reference to the original placeholder subVI because the Replace method invalidates that reference.

Caveats and Recommendations

- **Create an intermediate subVI that gathers and returns the desired references** — For VI scripting applications that make many modifications to a template, you can clarify your code by separating the code that gathers references into a subVI. To create a separate subVI that returns the references you want to modify, perform steps 2-4 in a new subVI rather than in your top-level scripting VI. Then the top-level scripting VI can call the new subVI and make modifications with the returned references.
- **Close all references that you open** —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and object references](#) for more performance details.

Additional Examples

Refer to the Creating New VI From Scratch VI in the labview\examples\Application Control\VI Scripting\Creating VIs directory for an example of creating a new VI one object at a time. Notice the complexity of this example and the number of references required to complete the task without using a template.

Closing References to VIs and VI Objects Using VI Scripting

Requires: [VI Scripting](#)

All VI Scripting tasks require you to open references to the VIs and objects that you want to create, edit, or inspect. You create or open a reference when you use VIs, functions, properties, and methods that return a reference. As long as a VI reference remains open, LabVIEW cannot remove the corresponding VI from memory. Therefore, when a large number of references are open, a large number of VIs and objects are in memory, causing LabVIEW to perform more slowly and possibly preventing LabVIEW from creating new VIs and objects if no more memory is available. To avoid these performance issues, close references to VIs or objects that you open within a scripting VI.

You do not have to close the following types of references since leaving them open does not impact performance:

- References that inherit from GObject
- This VI or This Application static references

Use the Close Reference function to close references that you create or open. Note that the following actions close the reference as part of their operation, so you do not need to call Close Reference.

Action	Result
Using the Generic:Delete method	Closes the reference to the object that you are deleting

Action	Result
Using the GObject:Replace method	Closes the reference to the object that you are replacing
Closing a reference to a VI	May invalidate references to objects within the VI

What to Use

This VI Scripting task requires the following objects:

Close Reference



What to Do

Before you start: Open the Closing References VI in the labview\examples\Application Control\VI Scripting\Managing References directory for an example that illustrates each step of the following procedure.

1. Identify references to VIs and VI objects on the block diagram of the scripting VI. Ignore the previously mentioned reference types that you do not need to close.

Example Details

In the example, the scripting VI opens a reference in the following places:

- Open VI Reference—Opens a reference to the target VI
- Traverse for GObjects—Opens a reference to each Function object in the target VI
- Function:Terminals[]—Opens references to all the terminals of each function
- Terminal:Connected Wire—Opens a reference to the wire that is connected to each terminal
- Wire:Terminals[]—Opens references to all the terminals of a wire
- Function:Replace—Opens a reference to the replacement function
- Function:Terminals[]—Opens references to all the terminals of the replacement function

2. Determine the last time that LabVIEW uses each reference.

When tracing the data flow of a reference, remember that LabVIEW requires an open reference to a node when you use a reference to any terminal of that node.

Example Details

In the example, the open references are last used in the following places:

- Target VI reference—At the end of the VI. LabVIEW uses this reference throughout the VI because the other open references are to objects within the target VI.
 - Function references—After the Function:Replace method
 - Function terminal references—After the Terminal:Connected Wire method
 - Wire reference—After the Wire:Delete method
 - Wire terminal references—After the Terminal:Connect Wire method
 - Replacement function reference—After the Function:Terminals[] method
 - Terminal references for replacement function—After the Terminal:Connect Wire method
3. Wire each open VI reference to a Close Reference function soon after LabVIEW last uses a reference to that VI or one of its object references. Use the data flow provided by the error cluster to control when LabVIEW executes each Close Reference function. Beware of the [results](#) when closing references.

Caveats and Recommendations

- **Close object references once you are done using all object references within that VI**—When you close a reference to a VI, all open references to objects within that VI might become invalid. You can safely close a VI reference only after you finish using both the VI reference and all references to objects within that VI.
- **Close references as soon as possible**—An excessive number of open references causes LabVIEW to run more slowly. For small applications, this effect is negligible, but large applications will exhibit poorer performance. When you close references early in the program, you prevent them from lingering in memory unnecessarily.
- **Close references before the end of a VI when possible**—LabVIEW automatically closes references in a VI that has finished executing, so you do not improve performance by closing references at the end of a VI.

Additional Examples

Refer to the Creating New VI From Scratch VI in the `labview\examples\Application Control\VI Scripting\Creating VIs` directory for another example of closing many references when using VI Scripting.

Navigating a VI Using VI Scripting

Requires: [VI Scripting](#)

Because block diagram objects in a VI are connected by terminals and wires, you can use a reference to one object to gain access to other connected objects. You can use this strategy of navigating a target VI in the following situations:

- Inspecting or editing connections among objects
- Obtaining a reference to an object that you cannot access directly with the Open VI Object Reference function or the Traverse for GObjects VI

If you can connect objects together manually in LabVIEW, LabVIEW most likely provides a property or method that allows you to navigate those connections programmatically. Therefore, all navigation tasks involve obtaining a reference to a starting object and then using the necessary properties and methods to obtain references to the connected objects. You can segment large navigation tasks into the following navigation subtasks:

- Identifying a starting point from which to navigate
- *Navigating from a node to a wire*
- *Navigating from a wire to a node*
- *Navigating across a structure*
- Navigating between a block diagram and its front panel

The following example demonstrates how you can combine some of the previous navigation subtasks to solve a larger navigation problem.

Example Task: Preserving Wiring when Replacing a Function

You can write a scripting VI that replaces a One Button Dialog function with a Two Button Dialog function in any target VI. However, after making this replacement, you must ensure that the object that was wired to the **button name** input of the One Button Dialog function is wired to the **T button name** input of the new Two Button Dialog function. The following images illustrate the desired change.



Why This Example Task Requires Navigation

To replace the One Button Dialog function, you need only a reference to the function. However, to ensure that the original wiring persists to the new Two Button Dialog function, you also require a reference to the object that is wired to the **button name** input of the One Button Dialog function. You can obtain this string reference by navigating the connections between the One Button Dialog function and the connected object.

How to Use Navigation to Complete This Example Task

The following outline highlights the navigation subtasks that are necessary to navigate from the One Button Dialog function to the object wired to its **button name** input. Refer to the individual task topics for specific information about how to implement each task.

- 1. **Identify a starting point for navigation** —You can use the One Button Dialog function as the starting point for navigation because you know that all target VIs of interest have a One Button Dialog function.
- 2. *Navigate from a node to a wire* —You can use the reference to the One Button Dialog function to obtain a reference to the wire that connects to the **button name** input of the function.
- 3. *Navigate from a wire to a node* —You can use the wire reference to obtain a reference to its source node.

After completing these three navigation tasks, you have access to all the references you need to complete the task.

Refer to the Navigation Overview VI in the labview\examples\Application Control\VI Scripting\Finding and Modifying Objects directory for a full solution to this scripting task.

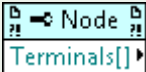
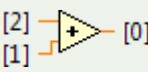
Navigating from a Node to a Wire Using VI Scripting

Requires: *VI Scripting*

You navigate from a node to a wire any time that you use a reference to a node to obtain a reference to a connected wire. Navigating from a node to a wire is a navigation subtask that you can combine with other navigation subtasks to accomplish larger navigation goals. Refer to the *navigation overview* to learn more about how navigating from a node to a wire interacts with other navigation tasks.

What to Use

This VI Scripting task requires the following objects:

Node:Terminals[]	<i>Visible Terminal Indexes</i>	Index Array	Terminal:Connected Wire	Close Reference
				

What to Do

Before you start: Open the Navigating Nodes and Wires VI in the labview\examples\Application Control\VI Scripting\Finding and Modifying Objects directory for an example that illustrates each step of the following procedure.

1. [Obtain a reference](#) to a node.

Example Details

In the example, the scripting VI uses the Traverse for GObjects VI to obtain references to all the Function nodes in the target VI. The Case structure checks the style of each Function node so that navigation continues only from the Function nodes whose style is One Button Dialog.

2. Use the Node:Terminals[] property to obtain references to the terminals of the node.

Example Details

In the example, the Node:Terminals[] property appears as Function:Terminals[] because the reference is a Function reference. Because Function is a subclass of Node, Function inherits the Terminals[] property of Node.

3. Refer to the **Context Help** window of the node to identify the terminal index of the terminal that connects the node to the destination wire.

You must [display additional VI scripting information in the Context Help window](#) to cause the connector pane in the Context Help window to display a bracketed number next to each terminal. This number represents the index in the Terminals[] array where the reference to the corresponding terminal is stored.

Example Details

In the example, the scripting VI requires the index of the **button name** input of the One Button Dialog function. According to the Context Help window for the One Button Dialog function, the index for that input terminal is 1.

4. Use the Index Array function with the desired terminal index to obtain a reference to the desired terminal in the Terminals[] array.
5. Use the Terminal:Connected Wire property of the returned terminal reference to obtain a reference to the wire connected to that terminal.
6. Use the returned wire reference to either modify the wire or to [continue navigating](#).

Example Details

In the example, the scripting VI uses the wire reference to continue [navigating from a wire to a node](#) by using the Wire:Terminals[] property of the wire.

7. Use the Close Reference function to close the node reference, the terminal reference, and the wire reference when you are finished using them.

Caveats and Recommendations

- [Display terminal indexes in the Context Help window](#) —You can access the desired terminal in the Node:Terminals[] array if you know the index of the desired terminal.

- **Beware of invalid references in the Node:Terminals[] array** —The Node:Terminals[] array contains one element for every terminal of a Node. However, each element is a valid reference to a terminal only if the terminal is wired. Because many VIs and functions have optional inputs that are not always wired, the array of terminals often contains invalid references for these terminals. Therefore, if you are not sure whether a specific terminal is wired, check the corresponding element in the terminals array to see if it is an invalid reference before attempting to use it elsewhere in a scripting VI.
- **Close all references that you open** —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and object references](#) for more performance details.

Additional Examples

Refer to the Navigation Overview VI in the labview\examples\Application Control\VI Scripting\Finding and Modifying Objects directory for a complete navigation example that includes navigating from a node to a wire and navigating across a structure.

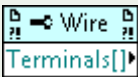
Navigating from a Wire to a Node Using VI Scripting

Requires: [VI Scripting](#)

You navigate from a wire to a node any time that you use a wire reference to obtain a reference to a node that it connects to. Navigating from a wire to a connected node is a navigation subtask that you can combine with other navigation subtasks to accomplish larger navigation goals. Refer to [navigation overview](#) to learn more about how navigating from a wire to a node interacts with other navigation tasks.

What to Use

This VI Scripting task requires the following objects:

Wire:Terminals[]	Index Array	Terminal:Owner	Close Reference
			

What to Do

Before you start: Open the Navigating Nodes and Wires VI in the labview\examples\Application Control\VI Scripting\Finding and Modifying Objects directory for an example that illustrates each step of the following procedure. Note that this example also illustrates the procedure for navigating from a node to a wire.

1. **Obtain a reference** to a wire.

When navigating from a wire to a node, you most commonly obtain the wire reference as the result of previous navigation.

Example Details

In the example, the scripting VI obtains the initial wire reference by navigating from the One Button Dialog function to the wire connected to the **button name** input of that function.

2. Use the Wire:Terminals[] property of the wire to obtain references to the terminals of the wire.
3. Identify which terminal or terminals of the wire best further your navigation goals.

In some navigation tasks, you might be interested in only one terminal of the wire. In other navigation tasks, you might be interested in inspecting or navigating across all the terminals of the wire.

Example Details

In the example, the scripting VI requires only the source terminal of the wire.

4. Use the Index Array function with the Terminals[] array from step 2 to obtain a reference to the desired terminal.

The first element in the Terminals[] array is always a reference to the source terminal of the wire. If the wire connects only two terminals, the second element in the array is a reference to the sink terminal of the wire. If the wire has multiple sink terminals, the array stores them in random order beginning at index 1 of the array.

When a wire has multiple sink terminals, you can use a For Loop to iterate across each terminal reference in the array. Repeat step 5 for each terminal reference.

Example Details

In the example, the scripting VI requires a reference only to the source terminal of the wire. Because the source terminal is always stored at the beginning of the Terminals[] array of a wire, the scripting VI uses the Index Array function to obtain a reference to the terminal at index 0 of the Terminals[] array.

5. Use the Terminal:Owner property of the returned terminal reference to obtain a reference to the connected node.
6. (Optional) Use the To More Specific Class function to cast the returned reference to its actual class. This step provides the following benefits:
 - Identifies class—The To More Specific Class function casts the referenced object to a more specific class only if the object is a member of that class. Therefore, you can use the To More Specific Class function to check the class of the reference. This check is necessary when you want to interact with the referenced object only if it is a member of a certain class.
 - Enables access to more properties and methods—When you cast the reference to a more specific class, you can use more specific properties and methods of the reference. The Terminal:Owner property in the previous step always returns a GObject reference, a class which supports only a limited set of properties and methods.

Example Details

In the example, the scripting VI can modify the returned node only if it is a string constant. Thus, the scripting VI tries to cast the reference to a string constant. The Case structure determines whether the To More Specific Class function returns an error and therefore whether the referenced object is a string constant.

7. Use the node reference to either edit the node or to continue navigating.

Example Details

In the example, the scripting VI edits the node rather than using it to continue navigating. The scripting VI sets the value of the node, a string constant, to Okay! by using the StringConstant:Value property.

8. Use the Close Reference function to close the wire reference, the terminal reference, and the node reference when you are finished using them.

Caveats and Recommendations

- **Downcast the reference returned by the Terminal:Owner property** —The Terminal:Owner property always returns a reference to a Generic object. However, if you know that the terminal connects the wire to a more specific class of object, you can use the To More Specific Class function to downcast the returned reference to the more specific class. When you downcast a reference to a more specific class, you are able to call more specific properties or methods on the reference.
- **Close all references that you open** —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and object references](#) for more performance details.

Additional Examples

Refer to the Navigation Overview VI in the `labview\examples\Application Control\VI Scripting\Finding and Modifying Objects` directory for a complete navigation example that includes navigating from a node to a wire.

Navigating across a Structure Using VI Scripting

Requires: [VI Scripting](#)

You navigate across a structure any time that you inspect, edit, or create references to a particular part of a structure. Navigating across a structure is a unique navigation task because the elements of a structure are connected by the structure itself rather than by wires. Therefore, to navigate a structure, you must not only be aware of the different elements that compose a structure but also of the properties that return references to these elements. You can combine navigating across structures with other navigation subtasks to accomplish larger navigation goals. Refer to the [navigation overview](#) to learn more about how navigating across structures interacts with other navigation tasks.

What to Use

To navigate across a structure, you use properties of the structure to obtain references to different parts of the anatomy of the structure. The properties you need to navigate across a structure depend on the structure that you are navigating. Refer to the following structure anatomies to identify which properties you need to navigate a particular structure.

- [Loop Properties](#)
- [Case Structure Properties](#)
- [Timed Sequence Structure Properties](#)

What to Do

Before you start: Open the Navigating Structures VI in the `labview\examples\Application Control\VI Scripting\Finding and Modifying Objects` directory for an example that illustrates each step of the following procedure.

1. [Obtain a reference](#) to the structure you want to navigate.

Example Details

In the example, the scripting VI uses the Traverse for GObjects VI to obtain references to all the Case structures in the target VI.

2. Use the structure reference to obtain a reference to the element of the structure that is relevant to your overall navigation task.

Possible structure elements include tunnels, shift registers, Case structure frame names, and so on. Refer to the [anatomies of common VI objects](#) for a visual depiction of the parts of different types of structures.

Example Details

In the example, the scripting goal is to navigate to and analyze the object that is wired to the case selector. Therefore, the scripting VI uses the Case Structure:Selector property to obtain a reference to the selector tunnel of each Case structure. This reference enables navigation to continue toward the wired object.

3. Use the reference to the structure element to either edit the element or to continue navigating.

As the structure anatomies indicate, some elements of a structure provide access to other elements of the structure. For example, you can use a reference to a tunnel to obtain a reference to the inner and outer terminals of that tunnel.

Example Details

In the example, the scripting VI uses the reference to the case selector to continue [navigating from a node \(the case selector\) to a connected wire](#). The scripting VI uses the Tunnel:Outside Terminal property of the case selector to obtain a reference to the outer terminal of the case selector. The scripting VI requires the terminal reference to obtain a reference to the connected wire.

4. Use the Close Reference function to close every reference to a structure or structure element when you are finished using them.

Caveats and Recommendations

Close all references that you open —An excessive number of open references negatively impacts the performance of a VI. Read about [closing VI and object references](#) for more performance details.

Additional Examples

None

VI Scripting Tutorial

Requires: [VI Scripting](#)

This book contains a tutorial that uses [VI Scripting](#) functions, properties, and methods to create a blank VI, add objects to the block diagram, reposition those objects, and wire the objects together.

[Begin the tutorial.](#)

For details about the programming concepts illustrated in the tutorial, read more about [common VI Scripting tasks](#).



(Windows) To view related topics, click the **Locate** button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the **Contents** tab so you can navigate the related topics.

Tutorial: Creating and Opening a VI Using VI Scripting

Complete the following steps to create a VI that uses VI Scripting to create and open a target VI.

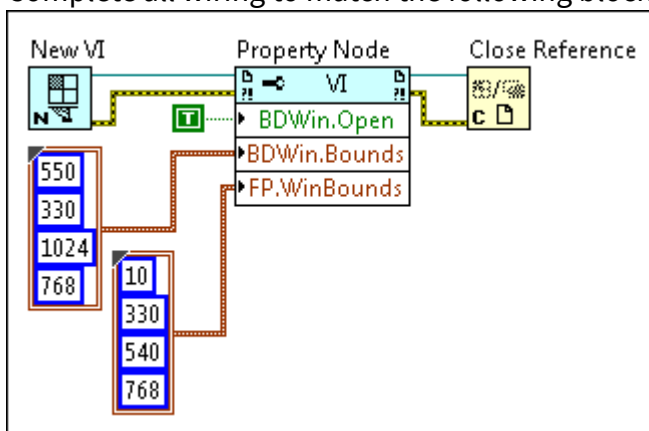


NOTE

- This topic assumes familiarity with the [VI Scripting](#) introduction.
- You must [enable VI Scripting](#) to display the VI Scripting VI and functions and use the associated properties and methods.

1. Select **File»New VI** to create a new VI.
2. Add the New VI function to the block diagram.

3. Add a Property Node to the right of the New VI function.
4. Wire the **vi refnum** output of the New VI function to the **reference** input of the Property Node.
5. *Resize the Property Node* so it contains three terminals.
6. Add the following properties to the Property Node.
 - Block Diagram Window: Open —Click the first terminal of the Property Node and select **Block Diagram Window»Open** from the shortcut menu. Because the Block Diagram Window properties belong to the VI Scripting class, the Property Node turns blue when you add the Block Diagram Window properties.
 - Block Diagram Window: Window Bounds —Click the second terminal of the Property Node and select **Block Diagram Window»Window Bounds** from the shortcut menu.
 - Front Panel Window: Window Bounds —Click the third terminal of the Property Node and select **Front Panel Window»Window Bounds** from the shortcut menu.
7. Right-click the Property Node and select **Change All To Write** from the shortcut menu.
8. Create constants for each property and configure the constants to match the block diagram in step 10.
9. Add the Close Reference function to the right of the Property Node.
10. Complete all wiring to match the following block diagram.



11. *Save the VI* as ScriptingTutorial.vi.
12. *Run the VI*.

LabVIEW creates and opens untitled front panel and block diagram windows.

To continue this tutorial, continue to the [Tutorial: Adding an Object to a VI Using VI Scripting](#) topic.

Tutorial: Adding an Object to a VI Using VI Scripting

You can add objects to the front panel and block diagram of VIs using VI Scripting. This tutorial adds an object to the block diagram. Complete the following steps to add an object to a VI using VI Scripting.



NOTE

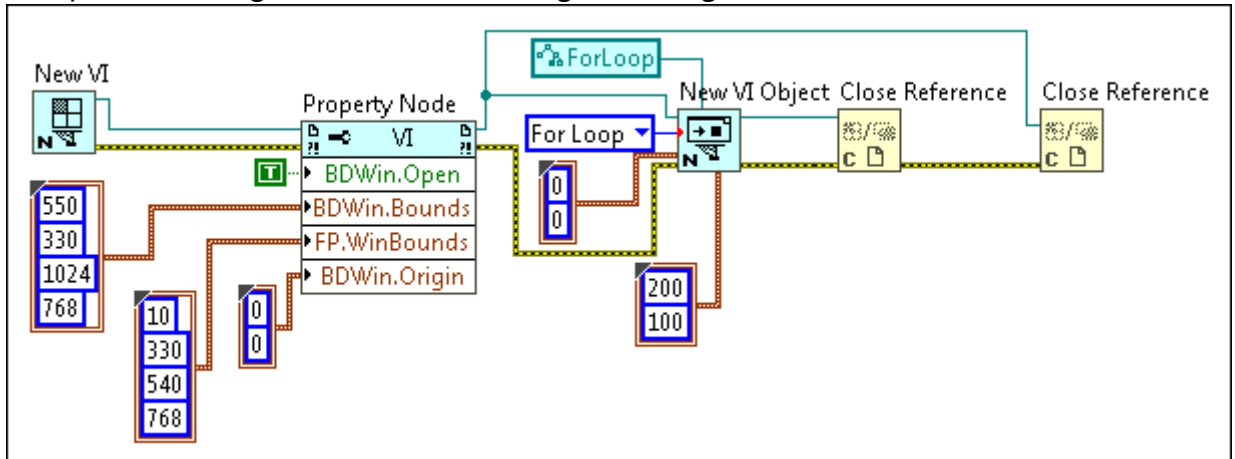
- This topic assumes familiarity with the [VI Scripting](#) introduction.
- You must [enable VI Scripting](#) to display the VI Scripting VI and functions and use the associated properties and methods.

1. Complete the [previous exercise](#) if you have not already completed it. If the untitled VI you created in the previous exercise is still open, close the VI.
2. Remove the wires from the Close Reference function and move the function to the right to make room on the block diagram.
3. Add the Block Diagram Window: Origin property to the Property Node and create a constant with **Horizontal** and **Vertical** input values of 0. This configuration sets the horizontal and vertical coordinates in the upper left-hand corner of the block diagram. Later in the exercise, you set the location of a For Loop so it also appears in the upper left-hand corner.
4. Add the New VI Object function to the right of the Property Node.
5. Wire the **reference out** output of the Property Node to the **owner refnum** input of the New VI Object function.
6. Right-click the **vi object class** input of the New VI Object function and select **Create»Constant** from the shortcut menu to create a class specifier constant.
7. Click the class specifier constant and select **Generic»GObject»Node»Structure»Loop»ForLoop** from the shortcut menu.

Note In this exercise, you add a For Loop to the block diagram. However, with knowledge of the [appropriate references](#), you can add and manipulate any front panel or block diagram object.

8. Right-click the **style** input of the New VI Object function, select **Create»Constant** from the shortcut menu, and select **For Loop** from the pull-down menu of the constant.
9. Right-click the **location** input of the New VI Object function, select **Create»Constant** from the shortcut menu, and enter 0 as the value for the **Horizontal** and **Vertical** inputs. This step specifies the coordinates of the location on the block diagram where you want the new object to appear.
10. Right-click the **bounds** input of the New VI Object function, select **Create»Constant** from the shortcut menu, and enter 200 and 100 for the **Width** and **Height** inputs, respectively. This step specifies the size of the For Loop.
11. Add another Close Reference function to the right of the existing Close Reference function. Closing both the reference to the VI and the reference to the For Loop ensures no memory leaks. Because the VI is the containing object of the For Loop, make sure to close the reference to the VI after you close the reference to the For Loop.

12. Complete all wiring to match the following block diagram.



13. *Save the VI.*

14. *Run the VI.*

LabVIEW creates and opens untitled front panel and block diagram windows. The block diagram contains a For Loop in the upper left-hand corner.

Additional Options for Creating Object References

Consider the following additional options to create references to objects:

- Add a custom object to the front panel or block diagram by wiring a path for the custom object to the New VI Object function, or use the Create from Reference method.
- Use the To More Specific Class function to convert a generic object reference to a more specific reference, like a control. You then can use the properties of the more specific reference type.

To continue this tutorial, continue to the [Tutorial: Repositioning an Object in a VI Using VI Scripting](#) topic.

Tutorial: Repositioning an Object in a VI Using VI Scripting

You can reposition objects on the front panel and block diagram of VIs using VI Scripting. This tutorial repositions an object on the block diagram. Complete the following steps to reposition an object on the block diagram of a VI using VI Scripting.

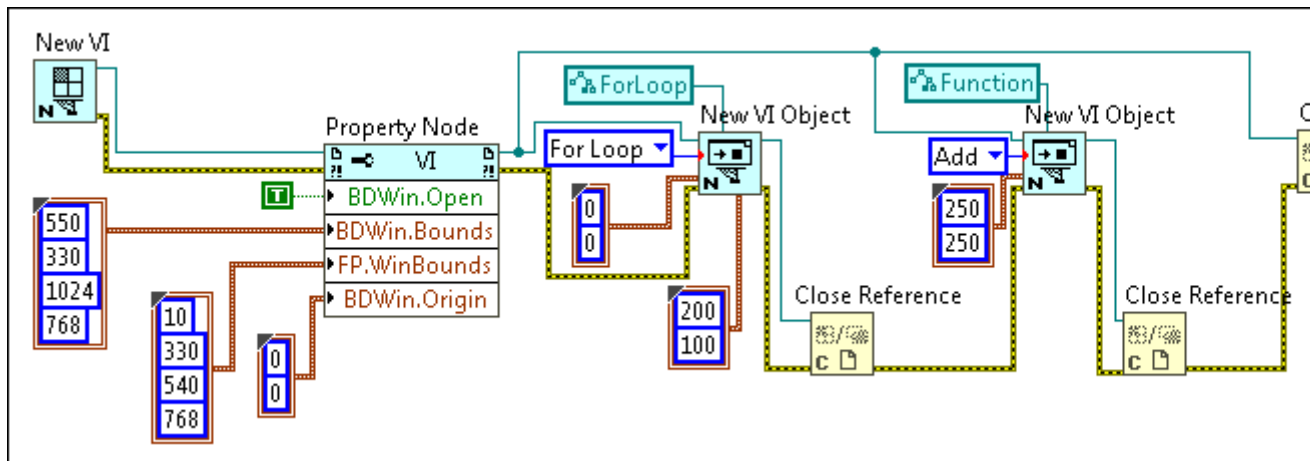


NOTE

- This topic assumes familiarity with the [VI Scripting](#) introduction.
- You must [enable VI Scripting](#) to display the VI Scripting VI and functions and use the associated properties and methods.

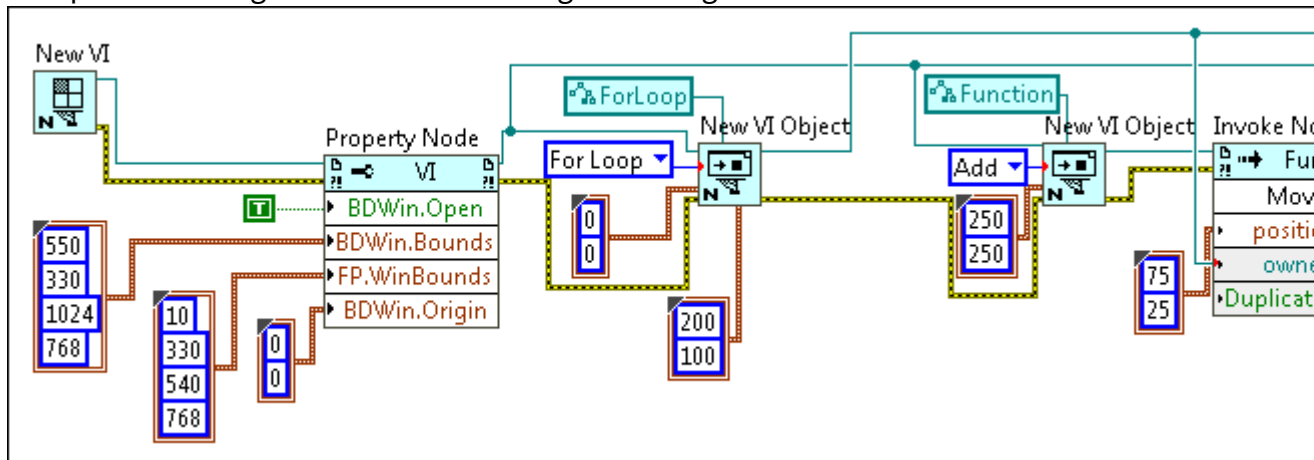
1. Complete the [previous exercise](#) if you have not already completed it. If the untitled VI you created in the previous exercise is still open, close the VI.
2. Remove the wires from the Close Reference function that closes the VI reference and move the function to the right to make room on the block diagram.
3. Add another New VI Object function to the block diagram to create an Add function on the block diagram of the target VI. Configure the New VI Object function and complete the wiring as shown in the following block diagram. Make sure to close all references. Specify the **owner refnum** as the VI.

Tip To specify Function in the class specifier constant, select **Generic»GObject»Node»Function»Function** from the shortcut menu.



4. **Save the VI.**
5. **Run the VI.** LabVIEW creates and opens untitled front panel and block diagram windows. The block diagram contains a For Loop in the upper left-hand corner and an Add function near the bottom of the block diagram. You can place the Add function inside the For Loop by wiring the **object refnum** output of the New VI Object function that creates the For Loop to the **owner refnum** input of the New VI Object function that creates the Add function. Then, you can specify coordinates that fall within the For Loop in the **location** input of the New VI Object function used to create the Add function. However, to examine another option for repositioning block diagram objects, the next steps of the tutorial use the Move method to move the Add function into the For Loop.
6. If the untitled VI LabVIEW created is still open, close the VI.
7. Remove the wires from all three Close Reference functions, and move the functions to the right to make room on the block diagram. Refer to the block diagram in step 13 for a visual reference.
8. Add an Invoke Node to the right of the New VI Object function that creates the Add function.
9. Wire the **object refnum** output of the New VI Object function to the **reference** input of the Invoke Node.
10. Click the **Method** terminal of the Invoke Node and select **Move** from the shortcut menu.
11. To specify the block diagram object into which you want to move the Add function, wire the **object refnum** output of the New VI Object function that creates the For Loop to the **owner** input of the Invoke Node.

12. To specify a location within the For Loop to where you want to move the Add function, right-click the **position** input of the Invoke Node, select **Create»Constant** from the shortcut menu, and enter 75 and 25 as values for the **Left** and **Top** inputs, respectively.
13. Complete all wiring to match the following block diagram.



14. Save the VI.
15. Run the VI.

LabVIEW creates and opens untitled front panel and block diagram windows. The block diagram contains a For Loop in the upper left-hand corner and an Add function inside the For Loop.



Note Use the Delete method to delete an object using VI Scripting. You also can use the Replace method to replace an object. To clean up any broken wires after a delete operation, use the Block Diagram:Remove Bad Wires method.

To continue this tutorial, continue to the [Tutorial: Wiring Objects Using VI Scripting](#) topic.

Tutorial: Wiring Objects Using VI Scripting

Use VI Scripting to wire objects on the block diagram programmatically. Most objects you can place on the block diagram are considered nodes. For example, functions, VIs, and structures are nodes. Typically, nodes have multiple terminals to which you can wire. Use the Connect Wires method to [connect a wire, or wires, to a node](#). Use the Connect Wire method to [connect a wire to a single terminal](#).



NOTE

- This topic assumes familiarity with the [VI Scripting](#) introduction.
- You must [enable VI Scripting](#) to display the VI Scripting VI and functions and use the associated properties and methods.

Wiring Nodes

Complete the following steps to wire two Add functions together using VI Scripting.



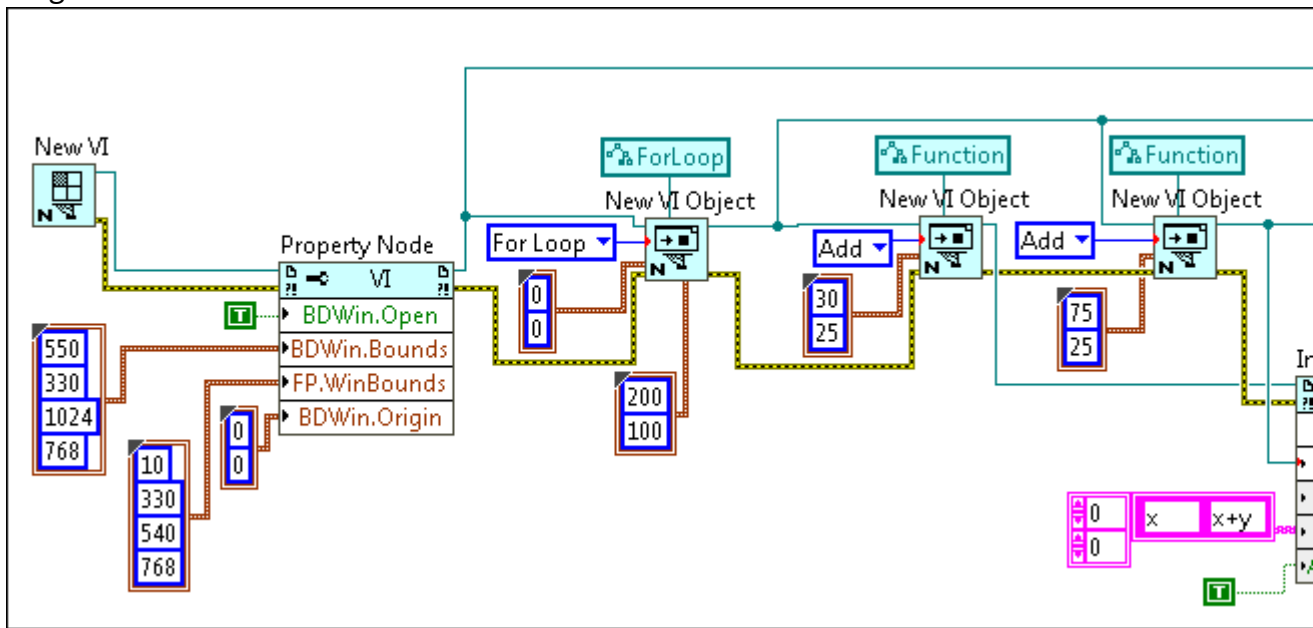
Note Nodes are in the **Generic»GObject»Node** class.

1. Complete the [previous exercise](#) if you have not already completed it. If the untitled VI you created in the previous exercise is still open, close the VI.
2. Delete the Move method and related wires from the block diagram.
3. Complete the following steps to place the Add function in the For Loop.
 1. Enter 30 and 25 as the values of the **Horizontal** and **Vertical** inputs, respectively, for the New VI Object function that creates the Add function.
 2. Remove the wire from the **owner refnum** input of the New VI Object function that creates the Add function, as well as any broken wires.
 3. Wire the **object refnum** output of the New VI Object function that creates the For Loop to the **owner refnum** input of the New VI Object function that creates the Add function.
4. Create another Add function inside the For Loop with the values 75 and 25 in the **Horizontal** and **Vertical** inputs, respectively. Refer to the block diagram in step 12 for a visual reference.
5. Add an Invoke Node to the block diagram to the right of the New VI Object function you added in the previous step.
6. Wire the **object refnum** output of the New VI Object function that creates the first Add function to the **reference** input of the Invoke Node.
7. Click the Invoke Node and select **Connect Wires** from the shortcut menu.
8. Wire the **object refnum** output of the New VI Object function that creates the second Add function to the **Wire Source** input of the Connect Wires method.
9. Right-click the **Wiring Specs** input of the Connect Wires method and select **Create»Constant** from the shortcut menu.
10. Enter the names of the input and output you want to connect with a wire in the **Wiring Specs** constant, in this case x and x+y.

Note Alternatively, situations might allow for auto-wiring rather than explicitly specifying the name of the terminals you want to wire. You can set the **Auto Wire?** input to TRUE to allow LabVIEW to wire the source object to a compatible, nearby object automatically.

11. To specify that LabVIEW finds the best path for the wire, right-click the **Auto Route?** input of the Connect Wires method, create a constant, and set the constant to TRUE.

12. Close references to the objects and complete all wiring to match the following block diagram.



13. *Save the VI.*
14. *Run the VI*

LabVIEW creates and opens untitled front panel and block diagram windows. The block diagram contains a For Loop in the upper left-hand corner and two Add functions in the For Loop. The Add functions are wired together.

Wiring Objects

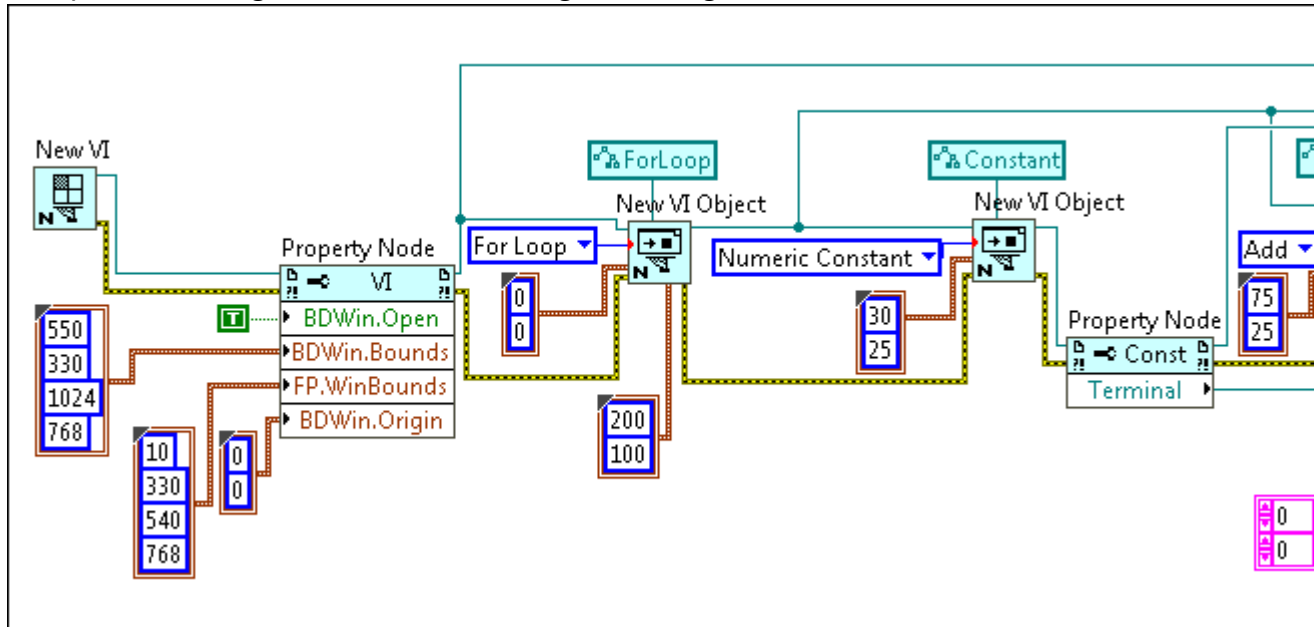
This exercise wires a terminal to a node. To wire to a terminal, use the Connect Wire method. Complete the following steps to wire a constant to an Add function using VI Scripting.



Note The terminal reference for a numeric constant is in the **Generic»GObject»Constant** class.

1. Complete the [Wiring Nodes](#) exercise, if you have not already completed it. If the untitled VI you created in the previous exercise is still open, close the VI.
2. On the New VI Object function that creates the first Add function, change the **style** input to **Numeric Constant**, change the **vi object class** input to **Constant**, and remove the wires from the **object refnum** and **error out** outputs.
3. Add a Property Node to the block diagram to the right of the New VI Object function that now creates a numeric constant. Refer to the block diagram in step 9 for a visual reference.
4. Wire the **object refnum** output of the New VI Object function that creates the numeric constant to the **reference** input of the Property Node.
5. Click the Property Node and select **Terminal»Reference** from the shortcut menu.

6. Wire the **Terminal** output of the Property Node to the **reference** input of the existing Invoke Node.
7. Click the Invoke Node and select **Connect Wire** from the shortcut menu.
8. Change the value of the **Wiring Specs** input of the Connect Wire method to the names of the input and output you want to connect with a wire, in this case x and x.
9. Complete all wiring to match the following block diagram.



10. Save the VI.
11. Run the VI

LabVIEW creates and opens untitled front panel and block diagram windows. The block diagram contains a For Loop in the upper left-hand corner, and a numeric constant and Add function in the For Loop. The numeric constant and the Add function are wired together.

Adding New Visible VIs Entries

Complete the following steps to add a new entry to the **Visible VIs** list.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Web Server from the **Category** list of the **Options** dialog box.
3. Enter the name of the VI or directory path in the **Visible VI** text box in the **Visible VIs** section. You also can *use wildcards* in the VI name or directory path. To specify a VI that is part of a LabVIEW project, you must include the project name, the project library, and the target in the path of the VI, when applicable. For example, if `MyVI.vi` resides in a project called `MyProject.lvproj` under target `My Computer`, enter the VI name as `MyProject.lvproj/My Computer/MyVI.vi`. If the VI resides in a project library called `MyLibrary`, also include the project library in the path, as in `MyProject.lvproj/MyLibrary/`

`MyLibrary.lvlib:MyVI.vi`. If the VI is not in a project or project library, you can enter the VI name without any additional information.

4. Click the **Allow Access** button to allow access to the selected item. Click the **Deny Access** button to deny access to the selected item. If you are controlling a front panel remotely, you can [set a time limit](#) on how long a remote client can control a VI when multiple clients are waiting to control the VI. A green checkmark appears to the left of the item when you allow access, and a red X appears when you deny access.
5. If an entry does not have a green checkmark or a red X by its name, the syntax for the entry is incorrect. Edit the entry or click the **Remove** button to delete it.
6. Click the **OK** button to close the **Options** dialog box and save the changes.

Editing Existing Visible VIs Entries

Complete the following steps to edit the name of a VI or directory path in the **Visible VIs** list.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Web Server from the **Category** list.
3. Select the VI name or directory path you want to modify in the **Visible VIs** list in the **Visible VIs** section. The name or path appears in the text box.
4. Edit the name or path in the text box or select the **Allow Access** or **Deny Access** radio button to allow or deny access. You also can [set a time limit](#) on how long a remote client can control a VI when multiple clients are waiting to control the VI. The default is 300 seconds. You can [use wildcards](#) in the VI name or directory path.
5. Click the **OK** button to close the **Options** dialog box and save the changes.

Removing Visible VIs Entries

Complete the following steps to remove a VI name or directory path from the **Visible VIs** list.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Web Server from the **Category** list.
3. Select the VI name or directory path you want to remove in the **Visible VIs** list in the **Visible VIs** section.
4. Click the **Remove** button.
5. Click the **OK** button to close the Options dialog box and save the changes.

Calling Non-Modal Windows Programmatically

You cannot open a non-modal window from a LabVIEW callback VI nor a DLL while any other process is running. If you want the callback VI or DLL to call a non-modal window, you must do so programmatically by completing the following steps:

1. Add an Open VI Reference node to the block diagram.
2. Wire the path for the non-modal window you want to open to the **vi path** input.



NOTE

You only can wire a path for a VI-based window to the **vi path** input. You can [use the Call Library Function Node](#) to open a window that is not VI-based. Before calling a window from a LabVIEW callback VI or DLL, be sure that window is configured as modal.

1. Add an Invoke Node to the block diagram.
2. Wire the **vi reference** output of the Open VI Reference node to the **reference** input of the Invoke Node.
3. Select **Run VI** as the method on the Invoke Node.
4. Wire a FALSE Boolean constant to the **Wait Until Done** input of the method.

Importing a Web Service as a Library of VIs

Use the Import Web Service wizard to create or update wrapper VIs based on the methods in a Web service. You must provide a valid URL to a Web Service Description Language (WSDL), and the WSDL must validate correctly with ASP.NET. WSDL is an XML-formatted language used to describe a Web service and its functionality.



NOTE

- You must have the .NET Framework 4.0 installed to use the Import Web Service wizard.
- If you change the URL for the Web Service you want to import, you must complete the following steps to regenerate the VIs for the Web Service.

Complete the following steps to create or update a library of VIs based on methods in a Web service.

1. Select **Tools»Import»Web Service** to launch the Import Web Service wizard.
2. Enter the URL to the WSDL for the Web service you want to import.
3. Click the **Next** button to display the next page of the wizard.
4. (Optional) Enter the authentication or proxy information.
5. Click the **Next** button to display the next page of the wizard.
6. Enter the name of the project library `.lvlib` file you want to create.

7. Enter the destination directory, which is the location to which you want to save the project library .lvlib file. You also can browse to navigate to and select a folder.
8. Click the **Next** button to display the next page of the wizard.
9. Select the methods to import. For each method you select, LabVIEW creates a VI with parallel functionality.
10. Click the **Next** button to display the **Summary** page of the wizard. Use this page to review the Web service information you have specified before LabVIEW creates the wrapper VIs. You can use the **Back** button to make any changes.
11. Click the **Generate** button when you are ready to import the Web service methods and generate wrapper VIs for them. The **Completed** page appears when generation is complete.
12. Select the appropriate options if you want to open the imported project library file or view the report upon closing the wizard.
13. Click the **Finish** button to close the wizard and optionally open the specified files.

Objects

Aligning Objects

Use the alignment grid to *align objects as you place them*.

To align objects along axes after you place them, select the objects you want to align and use the **Align Objects** pull-down menu  on the toolbar or select **Edit»Align Items**. You can align an object along a horizontal axis using its top or bottom edge or its center point. You can align an object along a vertical axis using its left or right edge or its center point. The alignment option you select remains in effect until you change it. A dark border around the icon in the pull-down menu indicates the current alignment option.

Complete the following steps to align objects.

1. *Select the objects* you want to align.
2. Select the **Align Objects** pull-down menu, located next to the **Text Settings** pull-down menu on the toolbar on the front panel and block diagram.
3. Select from the following options:
 - **Top Edges** —Aligns all top edges of the selected objects with the topmost object.
 - **Vertical Centers** —Aligns the objects at the point halfway between the topmost and bottommost objects.
 - **Bottom Edges** —Aligns all bottom edges of the selected objects with the bottommost object.
 - **Left Edges** —Aligns all left edges of the selected objects with the object farthest to the left.
 - **Horizontal Centers** —Aligns the objects at the point halfway between the object farthest to the left and the object farthest to the right.

- **Right Edges** —Aligns all right edges of the selected objects with the object farthest to the right.

To repeat an alignment operation on another group of objects, press the <Ctrl-Shift-A> keys. (macOS) Press the <Command-Shift-A> keys. (Linux) Press the <Alt-Shift-A> keys.

You also can select **Edit»Align Items** to repeat an alignment operation on another group of objects.

Coloring Objects and Backgrounds

You can add or change the color of front panel objects, front panel panes, and block diagram workspaces. Some objects have a foreground and a background you can color separately. For example, the foreground color of a knob on the **Classic Numeric** palette is the color of the main dial area, and the background color is the color of the raised edge. You can [create custom colors](#), [copy colors from one object to another](#), and [create transparent objects](#).

Complete the following steps to change the color of an object, the front panel background, or background of the block diagram workspace.

1. Use the Coloring tool to right-click an object or workspace to display the color picker. If [automatic tool selection is enabled](#), select the Coloring tool from the **Tools** palette to disable automatic tool selection.



TIP

Press the <Shift> key and right-click to display a temporary version of the **Tools** palette at the location of the cursor.

The top of the color picker contains a grayscale spectrum and a box you can use to [create transparent objects](#). The second spectrum contains muted colors that are well suited to backgrounds and front panel objects. The third spectrum contains colors that are well suited to highlights. The first row of color boxes displays [user-defined colors](#). The second row of color boxes displays the most recently selected colors and the [system colors](#). System colors allow you to set the appearance of a front panel to match the system colors of any computer that runs the VI. The system colors are three pairs of colors each containing a background color and a text color. There are colors for panel and object, window, and highlight for items like menus. The box at the bottom of the color picker displays the currently selected color. Objects that have foreground and background colors display the foreground color in the left half of this box and the background color in the right half.

2. Move the cursor through the color picker to select a color and preview the object or background in the selected color. As you move the cursor over a color, LabVIEW displays the RGB value of the color at the bottom of the color picker. If the object has foreground and background colors, press the <F> key to select the foreground color so you can set it or press the key to select the background color so you can set it. You also can press the spacebar to toggle between the foreground and background colors.
3. (Optional) Click the **More Colors** button in the bottom right corner of the color picker to [select a specific color](#) using the color utility on the operating system.

4. Click a color to apply that color to the object or background. To cancel the coloring operation, press the <Esc> key.

Copying and Pasting Objects

You can copy and paste objects in two ways. You can [select an object](#) and select **Edit»Copy** and **Edit»Paste**, or you can clone the object.

Complete the following steps to clone an object.

1. Press the <Ctrl> key while you use the Positioning tool to click the object. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.
2. Continue pressing the key while you drag the copy to a new location.

(Linux) You also can clone objects by clicking the object with the middle mouse button and dragging the object to a new location.

When you clone or copy objects that have labels, LabVIEW copies the name of the original object and appends a number.



NOTE

If you copy the object to another VI, you also copy the description. You might need to change the description for the new object.

When you clone or you copy a block diagram terminal, a new terminal appears on the block diagram. LabVIEW does not copy the local variables, control references, Property Nodes, and Invoke Nodes of the original terminal.

If you copy and paste a local variable or control reference, LabVIEW also copies the terminal. If you clone a local variable or control reference, LabVIEW creates a new reference to the original terminal.

When you copy front panel controls from the block diagram and paste them onto the block diagram of a new VI, the front panel placement differs from that of the original VI. LabVIEW places the front panel objects in the upper left corner to avoid overlapping or appearing in a non-visible region.

If you copy and paste an [implicitly linked Property Node or Invoke Node](#), LabVIEW disconnects the node from the terminal. If you clone an implicitly linked node, LabVIEW creates a new implicitly linked node. If you clone the node by dragging it to a new VI, LabVIEW disconnects the node from the terminal. You can [relink the node](#) to the terminal.

You can copy selected objects between VIs by dragging them from one VI and placing them on another VI.



Note If you copy a subVI, the Clipboard  appears in the VI Hierarchy window. Note that a subVI you copy might remain in memory in the Clipboard when you do not expect a VI to be in memory.

Copying Colors from One Object to Another

Complete the following steps to copy the color of one object and transfer it to a second object without using a color picker.

1. Use the Get Color tool to click the object whose foreground and background color you want to copy. If *automatic tool selection is enabled*, select the Get Color tool from the **Tools** palette to disable automatic tool selection.



TIP

Press the <Shift> key and right-click to display a temporary version of the **Tools** palette at the location of the cursor.

2. Use the Coloring tool to click the object to which you want to apply the color. The object becomes the color you copied from the first object.

Creating Transparent Objects

You can make front panel objects transparent, which allows you to layer multiple controls and indicators so that they appear in the same place when a user selects a particular option. Transparency affects only the appearance of an object. The object responds to mouse and key operations as usual.



NOTE

You cannot change the color of system controls because system controls change appearance depending on which platform you run the VI. These controls appear in the colors you have set up for your system.

Complete the following steps to make an object transparent:

1. Use the Coloring tool to right-click an object to display the color picker. If *automatic tool selection is enabled*, select the Coloring tool from the **Tools** palette to disable automatic tool selection.

Tip Press and hold <Shift> and right-click to display a temporary version of the **Tools** palette at the location of the cursor.

2. Select the **T** box in the upper right corner of the color picker.

You also can make parts of a front panel object transparent, including borders.

Complete the following steps to make the border of an object transparent:

1. Use the Coloring tool to right-click the border of an object such as a string control to display the color picker. If *automatic tool selection is enabled*, select the Coloring tool from the **Tools** palette to disable automatic tool selection.
2. Press and hold <F> and select the **T** box in the upper right corner of the color picker. LabVIEW makes the control border transparent. The background color of the front panel, or the object containing the control, determines the color of the control border.

Displaying Hidden Front Panel Objects

Complete the following steps to display a hidden front panel control or indicator. You also can *hide front panel controls and indicators*.

1. Find the block diagram terminal for the object. If you have several terminals with the same data type, right-click the terminals and select **Visible Items»Label** from the shortcut menu. Find the object with the label that matches the hidden front panel object.
2. Right-click the terminal and select **Show Control** from the shortcut menu if the object is a control or **Show Indicator** if the object is an indicator.

You also can use the Visible property to display controls and indicators programmatically.

You also can *display all hidden front panel controls and indicators*.

Distributing Objects

Use the **Distribute Objects** pull-down menu  or select **Edit»Distribute Items** to space objects evenly. The distribution option you select remains in effect until you change it. A dark border around the icon on the palette indicates the current distribution option.

Complete the following steps to distribute objects.

1. *Select the objects* you want to distribute.
2. Select the **Distribute Objects** pull-down menu, located to the right of the *Align Objects* pull-down menu on the toolbar in the front panel and block diagram windows.
3. Select from the following options:
 - **Top Edges** —Distributes the selected objects so that the top borders are spaced equally apart.
 - **Vertical Centers** —Distributes the selected objects so that the vertical centers are spaced equally apart.
 - **Bottom Edges** —Distributes the selected objects so that the bottom edges are spaced equally apart.
 - **Vertical Gap** —Distributes the selected objects so that the vertical space between the objects is equally sized.

- **Vertical Compress** —Compresses the selected objects so that there is no space between the top and bottom borders of the objects.
- **Left Edges** —Distributes the selected objects so that the left edges are spaced equally apart.
- **Horizontal Centers** —Distributes the selected objects so that the horizontal centers are spaced equally apart.
- **Right Edges** —Distributes the objects so that the right edges are spaced equally apart.
- **Horizontal Gap** —Distributes the selected objects so that the horizontal space between the objects is equally sized.
- **Horizontal Compress** —Compresses the selected objects so that there is no space between the left and right borders of the objects.

To repeat a distribution operation on another group of objects, press the <Ctrl-D> keys. (macOS) Press the <Command-D> keys. (Linux) Press the <Alt-D> keys.

You also can select **Edit>Distribute Items** to repeat a distribution operation on another group of objects.

Grouping and Locking Objects

Use the Positioning tool to select the front panel objects you want to group and lock

together. Click the **Reorder** button  on the toolbar and select **Group** or **Lock** from the pull-down menu. Grouped objects maintain their relative arrangement and size when you use the Positioning tool to move and resize them. Locked objects maintain their position on the front panel. You cannot delete individual objects that are grouped from the front panel or block diagram until you ungroup them. You cannot delete locked objects from the front panel or the block diagram until you unlock them. You can set objects to be grouped and locked at the same time. If you set objects to be grouped and locked at the same time, you must ungroup and unlock them before you can delete them from the front panel or the block diagram. Tools other than the Positioning tool work normally with grouped or locked objects.

Complete the following steps to group, ungroup, lock, or unlock front panel objects.

1. Select the objects with the Positioning tool by dragging an area around the objects, or holding down the <Shift> key while clicking each object. You cannot use the <Shift>-click method to select objects that are locked.
2. Click the **Reorder** button and select **Group**, **Ungroup**, **Lock**, or **Unlock** from the pull-down menu. You can set objects to be grouped and locked at the same time.



NOTE

When you group overlapping or layered objects, the objects that you select to group move to the same layer the frontmost object in the group occupies.

When you select an object that is part of a group, the group is highlighted with a yellow and black dotted line. When you select a locked object, it is highlighted with a gray and black

dotted line. When you select objects that are grouped and locked, the group is highlighted with a gray and black dotted line.

To change the relative arrangement of objects within the group, select **Ungroup**, rearrange the objects, and then select **Group**.

Reordering Objects

Placing a label or any other object over or partially covering a control or indicator slows down screen updates and can make the control or indicator flicker. Use the **Reorder** pull-down menu  to move objects relative to each other.

Complete the following steps to reorder objects that are stacked on top of each other.

1. Select the object you want to move.
2. Select the **Reorder** pull-down menu.
3. Select from the following options:
 - **Move Forward** —Moves the selected object one position higher in the stack.
 - **Move Backward** —Moves the selected object in the stack one position back in the stack.
 - **Move To Front** —Moves the selected object in the stack to the top.
 - **Move To Back** —Moves the selected object in the stack to the back.



NOTE

You also can *reroute all existing wires and rearrange all existing objects* on the block diagram automatically.

You also can use the following keyboard shortcuts to reorder objects in a stack.



Note The <Ctrl> key in these shortcuts corresponds to the (macOS) <Option> or <Command> key or to the (Linux) <Alt> key.

- **Move Forward** —<Ctrl+K>
- **Move Backward** —<Ctrl+J>
- **Move To Front** —<Ctrl+Shift+K>
- **Move To Back** —<Ctrl+Shift+J>

Replacing a Front Panel Object from the Clipboard

Complete the following steps to replace a front panel object with an object from the clipboard.

1. Use the Positioning tool to select the front panel object that will replace the existing control or indicator.
2. Select **Edit»Copy** to copy the object to the clipboard.
3. Use the Positioning tool to select the object that you want to replace.
4. Select **Edit»Paste**. LabVIEW pastes the new object in place of the old object.

**NOTE**

If you replace a control with an indicator, LabVIEW preserves the dataflow direction of the original object and does not *change the control to an indicator*. If you replace an indicator with a control, LabVIEW does not change the indicator to a control.

This method does not preserve any characteristics of the old object, but the wires remain connected to the object. *Right-click an object and select Replace from the shortcut menu* to preserve as much information as possible about the original object.

Replacing Front Panel Objects

You can replace a front panel object with a different control or indicator without losing the label, caption, and value you set for the object.

Complete the following steps to replace a front panel object.

1. Right-click a front panel object.
2. Select **Replace** from the shortcut menu to display a temporary version of the **Controls** palette.
3. Select a control or indicator on the temporary **Controls** palette to replace the current front panel object.

**NOTE**

If you replace a control with an indicator, LabVIEW preserves the dataflow direction of the original object and does not *change the control to an indicator*. If you replace an indicator with a control, LabVIEW does not change the indicator to a control.

You also can *replace a front panel object with another object from the clipboard*.

Scaling Front Panel Objects

You can set front panel objects to scale, or automatically resize in relation to the pane size, when you resize the pane. You can set one front panel object to scale, or you can set all objects on an owning pane to scale.

You also can set a front panel object to fit to a pane. LabVIEW resizes the object to fill the pane. When you resize the pane, the object scales, or resizes, with the pane.



RELATED INFORMATION

[Setting a Single Front Panel Object to Scale](#) on page 1381

[Setting All Objects on a Front Panel to Scale](#) on page 1381

[Fitting a Single Front Panel Object to a Pane](#) on page 1382

Setting the Tabbing Order of Front Panel Objects

The **tabbing order** determines the order in which LabVIEW selects controls and indicators when the user presses the <Tab> key while a VI runs. The tabbing order also determines the order in which the controls and indicators appear in the records of datalog files you create when you [log the front panel data](#). LabVIEW uses the order in which you add objects to the front panel or to the tab control as the default tabbing order.

Complete the following steps to change the tabbing order.

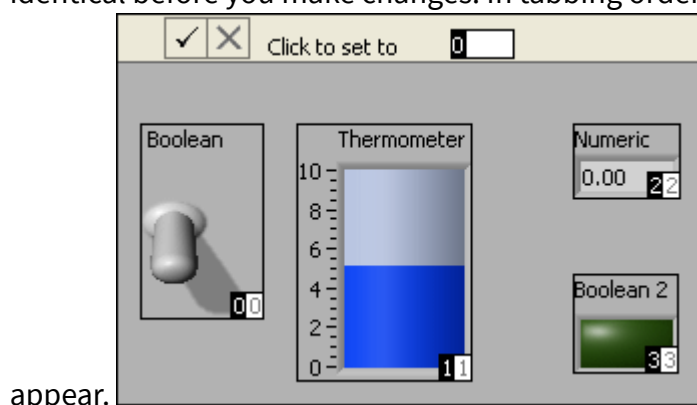
1. Select **Edit»Set Tabbing Order** to activate the tabbing order edit mode for the front panel.



NOTE

To set the tabbing order of controls and indicators on a **tab control** page, right-click the page and select **Reorder Controls In Page** from the shortcut menu.

The front panel appearance changes in tabbing order edit mode, as shown in the following illustration. The white boxes on the controls and indicators show the current tabbing order. Black boxes show the new tabbing order. These numbers are identical before you make changes. In tabbing order edit mode, all hidden controls



2. Enter a number in the **Click to set to** field on the toolbar. You cannot enter a number greater than the number of front panel or tab control controls and indicators.
3. Click an object with the tabbing order cursor to set the place number of the object as the number displayed in the **Click to set to** field on the toolbar. Notice that LabVIEW automatically rennumbers the other front panel or tab control objects based on the original tabbing order.

4. Repeat steps 2 and 3 until all elements are in the order you want.



Note LabVIEW skips hidden controls when you tab through a front panel. You do not need to set the order of hidden controls.

1. When the black boxes display the tabbing order you want, click the **Confirm** button on the toolbar to accept the new tabbing order and exit tabbing order edit mode. Click the **Cancel** button on the toolbar to revert to the old tabbing order and exit the tabbing order edit mode.
2. Run the VI and press the <Tab> key to navigate the front panel or tab control controls or indicators. Use the left and right arrow keys to navigate the pages of the tab control.



Note If you add a splitter bar to the front panel and create panes, the tabbing order of front panel objects does not change. A control in any pane can be the next control in the tabbing order.

Changing Values of Numeric Objects with a Scale

Complete the following steps to practice changing the value of a slide, scroll bar, or rotary object.

1. Use the Operating tool to click the slider, scroll box, or needle and drag it inside the slide, scroll bar, or rotary object. The value changes as you drag the slider, scroll box, or needle, and LabVIEW displays the value of the control in a value tip strip.
2. Click inside the slide, scroll bar, or rotary object, but not on the slider, scroll box, or needle. The slider or needle snaps to the value you click. Before you release the mouse button, LabVIEW displays the value of the control in a tip strip. The scroll box pages one page size. You can set the scroll bar page size on the Data Entry page of the **Scrollbar Properties** dialog box.



NOTE

When you click inside a knob or dial, the needle does not snap to the value you click. You must drag the needle to change the value.

You also can change the value of a slide or rotary object with a scale using the *digital display* just as you do for numeric *controls and indicators*.

Displaying Radixes on Numeric Objects

You can display a radix on a numeric object and use the radix to change the data to decimal, hexadecimal, octal, or binary format, or to SI notation.

Complete the following steps to display the radix on a numeric object and change the format of the data.

1. Right-click a numeric object and select **Visible Items»Radix** from the shortcut menu. A letter d, x, o, b, or p appears in the numeric display.
2. Use the Operating tool to click the letter and select **Decimal, Hex, Octal, Binary**, or **SI Notation** from the shortcut menu. If the representation of the numeric object is floating-point, you can select only **Decimal** or **SI Notation**.



NOTE

If the format of the numeric control changes, LabVIEW hides or changes the radix so it becomes compatible with the new format.

You also can use the Radix Visible? property to display the radix programmatically.

Controls and Indicators

Configuring Controls and Indicators

Configure the properties of controls and indicators to change how the objects appear or behave. You can use the Properties dialog box or the [shortcut menu](#) to configure properties.

To display the Properties dialog box, right-click the control or indicator and select **Properties** from the shortcut menu. You cannot display Properties dialog boxes while a VI runs.

To display the shortcut menu, right-click the control or indicator.



TIP

To perform a common operation on multiple objects simultaneously, select the objects and use the shortcut menu or the Properties dialog box.

The following options are common to all control and indicator shortcut menus:

- **Visible Items** —Lists the parts of an object you can choose to hide or show, such as the name label and the caption.
- **Find Terminal** — [Highlights the block diagram terminal](#) for the control or indicator. This menu item is useful for identifying a specific object on a complex block diagram.
- **Change to Indicator** and **Change to Control** — [Changes the current object to a control or indicator](#).
- **Description and Tip** —Displays a dialog box where you can [enter and view object descriptions and tips](#) for using the object.
- **Create** —Creates a [Local Variable](#), [Reference](#), [Property Node](#), or [Invoke Node](#) to control various properties of the object programmatically.

- **Replace** — *Replaces the front panel object* with a different control or indicator without losing all of the editing options you applied to the object up to that point.
- **Data Operations** —Contains a submenu of the following data editing options:
 - **Reinitialize to Default Value** —Sets the control to its default value. You also can use the Reinitialize To Default method to reinitialize a control to its default value programmatically.
 - **Make Current Value Default** —Makes the current value of the control its new default value. You also can use the Default Values:Make Current Default method to set the default value of all front panel controls programmatically.
 - **Cut Data, Copy Data, and Paste Data** —Cuts, copies, or pastes the contents of a front panel object. Because LabVIEW saves the data you cut or copied in the clipboard, you can paste the data outside LabVIEW.
- **Advanced** —Contains a submenu of the following advanced editing options:
 - **Synchronous Display** —Displays *every update*. Use this feature to display animations. You also can use the Synchronous Display property to set this option programmatically.
 - **Customize** —Displays the *Control Editor* window, which you can use to *customize front panel objects*.
 - **Hide Control** or **Hide Indicator** — *Hides the front panel object* from view. To access a hidden object, right-click the block diagram terminal and select **Show Control** or **Show Indicator** from the shortcut menu.
 - **Enabled State** —Determines if an object is enabled, disabled, or disabled and dimmed.

I/O Controls and Indicators

Use the I/O controls on the **I/O** and **Classic I/O** palettes with the functions for acquiring data, sharing data between applications, and controlling instruments. You can use the I/O controls to specify the instrument or device resource you want to communicate with.

- Use the waveform control to hold acquired data for analysis and display.
- Use the controls located on the **DAQmx Name Controls** palette to access the tasks, global channels, physical channels, terminals, scales, devices, and switch resources you configure using Measurement & Automation Explorer (MAX) and the DAQ Assistant.



NOTE

(Windows) LabVIEW supports NI-DAQmx and the DAQ Assistant. (Linux) LabVIEW supports NI-DAQmx but not the DAQ Assistant. (macOS) LabVIEW supports NI-DAQmx Base but not the DAQ Assistant.

- Use the VISA resource name and IVI logical name controls to access the VISA aliases or logical names you configure using MAX.
- Use the FieldPoint I/O point control to access the FieldPoint items you create and configure using MAX.
- Use the IMAQ session with image acquisition boards.

- Use the Motion resource I/O control to access a motion resource you configure using MAX. Any items you configure in MAX appear as options in the I/O control pull-down menu.
- Use the shared variable control to select, browse to, or enter shared variables for use in VIs.
- Use any of the other controls in the palette to access names or resources associated with them.

After you select an I/O control, **configure** it for use with the I/O VIs that communicate with the device or instrument.



NOTE

Many I/O name controls are available on all platforms. You can develop I/O VIs on any platform that can communicate with devices that are platform-specific. However, if you try to run a VI with a platform-specific I/O control on a platform that does not support that device, you will receive an error.

Some of the controls allow you to select multiple defined names. Use the Operating tool to click the control, select **Browse** from the pull-down menu, and press the <Ctrl> key. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.

Refer to the respective **toolkit, module, or driver** documentation for more information about LabVIEW add-ons.

Variant Control and Indicator

Use the variant control and indicator to pass data to or from subVIs. You also can use it to display but not modify the data. Use the **variant data type** to modify variant data.

Boolean Controls

Use the Boolean controls located on the **System, Classic, Express, and Modern** palettes to create user interfaces.

Configuring I/O Controls

Configure I/O controls to pass Traditional NI-DAQ (Legacy) virtual channels, NI-DAQmx resources (such as tasks, channels, and scales), IVI logical names, VISA resource names, FieldPoint I/O points, or Motion resource names to I/O VIs to communicate with an instrument or a measurement device.

1. Add one of the I/O controls from the **I/O** palette to the front panel.
2. Click the arrow in the I/O control and select a resource you configured. If the I/O control does not have items available in the pull-down menu, you must **configure the resource** associated with the control.

3. Display the block diagram.
4. Wire the I/O terminal to the I/O input of an I/O VI that communicates with a measurement device, instrument, or other device.

You also can configure the shared variable control to select a shared variable for use in VIs.

1. Add the shared variable control from the **I/O** palette to the front panel.
2. Click the down arrow on the right side of the shared variable control to select a shared variable. All deployed shared variables in the currently open project appear in the pull-down menu. If you want to use a shared variable that is not in the currently open project, click the down arrow and select **Browse**. Browse for a deployed shared variable in a different project or on a different computer.
3. Display the block diagram.
4. Wire the shared variable control to the **connection in** input of DataSocket Read and DataSocket Write functions.

Refer to the respective toolkit, module, or driver documentation for more information about LabVIEW add-ons.

Displaying All Hidden Controls and Indicators

You can display all hidden front panel controls and indicators of *custom controls* and global variables by selecting **Edit»Show Hidden Controls and Indicators**. This option is available only in the Edit menu of a custom control or global variable VI. Complete the following steps to display controls and indicators for VIs that are not custom control or global variable VIs.



NOTE

You cannot use this option on locked VIs or VIs reserved for running.

1. Open the ShowHidden Core VI in the `labview\project\_ShowHidden` directory. You must open the ShowHidden Core VI in the same *application instance* as the VI that contains the controls and indicators you want to display.
2. Enter the name of the target VI in the **VI Name** text box of the ShowHidden Core VI.
3. Click the **Run** button.

All hidden front panel controls and indicators of the target VI appear, and the **Ctl Labels** indicator of the ShowHidden Core VI displays the labels of the controls and indicators shown.

Displaying Scroll Bars of Controls and Indicators

Right-click a control or indicator and select **Visible Items»Scrollbar**, **Visible Items»Vertical Scrollbar**, or **Visible Items»Horizontal Scrollbar** from the shortcut menu

to display a vertical scroll bar or horizontal scroll bar. Use the Positioning tool to resize the front panel object so you can minimize the amount of space the object occupies.



NOTE

You can display scroll bars on a string or path control or indicator only if the object has a height and width of at least two scroll bars. You must *disable word wrapping* in a string control or indicator to display a horizontal scroll bar.

You also can use the Scrollbars Visible, Visible Items:Vertical Scrollbar Visible, or Visible Items:Horizontal Scrollbar Visible properties of a control or indicator to display a scroll bar programmatically.

Only arrays with more than one dimension can have both a vertical and horizontal scroll bar. Resize a 1D array vertically to add a vertical scroll bar, and horizontally to add a horizontal scroll bar.

Hiding Front Panel Controls and Indicators

Complete the following steps to hide a front panel control or indicator. You also can *display hidden front panel objects*.

1. Right-click the front panel object you want to hide.
2. Select **Advanced»Hide Control** from the shortcut menu if the object is a control or **Advanced»Hide Indicator** if the object is an indicator.
3. (Optional) Specify the behavior of hidden controls during editing.

Task

Select **Edit»Show Hidden Controls While Editing**.

Select **Edit»Hide Hidden Controls While Editing**.

You also can use the Visible property to hide controls and indicators programmatically.

Customizing Controls and Indicators that are Parts of Controls and Indicators

A control or indicator can include other controls or indicators as parts. In this topic, controls and indicators used as parts of other controls and indicators are referred to as component controls and indicators. For example, a slide control uses a digital numeric control for a digital display, making the digital numeric control a component control. You can find out if part of a control or indicator is a component control or indicator by trying to move the part around the front panel. You can move a component control or indicator independently of the parent control or indicator.

Often, component controls or indicators also are made up of parts. When you edit the original control or indicator in the *Control Editor* window, the component control or

indicator behaves as a single part. That is, you cannot change or move individual parts of the component control or indicator. However, you can customize the component control or indicator by displaying a second Control Editor window that contains only the component control or indicator. You do not have to place the original Control Editor window in customize mode to display a second Control Editor window unless you are unable to select the component control or indicator in edit mode.

Complete the following steps to customize a component control or indicator.

1. In the front panel window, right-click a control or indicator that you want to customize and select **Advanced»Customize** from the shortcut menu to display the Control Editor window.
2. Right-click the component control or indicator in the Control Editor window and select **Advanced»Customize** from the shortcut menu to display another Control Editor window that contains only the component control or indicator.
3. Click the **Change to Customize Mode** button to place the new Control Editor window in customize mode.
4. Make the changes you want to the individual parts of the component control or indicator. You can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.
5. Select **File»Apply Changes** to apply the changes to the component control or indicator in the original Control Editor window. The **Apply Changes** menu item is available only after you make changes to the original component control or indicator. If you make changes to a component control or indicator and try to close the Control Editor window without selecting **File»Apply Changes**, LabVIEW displays a message asking if you want to replace the original component control or indicator with the custom component control or indicator.

You can nest Control Editor windows indefinitely. However, most controls or indicators use other controls or indicators as parts only at the top level. An exception is the graph, which uses complicated controls as parts that in turn use other controls as parts.

Component Parts Visible in the Front Panel Window

If a component control or indicator is visible in the front panel window, you can display a Control Editor window that contains only the component control or indicator directly from the front panel window.

Complete the following steps to customize a component control or indicator directly from the front panel window.

1. Right-click the component control or indicator you want to customize and select **Advanced»Customize** from the shortcut menu to display a Control Editor window that contains only the component control or indicator. You also can use the Positioning tool to select the component control or indicator you want to customize and then select **Edit»Customize Control** to display the Control Editor window.
2. Make the changes you want to the component control or indicator. You can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.

3. Select **File»Apply Changes** to apply the changes to the component control or indicator. The **Apply Changes** menu item is available only after you make changes to the original component control or indicator. If you make changes to a component control or indicator and try to close the Control Editor window without selecting **File»Apply Changes**, LabVIEW displays a message asking if you want to replace the original component control or indicator with the custom component control or indicator.

Adding Custom Controls and Indicators to Front Panels and Block Diagrams

You can add *custom controls and indicators* to front panels and block diagrams by dragging them from the *Control Editor* window or by selecting them from the **Controls** palette or the **Functions** palette. Adding a custom control or indicator to a block diagram creates a constant with the same data type as the custom control or indicator. You also can *add custom controls and indicators to the Controls and Functions palettes*.

Complete the following steps to select a custom control or indicator and add it to a front panel or block diagram from the **Controls** or **Functions** palette.

1. Click the **Select a Control** icon on the **Controls** palette or the **Select a VI** icon or text on the **Functions** palette to display a file dialog box.
2. Navigate to the custom control or indicator you want to place.
3. In the file dialog box, double-click the custom control or indicator or click the **OK** button to place the control or indicator on the cursor.
4. Add the custom control or indicator to the front panel or block diagram.
5. Save the VI with the custom control or indicator in place.



NOTE

When you add a custom control or indicator to a VI, no connection exists between the custom control or indicator you saved and the instance of the custom control or indicator in the VI. Each instance of a custom control or indicator is a separate, independent copy. Therefore, changes you make to a custom control or indicator file do not affect VIs already using that custom control or indicator. If you want to link instances of a custom control or indicator to the custom control or indicator file, save the custom control or indicator as a *type definition or a strict type definition*. All instances of a type definition or a strict type definition link to the original file from which you created them. When you save a custom control or indicator as a type definition or strict type definition, any data type changes you make to the type definition or strict type definition affect all instances of the type definition or strict type definition in all the VIs that use it. Also, cosmetic changes you make to a strict type definition affect all instances of the strict type definition.

Changing Data Ranges of Numeric Controls and Indicators

You can limit intermediate values of numeric controls or indicators to certain increments. For example, you might limit 16-bit integers to increments of 10 or single-precision, floating-point numbers to increments of 0.25. If you change either the limits or the increment, you also should decide what to do if the user attempts to set a value outside the range or off the increment. You also can use the In Range and Coerce function to determine if a value falls outside of a range and coerce the value to fall within the range.

Complete the following steps to designate other limits within the *natural minimum and maximum data range limits*.



NOTE

Use the fields described below to set how to handle the numeric value the user enters if that value is outside of the specified numeric range. Changing these fields does not affect values that are set programmatically, including the values of indicators.

1. Right-click a numeric control and select **Data Entry** from the shortcut menu to display the Data Entry page of the **Numeric Properties** dialog box.
2. Remove the checkmark from the **Use Default Limits** checkbox.
3. In the **Minimum** field, set the low value limit and select **Ignore** or **Coerce** from the pull-down menu in the **Response to value outside limits** section.
4. In the **Maximum** field, set the high value limit and select **Ignore** or **Coerce** from the pull-down menu in the **Response to value outside limits** section.
5. In the **Increment** field, set the incremental value and select **Ignore**, **Coerce to nearest**, **Coerce up**, or **Coerce down** from the pull-down menu in the **Response to value outside limits** section.

Response to value outside limits	Description
LabVIEW does not change or flag invalid values. Clicking the increment and decrement buttons changes the value by the increment you set, but the value does not go beyond the minimum or maximum values.	
LabVIEW increments to the nearest value. For example, if the minimum is 3, the maximum is 10, and the increment is 2, valid values are 3, 5, 7, 9, and 10. LabVIEW coerces the value 0 to 3, the value 6 to 7, and the value 100 to 10.	
LabVIEW increments the value by one increment value.	
LabVIEW decrements the value by one increment value.	

You also can use the Numeric properties or AbsTime properties to set the data range and response to values outside limits programmatically.

Changing Fill Options for Slide Controls and Indicators

Vertical slides, horizontal slides, the tank, and the thermometer fill from the minimum value to the slider value by default.

Complete the following steps to change the fill options of a slide control or indicator.

1. Right-click the slider.
2. Select **Fill Options** from the shortcut menu and select from the following options:
 - **Fill To Minimum** —Fills from the minimum value to the slider value. This is the default.
 - **Fill To Maximum** —Fills from the slider value to the maximum value.
 - **Fill To Value Below** —Available for slide objects with *more than one slider*. Fills from the slider you right-clicked to the slider below. If a slider does not exist below the slider you right-clicked, this option fills to the minimum value.
 - **Fill To Value Above** —Available for slide objects with *more than one slider*. Fills from the slider you right-clicked to the slider above. If a slider does not exist above the slider you right-clicked, this option fills to the maximum value.
 - **No Fill** —Eliminates the fill.

You also can use the Fill Style property to change the fill options programmatically.

Changing Format and Precision of Numeric Controls and Indicators

By default, LabVIEW displays and stores numbers like a calculator. A numeric control or indicator displays up to six digits before automatically switching to exponential notation. You can configure the number of digits for the control or indicator, digital displays, and tip strips that LabVIEW displays before switching to exponential notation in the Display Format page of the **Numeric Properties** dialog box.

Complete the following steps to change the format and precision of a numeric object.

1. Right-click the numeric object and select **Display Format** from the shortcut menu to display the Display Format page of the **Numeric Properties** dialog box.
2. Select a format from the **Type** listbox. If the *representation* of the numeric object is floating-point, you cannot display the number as hexadecimal, octal, or binary.

3. Use the **Digits** and **Precision Type** options to change the precision of the number, if necessary. The precision you select affects only the display of the value. The internal accuracy still depends on the representation.

**NOTE**

Select **Advanced editing mode** to display the text options that let you *enter format strings* directly. Enter format strings to customize the numeric precision of the control.

1. To change the format and precision of the digital display, select **Digital Display** from the **Apply to** pull-down menu and repeat steps 2 and 3. The tip strip uses the same format and precision as the digital display of the control. If the control has more than one digital display, the tip strip uses the format and precision of the digital display that corresponds to the currently active needle or slider. The **Apply to** pull-down menu appears only for items that include a digital display.
2. Click the **OK** button.

You also can use the Digital properties to configure format and precision programmatically.

Importing Graphics into Boolean Controls and Indicators

You can use the *Control Editor* window to import graphics into Boolean controls and indicators to indicate the TRUE, FALSE, and transition states of the control or indicator.

Importing Graphics into a Boolean Control or Indicator with the Control Editor Window in Edit Mode

Complete the following steps to import graphics into a Boolean control or indicator with the Control Editor window in edit mode.

1. Right-click a front panel Boolean control or indicator you want to customize.
2. Select **Advanced»Customize** from the shortcut menu to display the Control Editor window. The Control Editor window opens in edit mode.
3. Select **Edit»Import Picture to Clipboard** to display a file dialog box.
4. Navigate to the graphic you want to import.
5. Select the graphic and double-click the graphic or click the **OK** button to place the graphic on the clipboard.
6. Right-click the Boolean control or indicator, select **Import Picture from Clipboard** from the shortcut menu, and select among the following items:
 - **False** —Applies the imported graphic to the FALSE state of the Boolean control or indicator. The imported graphic appears only when the control or indicator is in the FALSE state.

- **True** —Applies the imported graphic to the TRUE state of the Boolean control or indicator. The imported graphic appears only when the control or indicator is in the TRUE state.
- **Decal** —Applies the imported graphic to all states of the Boolean control or indicator. The graphic remains visible regardless of the state of the Boolean control or indicator.

LabVIEW imports the graphic into the normal state and into the corresponding transition state.



TIP

To undo changes to a Boolean control or indicator after you import graphics and save the VI, you can replace the control or indicator with a new version from the palettes. Display the front panel, right-click the Boolean control or indicator, and select **Replace** from the shortcut menu. Then select the Boolean control or indicator you want to use as a replacement.

Importing Graphics into a Boolean Control or Indicator with the Control Editor Window in Customize Mode

Complete the following steps to import graphics into a Boolean control or indicator with the Control Editor window in customize mode.

1. Right-click a front panel Boolean control or indicator you want to customize.
2. Select **Advanced»Customize** from the shortcut menu to display the Control Editor window.
3. Click the **Change to Customize Mode** button to place the Control Editor window in customize mode.
4. Right-click the Boolean control or indicator, select **Picture Item** from the shortcut menu, and select among the following items:
 - The first selection applies the graphic to the FALSE state.
 - The second selection applies the graphic to the TRUE state.
 - The third selection applies the graphic to the *Switch When Released* state, which is the TRUE to FALSE transition state.
 - The fourth selection applies the graphic to the *Latch When Released* state, which is the FALSE to TRUE transition state.



NOTE

Some system controls, such as the system button and the system cancel button, offer additional selections that represent additional possible states.

1. Right-click the Boolean control or indicator and select **Import from File** from the shortcut menu to display a file dialog box. You also can select **Import from File at Same Size** from the shortcut menu to import the graphic at the same size as the selected part of the Boolean control or indicator.
2. Navigate to the graphic you want to import.

3. Select the graphic and double-click the graphic or click the **OK** button to replace the selected part of the Boolean control or indicator with the graphic.

Creating Custom Controls and Indicators from the Front Panel

Complete the following steps to create a custom control or indicator from the front panel. You can edit only one control or indicator at a time. Refer to the [caveats and recommendations](#) before you begin creating custom controls and indicators.

1. Use the Positioning tool to select a control or indicator you want to customize.
2. Select **Edit»Customize Control** to display the Control Editor window.



NOTE

You also can right-click the control or indicator you want to customize and select **Advanced»Customize** from the shortcut menu to display the [Control Editor](#) window.

3. Make the changes you want to the control or indicator. You can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.
4. Select **File»Apply Changes** to apply the changes to the control or indicator. The **Apply Changes** menu item is available only after you make changes to the original control or indicator. If you make changes to a control or indicator and try to close the Control Editor window without selecting **File»Apply Changes**, LabVIEW displays a message asking if you want to replace the original control or indicator with the custom control or indicator.
5. (Optional) Select **File»Save** to save the custom control or indicator so you can use it in other VIs. You can save the custom control or indicator in a directory or in an [LLB](#). If you do not want to save the custom control or indicator for use in other VIs, close the Control Editor window without saving the custom control or indicator.

You also can create a custom control or indicator [using the New dialog box](#).



Note It is not necessary to save a custom control to replace the original control. If the control you customized is a [type definition](#) and you replace the original control, LabVIEW prompts you to save the type definition when you close the calling VI.

Editing Saved Custom Controls and Indicators

Complete the following steps to edit a saved custom control or indicator.

1. Select **File»Open** to display a file dialog box.
2. Navigate to the custom control or indicator you want to edit.

3. Select the custom control or indicator and double-click the custom control or indicator or click the **Open** button to display the custom control or indicator in the **Control Editor** window.
4. Make the changes you want to the custom control or indicator. You can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.
5. Select **File»Save** to save the custom control or indicator.



NOTE

When you place a custom control or indicator in a VI, no connection exists between the custom control or indicator you saved and the instance of the custom control or indicator in the VI. Each instance of a custom control or indicator is a separate, independent copy. Therefore, changes you make to a custom control or indicator file do not affect VIs already using that custom control or indicator. If you want to link instances of a custom control or indicator to the custom control or indicator file, save the custom control or indicator as a **type definition or a strict type definition**. All instances of a type definition or a strict type definition link to the original file from which you created them. When you save a custom control or indicator as a type definition or strict type definition, any data type changes you make to the type definition or strict type definition affect all instances of the type definition or strict type definition in all the VIs that use it. Also, cosmetic changes you make to a strict type definition affect all instances of the strict type definition.

Caveats and Recommendations when Creating Custom Controls and Indicators

The following list describes some of the caveats and recommendations to consider when you create custom controls and indicators:

- Graphics you create on one platform look slightly different when you load them on another platform. For example, a graphic with an irregular shape on a transparent background might have a solid white background on **another platform**.
- Customizing a control or indicator in the **Control Editor** window changes only the appearance of the control or indicator, not the behavior of the control or indicator. You cannot use the Control Editor window to change the way a control or indicator displays data. You also cannot use the Control Editor window to change the way a control or indicator behaves when you edit it, such as when you resize it. For example, when you make a ring control taller, the increment and decrement arrows also increase in height. If you move the increment and decrement arrows so they are side by side at the bottom of a ring control, the buttons become taller when the ring control becomes taller.
- If you experience irregular editing behavior using a custom control or indicator, use a **strict type definition** to correct this error.
- Controls and indicators may customize differently depending on the style of the control or indicator. For example, when you change the color of a modern style numeric control, the color of the increment and decrement buttons change as well. On a silver style numeric control, the color of the increment and decrement buttons do not change.

Controls

Setting Keyboard Shortcuts for Controls

You can assign keyboard shortcuts to controls so users can navigate the front panel window without a mouse. You can assign modifiers, such as the <Shift> and <Ctrl> keys, to some shortcut keys. You cannot assign the same key combination to more than one control.

LabVIEW does not respond to keyboard shortcuts for hidden controls.

Complete the following steps to set key navigation for a control and to set a control so it is not included when the user tabs from control to control while running the VI.

1. Right-click a front panel control and select **Advanced»Key Navigation** from the shortcut menu to display the **Key Navigation** page of the *Properties* dialog box. You also can right-click a control, select **Properties** from the shortcut menu, and click the **Key Navigation** tab. To set key navigation for an array control, right-click the array border. To set key navigation for an array element, right-click the element.



NOTE

The **Advanced»Key Navigation** shortcut menu item is dimmed for indicators because you cannot enter data in an indicator.

1. From the pull-down menu in the **Focus** section, select the shortcut key you want to assign to the control. If you assign <PageUp>, <PageDown>, <Home>, or <End> as a shortcut key in the **Focus** section, you no longer can use that key to select items in listboxes or tree controls.
2. To include the <Shift> key and/or <Ctrl> key in a keyboard shortcut, place a checkmark in the **Shift key** and/or the **Control key** checkboxes. These checkboxes are available if you select a function key, such as <F1>, <F2>, and so on, in the **Focus** pull-down menu.



Note (macOS) Keyboard shortcuts for the function keys work only if the function keys are not assigned as shortcut keys for Exposé. LabVIEW does not support <VolumeUp>, <VolumeDown>, and function keys <F15> to <F24> as shortcut keys on macOS.

1. To prevent users from accessing a control using the <Tab> key while the VI is running, place a checkmark in the **Skip this control when tabbing** checkbox.



Note You also can use the Key Down event to generate an event when the user presses a specific key on the keyboard.

You can assign keyboard shortcuts to toggle Boolean controls using the **Toggle** section in the **Key Navigation** page, or to increment and decrement Numeric, Slide, Knob, Ring, and Enum controls using the **Increment** and **Decrement** sections in the **Key Navigation** page. The **Toggle** section is available only for Boolean controls, and the **Increment** and **Decrement** sections are available only for Numeric, Slide, Knob, Ring, and Enum controls.

You also can use the Control properties to assign keyboard shortcuts to controls and set tabbing behavior programmatically.

Container Controls

Use the container controls located on the **Containers** and **Classic Containers** palettes to group controls and indicators or to display the front panel of another VI on the front panel of the current VI. (Windows) You also can use container controls to display .NET Framework and ActiveX objects in the front panel window. LabVIEW does not support .NET 8.0 (Core) objects on the front panel.



RELATED INFORMATION

[Creating Tab Controls](#) on page 1065

[Loading a Front Panel in a Subpanel Control](#) on page 1062

[Creating .NET Controls in a .NET Container](#) on page 1481

[ActiveX Container Overview](#) on page 1490

Configuring Subpanel Controls

Right-click a subpanel control and select from the following items on the shortcut menu to configure the subpanel control:

- **Scale Object with Pane** —Scales the object to fit the size of the *pane* as the pane resizes. This option is not available if another object is currently set to scale with the pane or the pane is set to scale all objects with the pane. You also can use the Set Scaling Mode method to configure this option programmatically.
- **Make Panel Transparent** —Draws the background of the front panel in the subpanel control as transparent. You also can use the Panel Transparent property to configure this option programmatically.
- **Allow User to Open Diagram** —Displays an **Open Block Diagram** item on the shortcut menu when the user right-clicks either of the following areas:
 - On the border of a subpanel control in a running VI
 - Within the border of a subpanel control but not on a front panel control or indicator loaded in a subpanel control in a running VI

You also can use the Allow User to Open Diagram property to configure this option programmatically.



NOTE

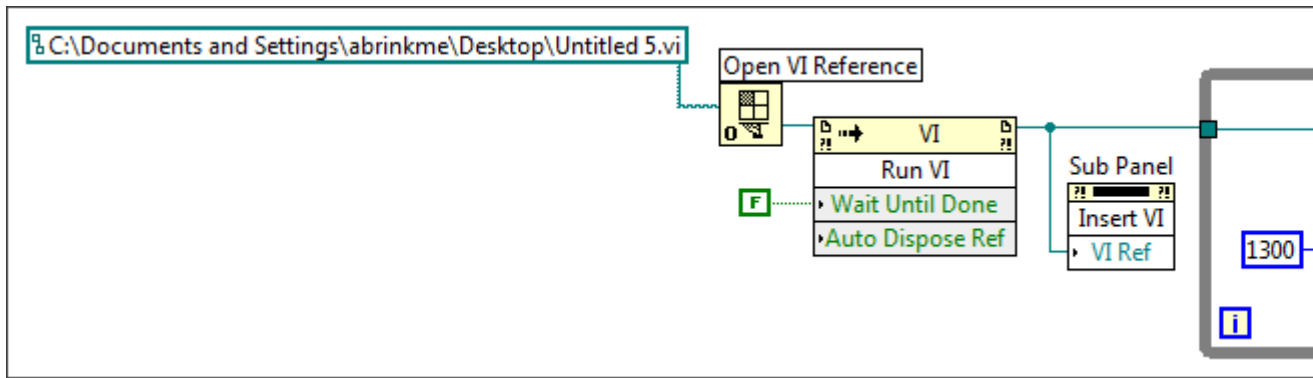
Use the pane of a subVI in a subpanel to configure the visibility of scroll bars for subpanel controls and to scale the front panel in subpanel controls.

Loading a Front Panel in a Subpanel Control

Use the subpanel control to display the front panel of another VI on the front panel of the current VI. For example, you can use a subpanel control to design a user interface that behaves like a wizard. Add the **Back** and **Next** buttons to the front panel of the top-level VI and use a subpanel control to load different front panels for each step of the wizard.

Complete the following steps to load a front panel in a subpanel control.

1. Add a subpanel control to the front panel. LabVIEW creates an Invoke Node on the block diagram with the Insert VI method selected.
2. Add the Open VI Reference function to the left of the Invoke Node on the block diagram.
3. Wire the path of the VI whose front panel you want to load to the **vi path** input of the Open VI Reference function.
4. Right-click the **vi reference** output of the Open VI Reference function and select **Create»Method for VI Class»Run VI** from the shortcut menu and add the Invoke Node that appears on the cursor to the right of the Open VI Reference function.
5. Wire the **vi reference** output of the Open VI Reference function to the **vi reference** input of the Run VI method.
6. Right-click the **Wait Until Done** input of the Run VI method and select **Create»Constant** from the shortcut menu to create a Boolean constant with a value of FALSE.
7. Wire the **vi reference** output of the Run VI method to the **VI Ref** input of the Insert VI method.
8. Add a While Loop to the right of the Insert VI method.
9. In the While Loop, right-click the conditional terminal and select **Create Control** from the shortcut menu to create a stop button.
10. Add the Wait Until Next ms Multiple function inside the While Loop and wire a numeric constant to the **millisecond multiple** input. Specify the number of milliseconds you want to lapse when the VI runs.
11. Add the Close Reference function to the right of the While Loop.
12. Wire the **vi reference** output of the Run VI method through the While Loop and to the **reference** input of the Close Reference function. Refer to the following block diagram for an example of loading a front panel in a subpanel control.



13. **Wire all error in and error out terminals** on the block diagram.

You also can use the SubPanel properties and methods to modify subpanel controls programmatically.

Refer to the `labview\examples\Controls and Indicators\Containers\Containers.lvproj` for examples of using subpanel controls.

Tab Controls

Adding Pages to Tab Controls

To add additional pages to a tab control, right-click a tab and select **Add Page Before** or **Add Page After** from the shortcut menu.

Configuring Options for Tab Controls

Configure tab controls by right-clicking a tab on the tab control and selecting from items on the shortcut menu.

Complete the following steps to configure a tab control.

1. Right-click a tab on the tab control.
2. Select from the following shortcut menu items:



NOTE

This is not a complete list of menu items you see when you right-click a tab on a tab control.

- **Reorder Controls In Page** — *Reorders the objects* on the active page for key navigation. LabVIEW disables this item if there are not at least two controls on the page.
- **Advanced** — Select from the following items:

- **Hide Page** —Hides the current page. If you hide the default page, the next page becomes the default page. If the only visible page is the default page, you cannot hide the default page. You also can use the Page Visible property to hide a page of the tab control programmatically.
- **Page Enabled State** —Modifies the enabled state of a page of the tab control. Select from **Enabled**, **Disabled**, and **Disabled & Grayed**. You also can use the Page Enabled State property to modify the enabled state of a page of the tab control programmatically.
- **Show Hidden Page** —Shows a specific hidden page by name. You also can use the Page Visible property to show a hidden page of the tab control programmatically.
- **Make Page Caption Match Label** —Remove the checkmark next to this option to change the tab text without changing the page label associated with the page. To display the page label, right-click the tab and select **Visible Items»Page Labels Display** from the shortcut menu. Page labels correspond to the values of the enumerated type control on the block diagram.
- **Auto Grow** —Increases the tab control size if an object you place on a page is outside the bounds of the tab control.
- **Size To Fit** —The tab control resizes to fit its contents. If you enable this shortcut item, LabVIEW enables the **Auto Grow** item automatically.
- **Allow Multiple Colors** —Allows different colors on the individual pages of a tab control.
- **Allow Multiple Rows** —Arranges any tabs that extend beyond the width of the tab control into an additional row of tabs. Remove the checkmark next to this option to display navigation buttons on the upper right of the tab control. Use these navigation buttons to scroll through tabs that are not visible.
- **Tab Size** —Adjusts the size of the tabs. Select from **Size Tabs To Contents**, **Size All Tabs To Largest Tab**, or **Fixed Tab Size**. If you select **Fixed Tab Size**, use the Positioning tool to resize the individual tabs. Place a checkmark by **Justify Tabs** to stretch each row of tabs to fill the entire width of the tab control. If the tab control has multiple rows of tabs, LabVIEW automatically justifies the tab size.
- **Tab Layout** —Changes whether you can use text and/or images on the tabs of each tab control. You also can use this shortcut menu to add and remove images from the tabs.
- **Tab Location** —Changes the location of the tabs on the tab control.

You also can use the TabControl and Page properties to configure tab controls and individual tab control pages programmatically. You must obtain a reference to a tab control page before you can [configure an individual page](#).

Configuring Tab Control Pages Programmatically

Complete the following steps to configure a tab control page programmatically.

1. Right-click a tab of a tab control and select **Create»Property Node»Pages** from the shortcut menu. LabVIEW creates an implicitly linked Property Node on the block diagram for the property you selected.
2. Add the Index Array function to the block diagram.
3. Wire the **Pages** output of the Property Node to the **n-dimension array** input of the Index Array function.
4. Right-click the **Index** input of the Index Array function and select **Create»Constant** from the shortcut menu to create a numeric constant.
5. Type the index of the page you want to configure and press the <Enter> key. The index corresponds to the order of the pages as they appear on the tab control, in the range 0 to $n-1$, where n is the number of pages on the tab control. For example, type 0 to configure the first page in the tab control.
6. Use the Positioning tool to *resize* the Index Array function if necessary.
7. Use the Wiring tool to right-click an **element** output of the Index Array function and select **Create»Property for Page Class** and a property from the shortcut menu. LabVIEW creates a Property Node for the property you selected.
8. Add the Property Node to the block diagram and wire the **element** output of the Index Array function to the **reference** input of the Property Node you created in step 8.
9. Configure the Property Node if necessary.

After you obtain a reference to a tab control page, you also can configure a control within a tab control page programmatically using the Controls on Page property.

Refer to the Programatically Manipulate a Tab Control VI in the `labview\examples\Controls and Indicators\Containers` directory for an example of configuring tab control pages programmatically.

Creating Tab Controls

You can use tab controls to overlap front panel controls and indicators in a smaller area. A tab control consists of pages and tabs. Add front panel objects to the pages of the tab control and use the tab as the selector to display each page.

Complete the following steps to create a tab control.

1. Add a tab control to the front panel window.
2. To switch between pages, use the Operating tool to click the tab for the page you want. You also can switch between pages by right-clicking a tab and selecting **Swap Pages** or **Go To Other Page** from the shortcut menu.
3. Add any front panel objects to the active page. A page is active when the tab for that page is flush with the page and the objects on the page are visible. Terminals for

controls and indicators you add to the tab control appear as any other block diagram terminal.



TIP

When you add objects to a tab control page, frequently go to the block diagram window and arrange the newly added terminals. When you add several objects to a tab control, the terminals on the block diagram can become cluttered.

On the block diagram, the tab control is an enumerated control or indicator. As a control, you can pass the value of the active page to other block diagram nodes. As an indicator, you can wire nodes to control which page is displayed. You do not need to wire the tab control terminal for the tab control to operate.



Tip You can wire the enumerated control terminal of the tab control to the selector of a Case structure to produce cleaner block diagrams. With this method, you associate each page of the tab control with a subdiagram in the Case structure. You add the control and indicator terminals from each page of the tab control—as well as the block diagram nodes and wires associated with those terminals—into the subdiagrams of the Case structure. If you need a component of a VI to run continuously, even when its controls are hidden, do not use a Case structure, because the structure executes only one case at a time.

Displaying Images in Tab Controls

Complete the following steps to configure a tab control to display an image.

1. Import an image to LabVIEW by selecting **Edit»Import Picture to Clipboard**.
2. Set the tab layout to display an image by right-clicking the tab and selecting **Advanced»Tab Layout»Image Only**.



NOTE

You can set the tab layout to display an image and text by selecting **Advanced»Tab Layout»Image – Text**.

3. Import the image from the clipboard to the tab by right-clicking the tab and selecting **Advanced»Tab Layout»Import Image from Clipboard**.

You can remove an image from a tab by right-clicking the tab and selecting **Advanced»Tab Layout»Remove Image from Tab**.

Duplicating Pages on Tab Controls

To create a duplicate of a page, right-click the tab of the page you want to duplicate and select **Duplicate Page** from the shortcut menu.

Making a Control Visible on All Pages of a Tab Control

You can add a control to multiple pages of a **tab control** that will be visible on all the pages of the tab control.

Complete the following steps to add a control to multiple pages of a tab control.

1. Create a **multi-paged tab control** in the front panel window.
2. Create a control in an empty space on the front panel.
3. Select the control you just created. Using the arrow keys, move the control onto the tab control.



NOTE

The control will display a drop shadow while the VI is in edit mode, indicating that it is not actually contained by the tab control. The drop shadow will not be displayed when the VI is running.

Removing Pages from Tab Controls

To remove a page and its contents, right-click the tab for the page you want to remove and select **Remove Page** from the shortcut menu. Removing a page also deletes all the front panel objects on the page.

Reordering Pages on Tab Controls

To reorder pages, right-click a tab and select **Rearrange Pages**—or **Swap Pages** if there are only two pages—from the shortcut menu to access the Rearrange Pages dialog box.

Refnum Controls

Refnum Controls

Use the reference number controls located on the **Refnum** and **Classic Refnum** palettes to work with files, directories, devices, and network connections. Use the **different types** of refnum controls when you want to pass a refnum into or out of a VI.

Because a refnum is a temporary pointer to an open object, it is valid only for the period during which the object is open. If you close the object, LabVIEW disassociates the refnum from the object, and the refnum becomes obsolete. If you open the object again, LabVIEW creates a new refnum that is different from the first refnum. LabVIEW allocates memory for an object that is associated with a refnum. Close the refnum to release the object from memory.

LabVIEW remembers information associated with each refnum, such as the current location for reading from or writing to the object and the degree of user access, so you can perform concurrent but independent operations on a single object. If a VI opens an object multiple times, each open operation returns a different refnum. LabVIEW automatically closes refnums for you when a VI finishes running, but it is a good programming practice to close refnums as soon as you are finished with them to most efficiently use memory and other resources. Close refnums in the opposite order that you opened them. For example, if you obtain a refnum to object A and invoke a method on object A to obtain a refnum to object B, close the refnum to object B first and then close the refnum to object A.

Creating Strictly Typed Refnums

You can specify the data type of any [refnum control](#), including an application, a VI, a control or indicator, or a strictly typed VI. Strictly typed VI refnums have data type information that includes the connector pane of the VI. You can use strictly typed VI refnums to call a dynamically loaded VI using the Call By Reference node or the Start Asynchronous Call node.

Complete the following steps to specify the type of a strictly typed VI refnum control.

1. Right-click the refnum control.
2. Select **Select VI Server Class»Browse** from the shortcut menu to select a VI from disk that has the connector pane you want.

You also can drag a VI icon, connector pane, or a subVI icon from a block diagram or the VI Hierarchy window to a VI refnum control.

Switching Between Strictly Typed and Weakly Typed Control Refnums

Strictly typed control refnums accept only control refnums of exactly the same data type. [Weakly typed control refnums](#) are more flexible in the type of data they accept.

There are five ways to switch between a strictly typed control refnum and a weakly typed control refnum.

- Right-click a strictly typed control refnum and remove the checkbox from the **Include Data Type** item in the shortcut menu to make it weakly typed.
- Right-click a weakly typed control refnum and select **Include Data Type** from the shortcut menu to make it strictly typed. This method does not work when the class of data is too general, such as GObject or General.
- Right-click a strictly typed control refnum and select **Select Ctrl Type** from the shortcut menu to change the class and disable the **Include Data Type** option, which makes the control refnum weakly typed.
- Right-click a strictly typed control refnum and select **Select VI Server Class** from the shortcut menu to select a generic control class, such as **Generic**, **GObject»GObject**, or **GObject»Control»Control**. Generic control classes cannot be strictly typed.

- Drag a control into the front panel control refnum control to remove the original control and create a strictly typed control refnum. A red star in the lower left corner of the control refnum control indicates the control reference is strictly typed. Press the <Ctrl> key as you drag a control into the control refnum control to keep the original control and create a strictly typed control refnum. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.

Types of Refnum Controls

LabVIEW includes the following *refnum controls* located on the **Refnum** palette:

	.NET Refnum —Use this refnum to launch the Select .NET Constructor dialog box and select a .NET Framework assembly in the Global Assembly Cache (GAC) or navigate to a private assembly.
	Application Refnum —Used with the VI Server. Use this refnum control when you open a reference to a LabVIEW application and you want to pass the reference as a parameter to another VI. By passing this refnum to the VI Server, you can control the behavior and read properties of the application. You can right-click an application refnum control and select Select VI Server Class from the shortcut menu to change the data type of the control—an application, a VI, a control or indicator, or a strictly typed VI. You can drag any control and drop it on an application refnum to specify the type of control to which it refers.
	Automation Refnum —Used with the ActiveX functions. Use this type of refnum when you open a reference to an ActiveX Server Object and you want to pass it as a parameter to another VI. To specify the type of the refnum, right-click it and select Select ActiveX Class»Browse from the shortcut menu to display the Select Object From Type Library dialog box and browse the list of available ActiveX Server applications.
	Bluetooth Network Connection Refnum —Used with the Bluetooth VIs and functions. Use this type of refnum to open a connection to a Bluetooth device.
	Byte Stream File Refnum —Used with text or binary byte stream files. Use this type of refnum when you open or create a file in one VI but want to perform I/O operations on the file in another VI. You must have a refnum control on the front panel of the VI that performs I/O operations and a refnum indicator on the front panel of the VI that opens or creates the file.
	Control Refnum —Used with the VI Server. Use this refnum control when you open a reference to a front panel control or indicator and you want to pass the reference as a parameter to another VI. By passing this refnum to the VI Server, you can control the behavior and properties of the control. You can right-click a control refnum control and select Select VI Server Class from the shortcut menu to specify the data type of the control—an application, a VI, a control or indicator, or a strictly typed VI. Right-click a control refnum control on the front panel and select Show Icon from the shortcut menu to view the refnum as an icon. Right-click the refnum on the front panel and select Show Control on the shortcut menu to view the refnum as a control. Note that the control is for display only. You can drag any control and drop it on a control refnum to specify the type of control to which it refers.

	Data Log File Refnum —Used with datalog files . Because datalog files have an inherent structure, the Data Log File Refnum passes the refnum and a description of the file type to and from calling VIs. Place a control inside the refnum to define the structure of the file. For a file that contains numbers, create a datalog refnum that contains a number. If each record in the file contains a pair of numbers, place a cluster inside the refnum and place two numeric controls inside the cluster. You can drag any control and drop it on a data log file refnum to specify the data type of the datalog file.
	DataSocket Refnum —Used with DataSocket functions. Use this type of refnum when you want to open a reference to a data connection and you want to pass it as a parameter to another VI. To specify the type of the refnum, right-click it and select Select DataSocket Class from the shortcut menu to browse to the DataSocket class.
	Event Callback Refnum —Use this refnum to unregister or re-register the event callback. You also can use this refnum to pass the event callback registration to a subVI.
	IrDA Network Connection Refnum —Used with the IrDA functions. Use this type of refnum when you want to open a connection to an IrDA device.
	Menu Refnum —Used with the Menu functions. Use this type of refnum when you want to pass a VI menu reference to a subVI.
	Occurrence Refnum —Used with the Occurrence functions. Use this type of refnum when you generate an occurrence in one VI but want to set or wait for the occurrence in another VI.
	TCP Network Connection Refnum —Used with the TCP VIs and functions. Use this type of refnum when you open a network connection in one VI but want to perform I/O operations on the network connection in another VI.
	UDP Network Connection Refnum —Used with the UDP VI and functions. Use this type of refnum when you want to uniquely identify a UDP socket. You can pass the UDP socket reference to a subVI so the subVI can use the socket.
	VI Refnum —Used with the VI Server. Use this refnum control when you open a reference to a VI and you want to pass the reference as a parameter to another VI. By passing this refnum to the VI Server, you can control the behavior and properties of the VI. Use VI refnum controls to pass a VI refnum from one VI to another. The Open VI Reference function requires a strictly typed VI refnum control as a type specifier input when you want a strictly typed reference to a VI. In this case, the value of the refnum control is unimportant. The function uses only the type. You can drag any control and drop it on a VI refnum to specify the type of control to which it refers.

The following refnums are not located on the [Refnum](#) palette:

- **Config Data Refnum** —Used with the Configuration File VIs when you want to open a configuration file in one VI and read from it or write to it in another VI. Right-click a **refnum** input terminal of a Configuration File VI and select **Create»Control** from the shortcut menu to create this refnum control. Creating a config data refnum control does not create a config data reference. You must use the Open Config Data VI to create a config data reference.
- **IMAQ Session Refnum** —Used with the Image Acquisition VIs.
- **Motion Refnum** —Used with the NI SoftMotion VIs and functions.

- Notifier Refnum** —Used with the Notifier Operations functions when you want to suspend the execution of a block diagram until you receive data from another section of the block diagram or from another VI running on the same computer. Right-click a **refnum** input terminal of a Notifier Operations function and select **Create»Control** from the shortcut menu to create this refnum control. Creating a notifier refnum control does not create a notifier reference. You must use the Obtain Notifier function to create a notifier reference. Right-click a notifier refnum on the front panel and select **Show Icon** from the shortcut menu to view the refnum as an icon. Right-click the refnum on the front panel and select **Show Control** on the shortcut menu to view the refnum as a control. Note that the control is for display only. You can drag any control and drop it on a notifier out refnum to specify the data type of the notifier.
- Queue Refnum** —Used with the Queue Operations functions when you want to store data in a queue, which you can later retrieve as individual elements or as an array of all the data elements. Right-click a **refnum** input terminal of a Queue Operations function and select **Create»Control** from the shortcut menu to create this refnum control. Creating a queue refnum control does not create a queue reference. You must use the Obtain Queue function to create a queue reference. Right-click a queue refnum on the front panel and select **Show Icon** from the shortcut menu to view the refnum as an icon. Right-click the refnum on the front panel and select **Show Control** on the shortcut menu to view the refnum as a control. Note that the control is for display only. You can drag any control and drop it on a queue refnum to specify the data type of the queue.
- Rendezvous Refnum** —Used with the Rendezvous VIs when you want to create a rendezvous in one VI and wait at the rendezvous in another VI. Right-click a **refnum** input terminal of a Rendezvous VI and select **Create»Control** from the shortcut menu to create this refnum control. Creating a rendezvous refnum control does not create a valid rendezvous reference. You must use the Create Rendezvous VI to create a rendezvous reference.
- Semaphore Refnum** —Used with the Semaphore VIs when you want to prevent two VIs or two sections of the same VI from executing in parallel. Right-click a **refnum** input terminal of a Semaphore VI and select **Create»Control** from the shortcut menu to create this refnum control. Creating a semaphore refnum control does not create a valid reference to a semaphore. You must use the Obtain Semaphore Reference VI to create a reference to a semaphore.
- Shared Variable Refnum** —Used with the Variable properties to configure shared variables programmatically. Right-click the **Variable Reference** output of a Variable Reference property and select **Create»Control** from the shortcut menu to create this refnum control.
- TDMS File Refnum** —Used with the TDMS Streaming VIs and functions to read and write waveforms and waveform properties to binary measurement files .tdms. Right-click the **tdms file** input of a TDMS Read function and select **Create»Control** from the shortcut menu to create this refnum control. Creating a tdms file control does not create a tdms file reference. You must use the TDMS Open function to create a tdms file reference.
- Data Value Refnum** —Used with the In Place Element structure when you want to operate on a data value without requiring the LabVIEW compiler to copy the data values and maintain those values in memory. Right-click the **Data Value Reference** output of the In Place Element structure to create this refnum control. Creating a data value refnum control does not create a [data value reference](#). You must use the New Data Value Reference function to create a data value reference. Right-click a data value refnum on the front panel and select **Show Icon** from the shortcut menu

to view the refnum as an icon. Right-click the refnum on the front panel and select **Show Control** on the shortcut menu to view the refnum as a control. Note that the control is for display only.

- **External Data Value Refnum** —Used with the In Place Element structure when you want to operate on an external data value without incurring a data copy to bring the data into LabVIEW. Creating an external data value refnum control does not create an *external data value reference*. You must get the external data value reference from a device driver that supports external data value references. To change a data value reference to an external data value reference, right-click the data value reference and select **External** from the shortcut menu. You also can use the Is External property to change a data value reference to an external data value reference programmatically. Right-click an external data value refnum on the front panel and select **Show Icon** from the shortcut menu to view the refnum as an icon. Right-click the refnum on the front panel and select **Show Control** on the shortcut menu to view the refnum as a control. Note that the control is for display only.
- **Variable Refnum** —Used with the Shared Variable functions and the Variable properties and methods to find, read, and write shared variables programmatically. Right-click the **shared variable refnum in** input terminal of a Shared Variable function to create this refnum control.

Ring and Enum Controls

Ring & Enum Controls

Use the *ring and enumerated type controls and indicators* located on the **Ring & Enum** and **Classic Ring & Enum** palettes to create a list of strings you can cycle through.

Adding Picture Items to Ring Controls

A new picture ring control or text and picture ring control has one item with an empty picture display. You must copy a picture to the clipboard before you can import the picture into a ring control. LabVIEW can import all graphic files your graphics software supports.

Complete the following steps to import pictures into ring controls.

1. Copy a picture to the clipboard.
2. Right-click the picture ring control or text and picture ring control and select **Import Picture from Clipboard** from the shortcut menu.
3. Copy another picture to the clipboard.
4. Right-click the picture ring control or text and picture ring control and select **Import Picture Before** or **Import Picture After** from the shortcut menu.

Use the Operating tool to click the picture box to view all imported pictures.

Assigning Specific Numeric Values to Ring Control Items

When you [configure the list of items for a ring control](#), you can assign a specific numeric value to each item. If you do not assign specific numeric values to the items, LabVIEW assigns sequential values that correspond to the order of the items in the list, starting with a value of 0 for the first item.

Complete the following steps to assign a specific numeric value to an item in a ring control.

1. Right-click the ring control and select **Edit Items** from the shortcut menu to display the Edit Items page of the **Ring Properties** dialog box.
2. Remove the checkmark from the **Sequential values** checkbox.
3. Double-click in the cell in the **Values** column that contains the numeric value you want to change.
4. Enter the numeric value you want to use on the block diagram when the user selects the corresponding label in the control. Each item in the ring control must have a unique numeric value.



NOTE

The value you enter must be a valid value for the [numeric representation](#) of the ring control. For example, if the representation of the ring control is 16-bit unsigned integer, you cannot enter a negative value.

1. Repeat steps 3–4 for each numeric value you want to change.
2. Click the **OK** button.

You also can use the Strings And Values [] property to assign specific numeric values programmatically.

Configuring Ring Controls to Allow Undefined Numeric Values

You can configure the ring control so users can enter numeric values not already associated with any entries in the list of items defined for the control. Right-click the ring control and select **Allow Undefined Values** from the shortcut menu to allow the user to enter any numeric value in the digital display of the control.

You also can use the Allow Undefined Values property to allow undefined numeric values programmatically.



RELATED INFORMATION

[Selecting Items in Ring and Enumerated Type Controls at Run Time](#) on page 1074

[Showing and Hiding Labels and Digital Displays](#) on page 1187

Displaying Numeric Values of Ring Controls and Constants

Right-click the ring control or constant and select **Visible Items»Digital Display** from the shortcut menu to display the numeric value associated with the current item.

Editing Items in Ring Controls and Enumerated Type Controls

Complete the following steps to edit items in a ring control or enumerated type control:

1. Right-click the control and select **Edit Items** from the shortcut menu to display the Edit Items page of the Properties dialog box.
2. Complete one or more of the following actions:
 - Press the <Insert> key to add an item. (macOS) Click the **Insert** button.
 - Select an item and press the <Delete> key to delete an item.
 - Select an item and click the **Disable** button to disable an item. Users cannot select disabled items while the VI is running. If the item is already disabled, you can click the **Enable** button to enable it. You also can use the Disabled Items [] property to disable or enable items programmatically.
 - Double-click an item to edit the name of the item.
 - Drag and drop items to change their order.
 - Enter a hyphen - in an empty cell to create a separator in the list of items for the control. A separator is a thin bar that separates items in the menu of a ring or enumerated type control.



NOTE

You can drag and drop, enable or disable, and delete multiple items at once by pressing and holding the <Ctrl> key, selecting the items, and completing the desired action. (macOS) Press the <Command> key. (Linux) Press the <Alt> key.

3. Click **OK**.

You also can use the Strings [] property to add or remove items from ring controls programmatically.

Selecting Items in Ring and Enumerated Type Controls at Run Time

Use the following ways to select items in a ring or enumerated type control while a VI is in run mode:

- Click the increment or decrement buttons.
- Press the <Tab> key to *navigate to the control* and press the up and down arrow keys.

Use the following ways to select items in a ring or enumerated type control while a VI is in run mode or to select items in a ring or enum constant while a VI is in edit mode:

- Click the control to display the shortcut menu and use the up and down arrow keys to navigate to the item you want to select. After you highlight the item in the shortcut menu, press the <Enter> key to select the item.
- Click the control to display the shortcut menu and type the beginning letters of the string you want to select. LabVIEW highlights the first item in the control that begins with the letters you type. Press the <Enter> key to select the item.
- If the **digital display** is visible, place the cursor in the digital display, enter a numeric value, and press the <Enter> key. If you enter a value that does not exist in the control, LabVIEW changes the value you entered to the closest valid value.
- If a ring control allows **undefined numeric values**, click the control, select **<Other>** from the shortcut menu, enter a numeric value in the digital display that appears, and press the <Enter> key. The undefined value appears in the ring control in angle brackets. If the digital display is visible, you also can place the cursor in the digital display, enter a numeric value, and press the <Enter> key.

Using Enumerated Type Controls

Use **enumerated type controls** to give users a list of items from which to select. An enumerated type control, or enum, is similar to a text or menu **ring control** in that both contain lists of strings through which you can cycle. However, the data type of an enumerated type control includes information about the numeric values and the string labels in the control. The data type of a ring control is numeric.



NOTE

You cannot allow the user to enter undefined values in enumerated type controls, and you cannot assign specific numeric values to items in enumerated type controls. If you need this functionality, use a **ring control**.

You can use an enumerated type control to select the cases of a Case structure.

The numeric representation of the enumerated type control is 8-, 16-, or 32-bit unsigned integer. Right-click the enumerated type control and select **Representation** from the shortcut menu to **change the representation** of the control.

Using Ring Controls

Use a ring control to select an item from a list without having to know what value that item represents. LabVIEW passes the value associated with the selected item to the block diagram. Select a case from a Case structure (conditional code) that carries out the selected item.

Complete the following steps to create a ring control.

1. Add a ring control to the front panel.

2. Add text or picture items to the ring control.
3. (Optional) Remove items from the ring control.



RELATED INFORMATION

[Editing Items in Ring Controls and Enumerated Type Controls](#) on page 1074

[Adding Picture Items to Ring Controls](#) on page 1072

[Editing Items in Ring Controls and Enumerated Type Controls](#) on page 1074

Changing the Value of a Path Control

When the **Path** symbol  in a path control or indicator changes to the **Not a Path** symbol , <Not a Path> appears in the text field to indicate the path is invalid. Use the <Not a Path> value as the default value for a path control to detect when the user fails to provide a path and display a file dialog box with options for selecting a path. Use the File Dialog Express VI to display a file dialog box.

Complete the following steps to change the value of a path control.

1. Click the **Path** symbol.
2. Select **Valid Path** or **Not a Path** from the shortcut menu.

Configuring Browse Options of Path Controls

You can display a browse button for path controls by right-clicking the path control and selecting **Visible Items»Browse Button** from the shortcut menu to place a checkmark next to it. You also can use the Browse Button:Visible property to display the browse button programmatically.

Complete the following steps to customize the settings of the file dialog that appears when you click the **Browse** button of the file path control, such as which types of paths you can select, a start path, a pattern to match, and so on.

1. Right-click a path control and select **Browse Options** from the shortcut menu to display the Browse Options page of the **Path Properties** dialog box.

2. Select the appropriate option in the **Selection Mode** section. The options in this section are similar to the **Selection Mode** options in the configuration dialog box of the File Dialog Express VI.



NOTE

To select an LLB or packed library using a path control, you can remove the checkmark from the **Allow selection of files in LLBs and packed project libraries** checkbox and select the LLB or packed library from the file dialog box. You also can place a checkmark in the **Allow selection of files in LLBs and packed project libraries** checkbox, select the **Files or Folders** option button, choose an LLB or packed library from the file dialog box, and select the second folder icon from the file list in the second file dialog box that appears.

1. Click the **OK** button.



Tip To list multiple file types in a file dialog box using the **File Dialog** Express VI or Open/Create/Replace File function, wire a string constant or control to the **pattern** terminal and use semicolons to separate the patterns, for example `fx*.bin;*.c`.

Configuring Combo Box Controls to Allow Undefined String Values

By default, the combo box control allows users to *enter string values* not already in the list of strings defined for the control. Right-click the combo box control and select **Allow Undefined Strings** from the shortcut menu to remove the checkmark and prevent the user from entering undefined string values in the control.

You also can use the Allow Undefined Strings property to allow undefined string values programmatically.

Editing Items in Combo Box Controls

Complete the following steps to edit items in a combo box control:

1. Right-click the control and select **Edit Items** from the shortcut menu to display the Edit Items page of the Properties dialog box.
2. Complete one or more of the following actions:
 - Press the <Insert> key to add an item. (macOS) Click the **Insert** button.
 - Select an item and press the <Delete> key to delete an item.
 - Double-click an item to edit the name of the item.
 - Drag and drop items to change their order.

- Enter a hyphen - in an empty cell to create a separator in the list of items for the control. A separator is a thin bar that separates items in the menu of a combo box control.

**NOTE**

You can drag and drop or delete multiple items at once by pressing and holding the <Ctrl> key, selecting the items, and completing the desired action. (macOS) Press the <Command> key. (Linux) Press the <Alt> key.

3. Click **OK**.

You also can use the Strings [] property or the Strings And Values [] property to edit strings in combo boxes programmatically.

XControls

Creating XControls

Use **XControls** to design and create complex controls and indicators in LabVIEW.

**NOTE**

You can create and edit XControls only in the LabVIEW Professional Development System. If a VI contains an XControl, you can run the VI in all LabVIEW packages.

Complete the following steps to create an XControl:

1. Open the project in which you want to use an XControl. You can [create a new project](#) to display the Project Explorer window.
2. In the [Project Explorer](#) window, right-click **My Computer** and select **New»XControl** from the shortcut menu. LabVIEW creates a new XControl library in the project. Within the XControl library, LabVIEW automatically creates required [abilities](#).

**TIP**

You also can use the XControl template in the New dialog box to create an XControl.

1. Double-click the Data ability in the XControl library to open and edit the Data type definition. Replace the numeric control with a control that represents the data type of the new XControl. Select **File»Apply Changes** to propagate the change in the Data ability.
2. Double-click the State ability in the XControl library to open and edit the State type definition. Include any information that affects the appearance or state of the XControl as part of the cluster control. Select **File»Apply Changes** to propagate the change in the State ability.

3. Double-click the Facade ability in the XControl library to open and [edit the Facade VI](#).
4. Right-click the XControl library and select **Properties** from the shortcut menu to display the XControl Properties dialog box. Use this dialog box to configure the XControl library, document the XControl, and configure settings for the abilities, properties, methods, and VIs of an XControl.
5. Right-click the XControl library and select **Save** from the shortcut menu to save the XControl.

You also can add [optional abilities](#), [properties](#), [methods](#), and VIs to the XControl library. Right-click the XControl library and select **New** from the shortcut menu to add an item to the XControl library.

Refer to Simple Dual Mode Thermometer XControl.lvproj in the labview\examples\Controls and Indicators\XControls\Dual Mode Thermometer directory for an example of creating an XControl.

Creating XControl Abilities

XControl abilities are components of the XControl that the XControl requires to function properly. When you create an XControl, LabVIEW automatically creates the required abilities. Complete the following steps to add an optional ability to an XControl library.

1. In the Project Explorer window, right-click the XControl library and select **New»Ability** from the shortcut menu to display the Select Ability dialog box. This dialog box lists all required and optional abilities that are applicable to the XControl. Use the **Select Ability** dialog box to add optional abilities to the XControl library.
2. In the **Abilities** section, select an ability and click the **Add** button to add the ability to the XControl library.
3. To edit the ability, double-click the ability in the XControl library to open the VI or type definition that represents the ability.



RELATED INFORMATION

[Using XControl Abilities](#) on page 174

[Creating XControls](#) on page 1078

[Using XControl Abilities](#) on page 174

[Using XControls](#) on page 174

[Managing a Project in LabVIEW](#) on page 365

Creating XControl Properties

[XControl](#) properties allow the user of an XControl to configure the XControl programmatically. The user of an XControl uses a Property Node to get and set XControl

properties. LabVIEW calls the *Facade VI* after the user gets or sets a custom XControl property. If using the property changes the display state, the Facade VI updates the display state so that the XControl can update its appearance. A Display State Change event is generated on the Facade VI in this case.

In the XControl library, each XControl property is represented by one or more VIs, depending on if the property is read-only, write-only, or read/write. Each VI represents the read or write functionality of the property.

Complete the following steps to add a property to an *XControl library*.

1. Right-click the XControl library and select **New»Property** from the shortcut menu to display the Create Property dialog box. Use this dialog box to add a property to the XControl library. LabVIEW creates a folder with the same name as the property in the XControl library. If you create a read/write property, the property folder contains a VI for reading from the property and a VI for writing to the property. If you create a read-only or write-only property, the property folder contains only one VI for reading from or writing to the property, respectively.



NOTE

The property folder cannot contain more than two VIs.

1. To edit the property, double-click each VI in the property folder to open the VI that represents read or write functionality of the property.

Editing the Read VI

The read VI represents the read functionality of an XControl property. The read VI includes display state and container state controls and display state, value, and error indicators. Do not delete or change any of these controls or indicators because LabVIEW uses them to pass in display state and container state to the property and retrieve any error information. By default, the read VI also includes an LED indicator for the value. Replace the LED indicator with an indicator of the appropriate data type to configure the type of data that the property reads.



Note If the XControl property is read/write, the data type of the **Value** indicator of the read VI must match the data type of the **Value** control of the write VI.

To delete the VI, right-click the VI and select **Remove from Project**. Right-click the property folder and select **New»Read VI** to add a read VI to a write-only property.

Editing the Write VI

The write VI is similar to the read VI. It includes the same display state, container state, and error controls and indicators as the read VI. Do not delete or change any of these controls or indicators because LabVIEW uses them to pass in display state and container state to the property and retrieve any error information and the updated display state. By default, the

write VI includes an LED control for the value. Replace the LED control with a control of the appropriate data type to configure the type of data the property writes.



Note If the XControl property is read/write, the data type of the **Value** indicator of the read VI must match the data type of the **Value** control of the write VI.

To delete the VI, right-click the VI and select **Remove from Project**. Right-click the property folder and select **New»Write VI** to add a write VI to a read-only property.

Creating XControl Methods

XControl methods allow the user of an XControl to configure the XControl programmatically. The user of an XControl uses an Invoke Node to invoke XControl methods. LabVIEW calls the **Facade VI** after the user invokes a custom XControl method. If invoking the method changes the display state, the Facade VI updates the display state so that the XControl can update its appearance. A Display State Change event is generated on the Facade VI in this case.

In the XControl library, each XControl method is represented by a VI.

Complete the following steps to add a method to the **XControl library**.

1. Right-click the XControl library and select **New»Method** from the shortcut menu to add a new method to the XControl library. LabVIEW creates a new method in the XControl library.
2. Double-click the method in the XControl library to add the controls and indicators that represent the parameters of the method.

Editing the Method VI

The method VI represents an XControl method. The method VI includes a display state control and display state, container state, and error indicators. Do not delete or change any of these controls or indicators because LabVIEW uses them to pass in display state to the method and retrieve any error information.

Add controls or indicators to represent the parameters of the method. Parameters can be input, output, or input/output parameters. Each control or indicator that you add to the VI can be part of only one parameter. LabVIEW uses the name of the control or indicator as the name of the parameter.



NOTE

If the parameter is an input/output parameter, you must use a control/indicator pair to represent the parameter. The control/indicator pair must be of the same data type.

Wire the controls or indicators to the connector pane. Right-click the method and select **Configure Method** from the shortcut menu to display the Configure Method dialog box. Use this dialog box to configure the parameters you added to the method.

To delete the VI, right-click the VI and select **Remove from Project**.

Altering the Appearance of XControls

Set the *window appearance properties* of the *Facade VI* to adjust the appearance of the *XControl*.

XControl Scroll Bars

To show or hide the scroll bars for an XControl, show or hide the scroll bars for the front panel window of the Facade VI. Use the **Show vertical scroll bar** and **Show horizontal scroll bar** checkboxes of the Customize Window Appearance dialog box to show or hide scroll bars. If the XControl has multiple panes, you must show or hide the scroll bars for each pane of the Facade VI. Right-click the splitter bar and choose **Visible Items»Horizontal Scrollbar** or **Visible Items»Vertical Scrollbar** from the shortcut menu of each pane to display the scroll bars.

XControl Borders

To show or hide the border of an XControl, show or hide the title bar of the Facade VI. Use the **Window has title bar** checkbox of the Customize Window Appearance dialog box to show or hide the title bar.

XControl Backgrounds (Windows and macOS)

To make the background of the XControl transparent, make the front panel of the Facade VI transparent. Place a checkmark in the **Window Runs Transparently** checkbox of the Customize Window Appearance dialog box to make the XControl transparent.

Resizing XControls

You can configure an *XControl* to respond to resizing by enabling front panel scaling in the *Facade VI* or handling the Panel Resize event in the Event structure in the Facade VI.

If you enable front panel scaling in the Facade VI, the controls that make up the appearance of the XControl automatically resize when the XControl resizes. Complete the following steps to enable scaling of the front panel in the Facade VI.



NOTE

If the Facade VI has multiple panes, you must complete these steps for each pane.

1. Double-click the Facade ability in the XControl library to open and *edit the Facade VI*.
2. Set all front panel objects to *scale automatically* when the front panel resizes.

Alternatively, you can handle the Panel Resize event in the Event structure in the Facade VI. When the XControl is resized, LabVIEW calls the Facade VI with a Panel Resize event. In response to the event, you can move and resize each control individually that makes up the appearance of the XControl. Using the Panel Resize event is useful because you have more flexibility when you resize; however, it requires more coding.

Complete the following steps to configure the Event structure in the Facade VI to handle the Panel Resize event.

1. Double-click the Facade ability in the XControl library to open and edit the Facade VI.
2. Right-click the Event structure on the block diagram and select **Add Event Case** from the shortcut menu to display the Edit Events dialog box.
3. In the dialog box, select **<This VI>** from the **Event Sources** list.
4. Select **Panel Resize** from the **Events** list.
5. Click the **OK** button to save the configuration and close the dialog box.



Note If the XControl must have a fixed size, disable resizing on the front panel of the Facade VI. Remove the checkmark from the **Allow user to resize window** checkbox in the Customize Window Appearance dialog box to disable resizing.

If the XControl must have a minimum size, specify the minimum size for the Facade VI. Enter width and height values for the **Minimum Panel Size** option of the Window Size page of the VI Properties dialog box.

Debugging XControls

To debug the Facade VI, open the front panel of a VI, right-click the XControl, and select **Advanced»Show Block Diagram** from the shortcut menu. Use breakpoints, single-stepping, and probes to debug the Facade VI just as you would debug any VI.

To debug abilities, properties or methods, set a breakpoint in the ability, property, or method VI. When LabVIEW calls the VI, LabVIEW pauses execution at the breakpoint and

displays the block diagram of the VI. Use single-stepping and probes to debug the VI just as you would debug any VI.



RELATED INFORMATION

[Using the XControl Facade Ability](#) on page 175

[Managing Breakpoints](#) on page 1427

[Single-Stepping through a VI](#) on page 953

[Using the Probe Tool](#) on page 1425

[Using XControl Abilities](#) on page 174

[Creating XControl Properties](#) on page 1079

[Creating XControl Methods](#) on page 1081

Editing XControls That Are in Use

When you add an **XControl** to the front panel of a VI, LabVIEW loads the XControl library. LabVIEW locks the XControl library while the XControl is in use by a VI. While the XControl library is locked, you cannot edit it. Complete the following steps to unlock and edit an XControl that is in use by a VI.

1. Open the project file `.lvproj` for the XControl library in the **Project Explorer** window.
2. Right-click the XControl library and select **Unlock Library for Editing** from the shortcut menu.



NOTE

You must stop any VIs that use the XControl that are running before LabVIEW can unlock the XControl library for editing.

1. When you use the **Unlock Library for Editing** item, LabVIEW unlocks the library; however, any VIs that use the XControl are now broken. Right-click the XControl library and select **Apply Changes to Instances** from the shortcut menu to apply changes to all of the VIs that use the XControl. The VIs should no longer be broken.

Obtaining a Reference in a Facade VI

Complete the following steps to obtain a reference to the VI that owns a specific XControl.

1. Double-click the Facade ability in the XControl library to open and edit the **Facade VI**.
2. Right-click the output of the **Container State** cluster on the block diagram and select **Cluster, Class, & Variant Palette** from the shortcut menu.

3. Select the Unbundle by Name function from the palette and wire the function to the **Container State** cluster.
4. Right-click the Unbundle by Name function and select **Select Item»Refnum** from the shortcut menu.
5. Right-click the **Refnum** and select **Create»Property for Control Class»Owner** from the shortcut menu. Place the Property Node next to the Unbundle by Name function.
6. Wire the output of the Unbundle by Name function to the **reference** input of the Property Node.
7. Right-click the **Refnum** and select **Create»Property for Control Class»Owning VI** from the shortcut menu to create a second Property Node. Place the Property Node to the right of the **Owner** Property Node you created in step 5.
8. Wire the **reference out** output of the **Owner** Property Node to the **reference** input of the **Owning VI** Property Node.
9. Run the VI.

Configuring Run-Time Drag-and-Drop Behavior for Text-Based Controls

To enable users to drag and drop data between text-based controls in a VI, such as string, path, listbox, multicolumn listbox, and tree controls, you must configure drag-and-drop behavior for those controls. You can configure the drag-and-drop behavior of text-based controls in the following ways:

- *Allow dragging from the control during run time.*
- *Allow dropping in the control during run time.*
- *Select between moving and copying data from the control.*
- *Allow dropping in controls other than the control from which the drag originated.*
- *Allow dragging of parent items in tree controls.*
- *Allow items dropped in tree controls to drop as parent items.*

Enable multiple drag-and-drop settings simultaneously to configure specific behavior for text-based controls. For example, to enable dragging from a listbox to a tree control, you must allow dragging on the listbox, allow data from the listbox to drop in controls other than the listbox, and allow dropping in the tree control.

Allow Dragging from the Control during Run Time

Control	Shortcut Menu Item	Property
String	Advanced»Allow Dragging	String Properties: Allow Dragging
Listbox	Drag and Drop»Allow Dragging	Listbox Properties: Drag/Drop:Allow Dragging

Multicolumn Listbox	Drag and Drop»Item Dragging»Allow Dragging	Multicolumn Listbox Properties: Drag/Drop:Allow Dragging
Tree	Drag and Drop»Allow Dragging	Tree Properties: Drag/Drop:Allow Dragging

Allow Dropping in the Control during Run Time

Control	Shortcut Menu Item	Property
String	Advanced»Allow Dropping	String Properties: Allow Dropping
Path	Advanced»Allow Dropping	Path Properties: Allow Dropping
Listbox	Drag and Drop»Allow Dropping	Listbox Properties: Drag/Drop:Allow Dropping
Multicolumn Listbox	Drag and Drop»Allow Dropping	Multicolumn Listbox Properties: Drag/Drop:Allow Dropping
Tree	Drag and Drop»Item Dragging»Allow Dropping	Tree Properties: Drag/Drop:Allow Dropping

Select between Moving and Copying Data from the Control

Control	Shortcut Menu Item	Property
Listbox	• Drag and Drop»Drag Mode»Move Only • Drag and Drop»Drag Mode»Copy Only • Drag and Drop»Drag Mode»Copy or Move	Listbox Properties: Drag/Drop:Drag Mode
Multicolumn Listbox	• Drag and Drop»Drag Mode»Move Only • Drag and Drop»Drag Mode»Copy Only • Drag and Drop»Drag Mode»Copy or Move	Multicolumn Listbox Properties: Drag/Drop:Drag Mode
Tree	• Drag and Drop»Drag Mode»Move Only • Drag and Drop»Drag Mode»Copy Only • Drag and Drop»Drag Mode»Copy or Move	Tree Properties: Drag/Drop:Drag Mode

Allow Dropping in Controls Other Than the Control from Which the Drag Originated

Control	Shortcut Menu Item	Property

Listbox	Drag and Drop»Item Dragging»Allow Drag Drop Outside Control	Listbox Properties: Drag/Drop:Allow Drag and Drop Outside Control
Multicolumn Listbox	Drag and Drop»Item Dragging»Allow Drag Drop Outside Control	Multicolumn Listbox Properties: Drag/Drop:Allow Drag and Drop Outside Control
Tree	Drag and Drop»Item Dragging»Allow Drag Drop Outside Control	Tree Properties: Drag/Drop:Allow Drag and Drop Outside Control

Allow Dragging of Parent Items in Tree Controls

Control	Shortcut Menu Item	Property
Tree	Drag and Drop»Item Dragging»Allow Dragging of Parent Items	Tree Properties: Drag/Drop:Allow Dragging of Parent Items



NOTE

If you drag a parent item with child items from a tree control to another text-based control, only the parent item drops. Parent and child items drop only if you drop the items in another tree control.

Allow Items Dropped in Tree Controls to Drop as Parent Items

Control	Shortcut Menu Item	Property
Tree	Drag and Drop»Item Dropping»Allow Dropping Between Items	Tree Properties: Drag/Drop:Allow Dropping Between Items

Implementing Custom Drag-and-Drop Behavior

Although LabVIEW provides built-in drag-and-drop behavior for text-based controls, you can programmatically implement custom drag-and-drop behavior for any controls, including non-text-based controls. Programmatically implement drag-and-drop behavior with drag-and-drop events to accomplish the following tasks:

- Allow users to drag and drop data between non-text-based controls.
- Provide custom drag-and-drop behavior for text-based controls.



NOTE

These topics assume familiarity with Event structures, properties, and methods.

Complete the following steps to implement custom drag-and-drop behavior:

1. Identify which controls to use as the drag source and drop target.
2. Prepare the VI to detect when the user causes drag-and-drop events to occur.
3. Detect when the user aborts a drag-and-drop operation.
4. Start the drag-and-drop operation by providing the drag data.
5. Verify that the drop target can accept the drag data.
6. Update the drop target with the drag data.
7. Determine when to stop the VI.

**NOTE**

Error handling practices are omitted from the block diagrams in these topics to improve the clarity of the block diagram and to focus on the steps necessary for implementing custom drag-and-drop behavior. Provide error handling mechanisms to tell you why and where errors occur.

Additional Examples

Refer to the Drag and Drop - Passing Custom Drag Data VI in the `labview\examples\Structures\Event Structure` directory for an example of using drag-and-drop events to implement drag-and-drop behavior.

Refer to the Drag and Drop - Initiating a Custom Drag VI in the `labview\examples\Structures\Event Structure` directory for an example of implementing drag-and-drop behavior on the plots of graphs.

**RELATED INFORMATION**

[Configuring Run-Time Drag-and-Drop Behavior for Text-Based Controls](#) on page 1085

[Choosing How the Event Structure Monitors For Events](#) on page 454

[Creating Properties and Methods](#) on page 1476

[Creating Properties and Methods](#) on page 1476

[Step 1: Identify Which Controls to Use as the Drag Source and Drop Target](#) on page 1089

[Step 2: Prepare the VI to Detect When the User Causes Drag-and-Drop Events to Occur](#) on page 1089

[Step 3: Detect When the User Aborts a Drag-and-Drop Operation](#) on page 1091

[Step 4: Start the Drag-and-Drop Operation by Providing the Drag Data](#) on page 1091

[Step 5: Verify that the Drop Target Can Accept the Drag Data](#) on page 1093

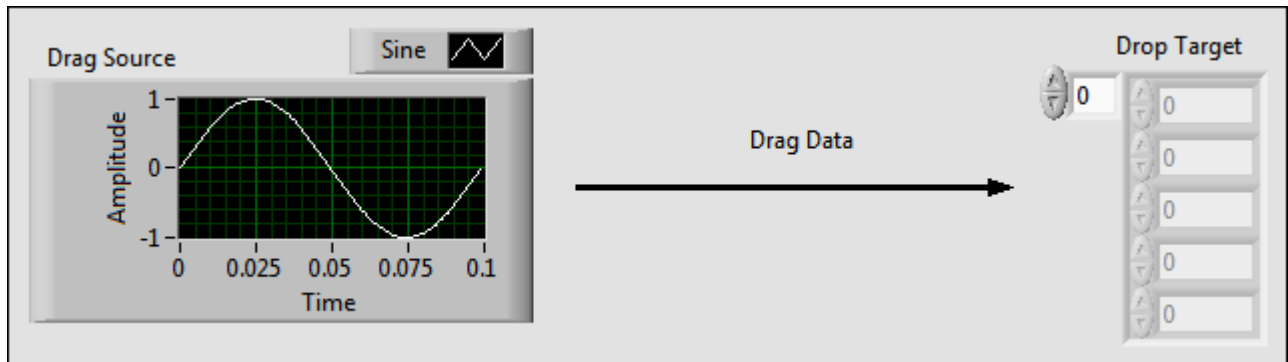
[Step 6: Update the Drop Target with the Drag Data](#) on page 1094

[Handling a Latched Stop Boolean Control in an Event Structure](#) on page 1325

[Handling Errors](#) on page 338

Step 1: Identify Which Controls to Use as the Drag Source and Drop Target

Drag-and-drop events are associated with the drag source and drop target. The drag source is the control that you want to provide the drag data, and the drop target is the control that you want to accept the data. This topic explains how to drag and drop data from a waveform graph to an array of numeric controls. Therefore, the drag source is the waveform graph, and the drop target is the array, shown as follows:



When you *prepare the VI to detect when the user causes drag-and-drop events to occur*, identify controls as the drag source and drop target by setting the controls to be the event source for the relevant events.

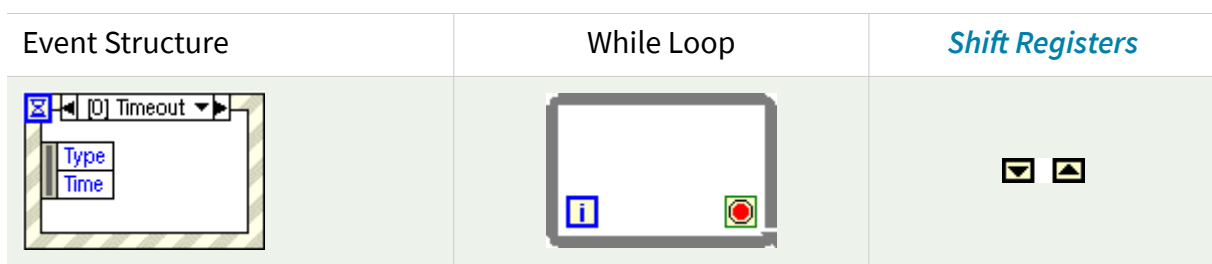
After you identify which controls to use as the drag source and drop target, *prepare the VI to detect when the user causes drag-and-drop events to occur*.

Step 2: Prepare the VI to Detect When the User Causes Drag-and-Drop Events to Occur

When the user tries to drag and drop data between controls, he or she causes particular events to occur. To allow the user to successfully copy or move data from the drag source to the drop target, you must configure the relevant events to interpret user actions as drag-and-drop behavior.

What to Use

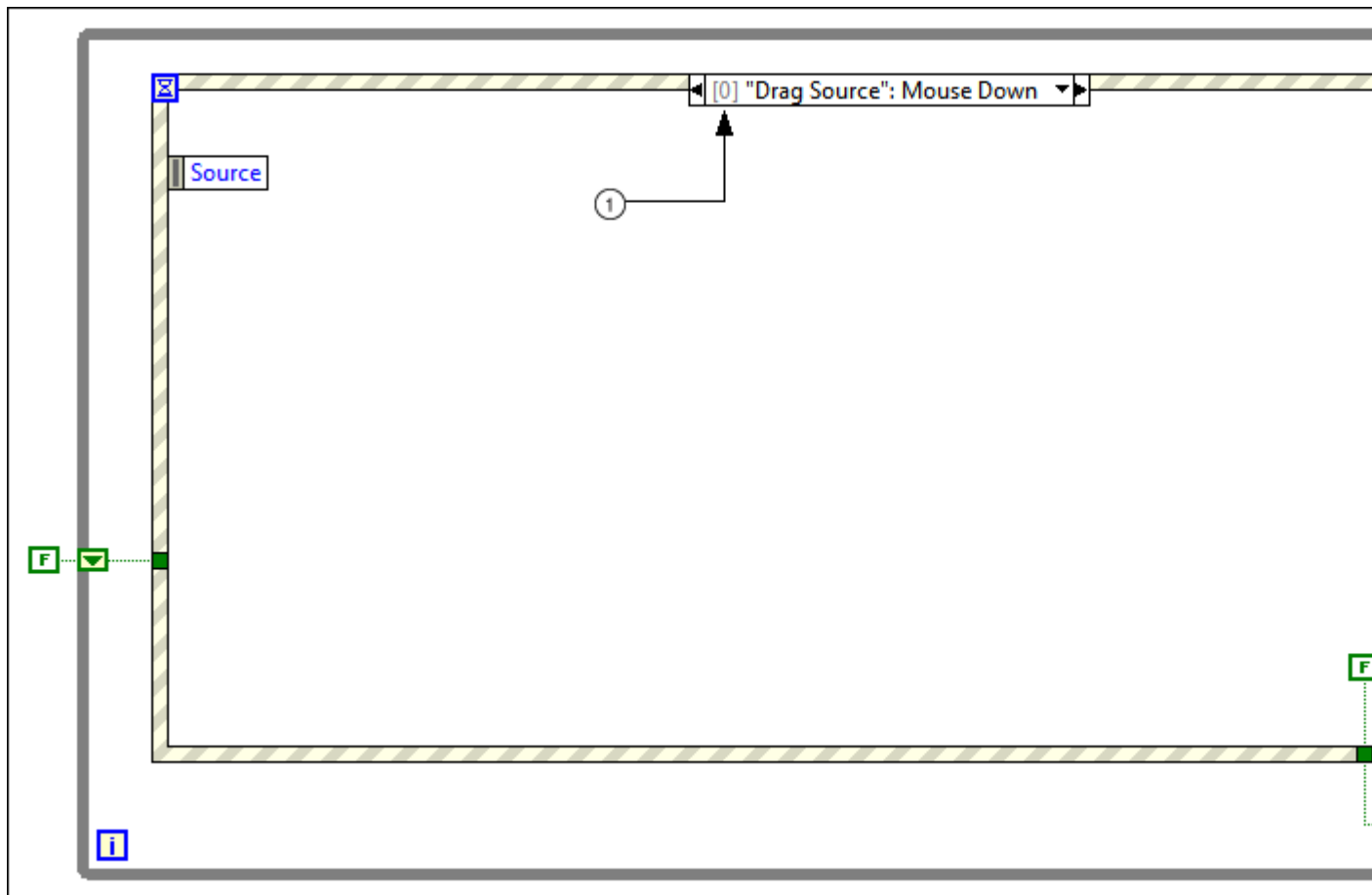
Use the following block diagram objects to prepare the VI to detect when the user causes drag-and-drop events to occur:



Create

What to Do

Create the following block diagram to prepare the VI to detect when the user causes drag-and-drop events to occur.



The following list describes important details about the previous block diagram:



NOTE

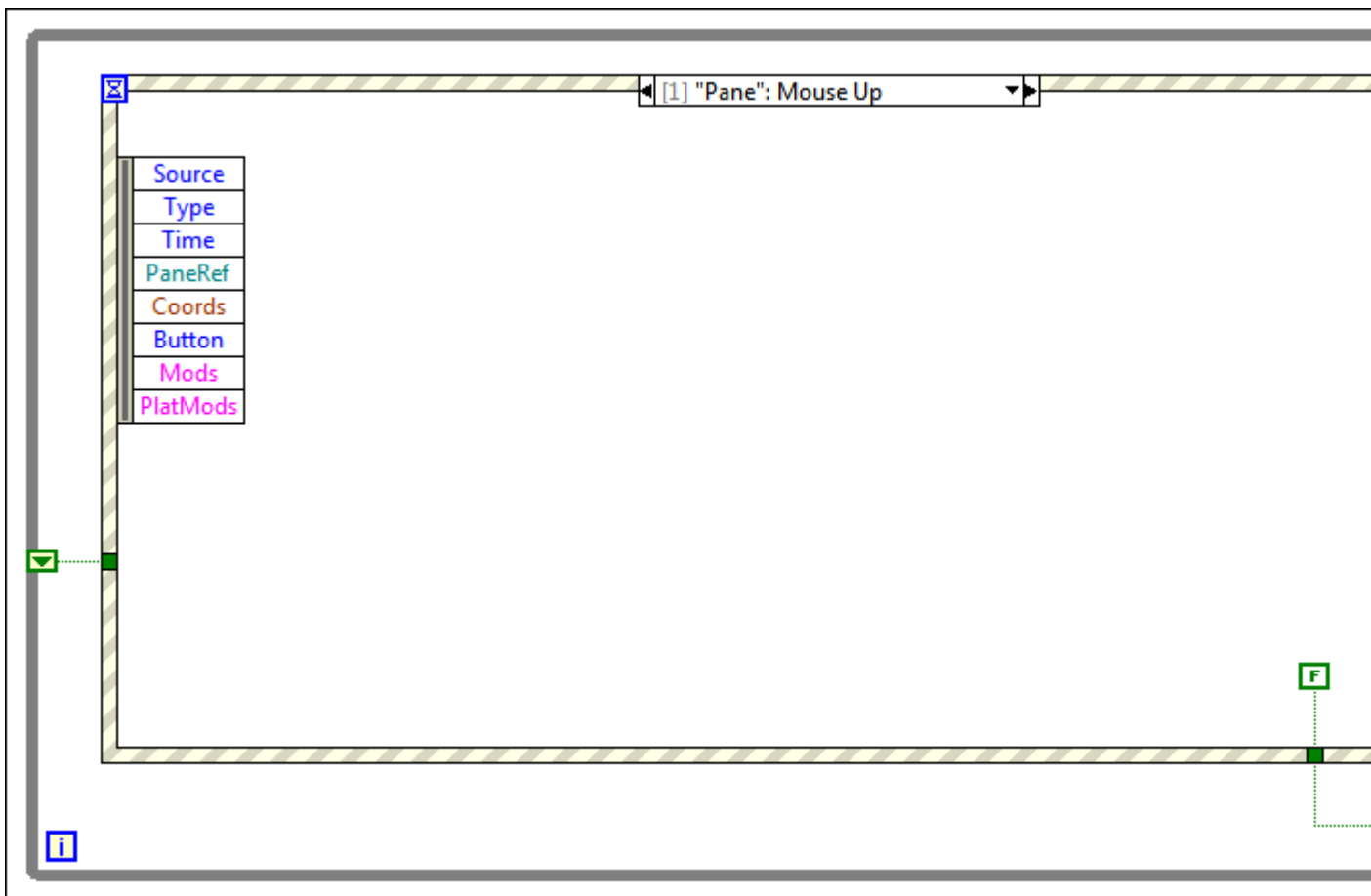
The controls in the events above are labeled Drag Source and Drop Target. To implement drag-and-drop behavior, set the desired drag source to be the event source for the Mouse Down and Mouse Leave events, and set the desired drop target to be the event source for the Drag Enter and Drop events.

After you prepare the VI to detect when the user causes drag-and-drop events to occur, *detect when the user aborts a drag-and-drop operation.*

Step 3: Detect When the User Aborts a Drag-and-Drop Operation

The user aborts a drag-and-drop operation if he or she releases the mouse button over any location on the front panel other than the drop target. To end the drag-and-drop operation when the user releases the mouse button, you must detect the Mouse Up event and prevent the drag from starting if the event occurs. Providing a FALSE value to the shift register prevents the VI from starting the drag if the Mouse Up event occurs.

Create the following block diagram to detect when the user aborts a drag-and-drop operation.



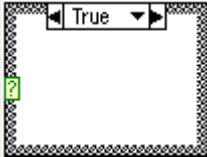
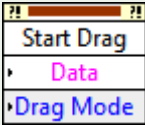
After you detect when the user aborts a drag-and-drop operation, *start the drag-and-drop operation by providing the drag data*.

Step 4: Start the Drag-and-Drop Operation by Providing the Drag Data

To transfer data from the drag source to the drop target, you first must store the drag data when the user presses the mouse button and moves the cursor outside the bounds of the drag source. LabVIEW can retrieve the stored drag data later to update the drop target when the user causes the drop operation to occur.

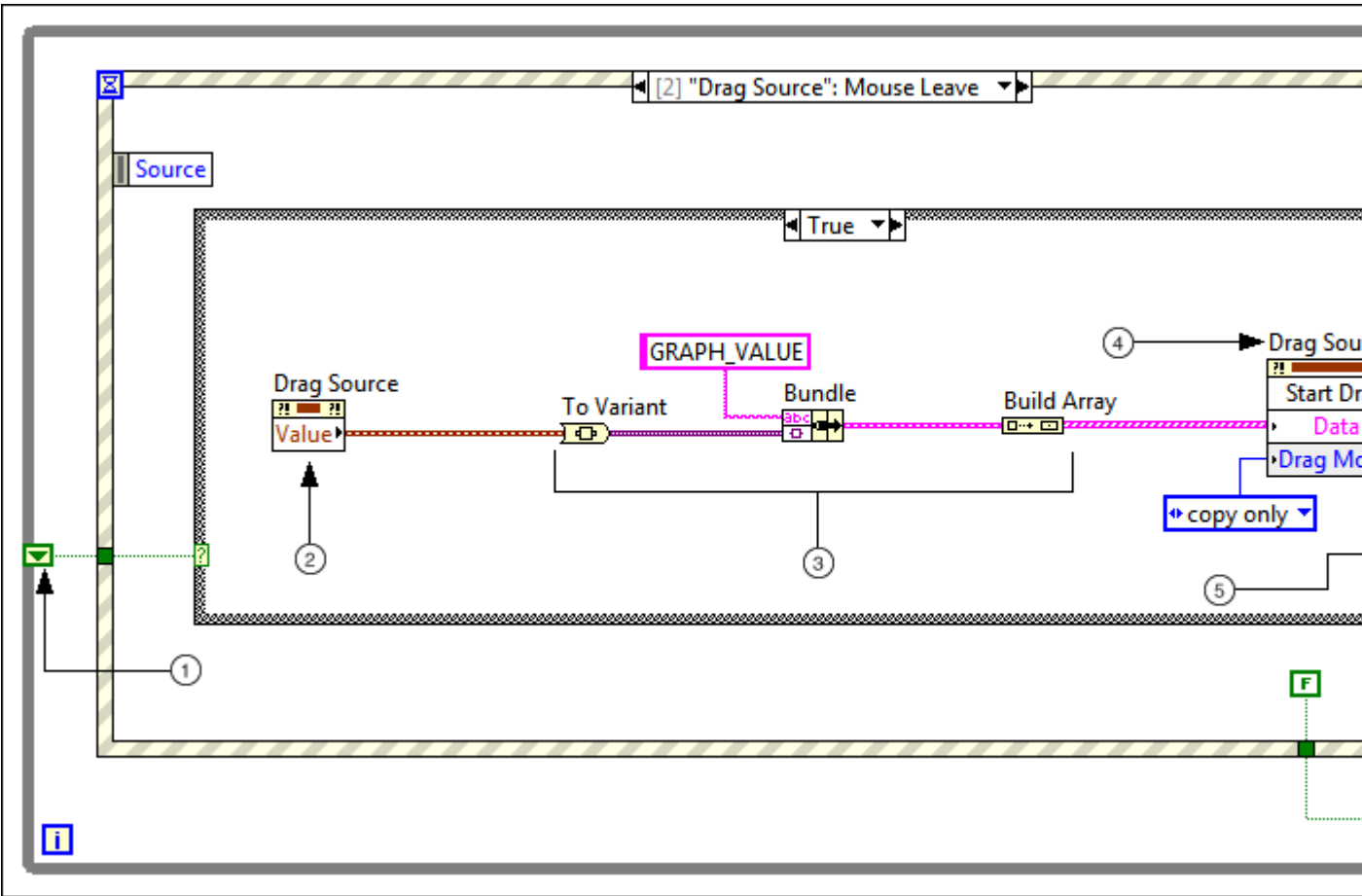
What to Use

Use the following block diagram objects to start the drag by providing the drag data:

Case Structure	Value	To Variant	Bundle	Build Array	Start Drag
					

What to Do

Create the following block diagram to start the drag by providing the drag data.



The following list describes important details about the previous block diagram.



NOTE
When you choose custom data names, do not begin the data names with the prefix LV_. The LV_ prefix is reserved for built-in LabVIEW data types.

After you start the drag-and-drop operation by providing the drag data, *verify that the drop target can accept the drag data*.

Step 5: Verify that the Drop Target Can Accept the Drag Data

To determine the expected result of a drop operation and to indicate the result to the user, verify that the drop target can accept the drag data. By default, LabVIEW changes the cursor to indicate that if the drop occurs now, the drop target updates with the drag data.

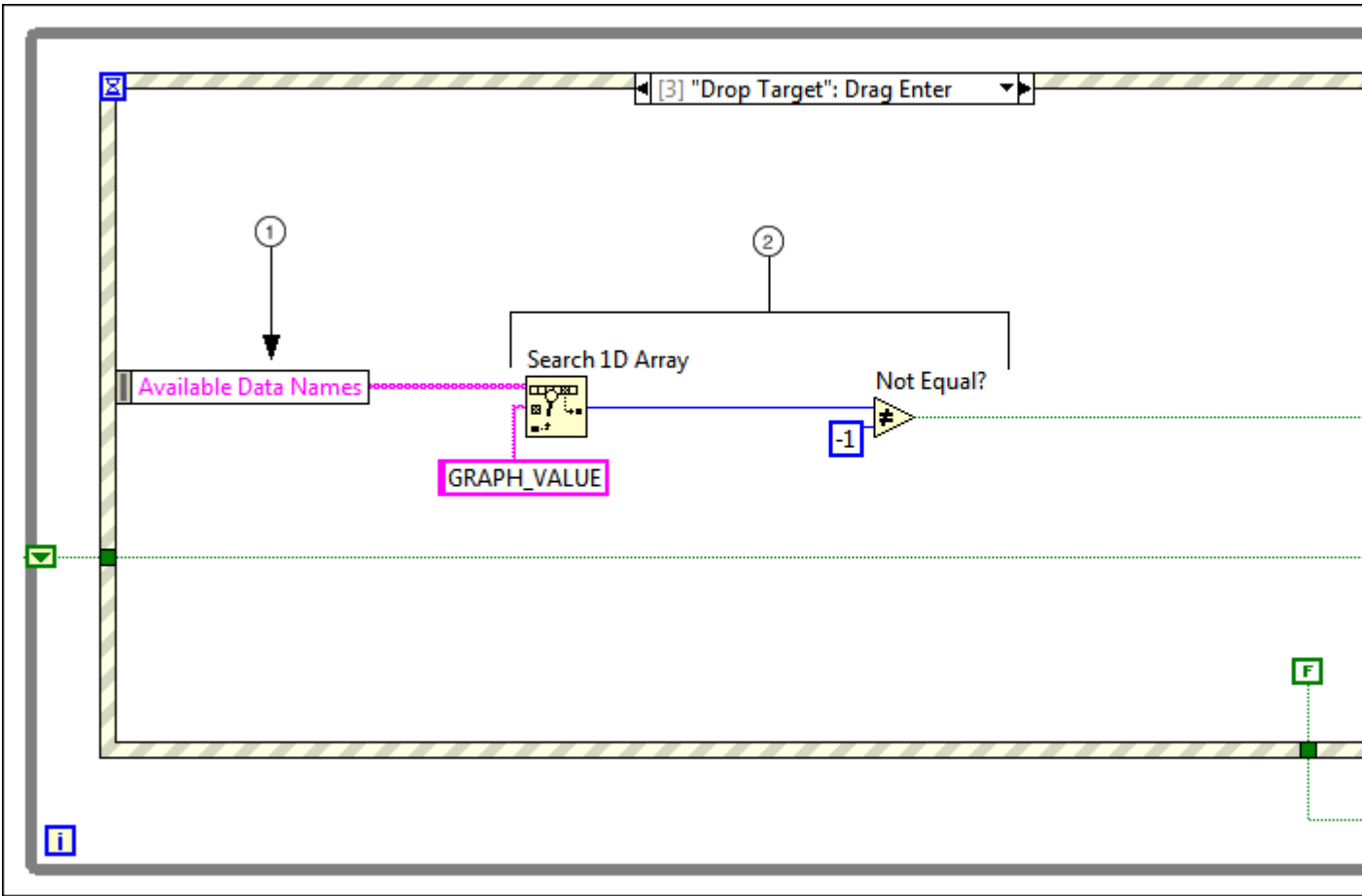
What to Use

Use the following block diagram objects to verify that the drop target can accept the drag data:

Search 1D Array	Not Equal?
	

What to Do

Create the following block diagram to verify that the drop target can accept the drag data.



The following list describes important details about the previous block diagram.

①	The Available Data Names parameter provides an array of strings containing the custom data names you pass to the Start Drag method in the Mouse Leave event.
②	Search the Available Data Names for the custom data name you want the drop target to accept. If the Search 1D Array function finds the data name, the drag data is acceptable.

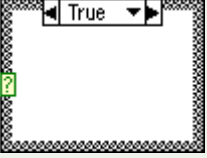
After you verify that the drop target can accept the drag data, *update the drop target with the drag data*.

Step 6: Update the Drop Target with the Drag Data

To update the drop target with the drag data, you must determine whether the drop target accepts the custom data name. If the drop target accepts the data name, retrieve the stored drag data and provide the data to the drop target.

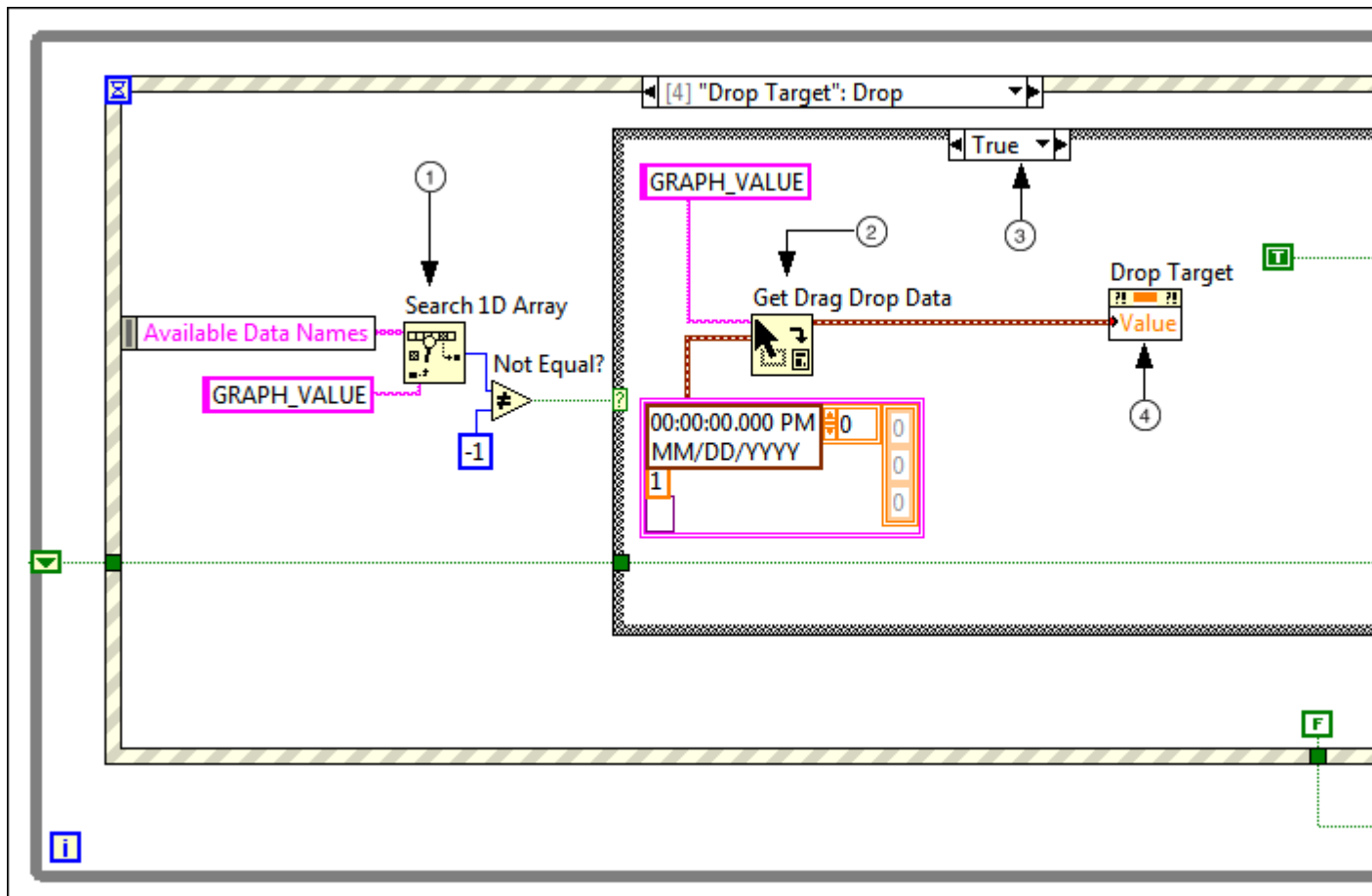
What to Use

Use the following block diagram objects to update the drop target with the drag data:

Search 1D Array	Not Equal?	Case Structure	Get Drag Drop Data	Value
				

What to Do

Create the following block diagram to update the drop target with the drag data.



The following list describes important details about the previous diagram:



TIP

To ensure that you use the correct data type, right-click the drag source on the block diagram and select **Create»Constant**. Wire the constant you create to the **type** parameter of the Get Drag Drop Data function.

Creating Custom Controls and Indicators Using the New Dialog Box

Complete the following steps to create a custom control or indicator from the New dialog box. Refer to the [caveats and recommendations](#) before you begin creating custom controls and indicators.

1. Select **File»New** to display the **New** dialog box.
2. From the **Create New** list, select **Other Files»Custom Control**.
3. Click the **OK** button to display the [Control Editor](#) window.
4. Place a control or indicator in the Control Editor window.
5. Make the changes you want to the control or indicator. You can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.
6. Select **File»Save** to save the custom control or indicator so you can use it in other VIs. You can save the custom control or indicator in a directory or in an [LLB](#). If you do not want to save the custom control or indicator for use in other VIs, close the Control Editor window without saving the custom control or indicator.



NOTE

When you add the custom control or indicator to a front panel, the custom control does not require the `.ctl` file. When you add a [type definition](#) to a front panel, the type definition requires the `.ctl` file.

You also can create a custom control or indicator using the [Edit menu](#).

Arrays and Clusters

Arrays

Array, Matrix & Cluster Controls and Indicators

Use the array, matrix, and cluster controls and indicators on the **Array, Matrix & Cluster** and **Classic Array, Matrix & Cluster** palettes to create arrays and clusters of other controls and indicators and matrices of real or complex data types.

These palettes also contain standard error cluster controls and indicators. The variant control appears on the **Classic Array, Matrix & Cluster** palette.



RELATED INFORMATION

[Clusters](#) on page 1109

[Using Error Clusters](#) on page 1437

[Variant Control and Indicator](#) on page 1049

Adding Elements to Arrays

Complete the following steps to add elements to an array.

1. Right-click the index display and select **Advanced»Show Last Element** from the shortcut menu. The index display changes to the index value of the last element in the array.
2. Click the increment button on the index display to increment the index number of the dimension(s) you want to increase in size. You do not have to increase each dimension. For example, to increase a $2 \times 2 \times 10$ array to $3 \times 3 \times 30$, enter 2, 2, and 29 in the indexes to move to the new last position (2, 2, 29).
3. Assign a value to the new last element. LabVIEW pads the elements between what was the old last element and the new last element with the scalar [default value](#).

If you want to shrink an array and preserve some or all the existing data values, [cut or copy and paste the array cells](#) or [clear the array](#).

Changing Array Default Values

Complete the following steps to manipulate the default values of an array.

1. [Create](#) a 1D array shell in the front panel window.
2. Add a digital numeric control to the array shell in the front panel window.
3. [Resize](#) the array shell to show at least six elements.
4. Set the value of the third element of the array to 1.0. LabVIEW sets the values for the first two elements to 0.0, which is the initial scalar default for the array.
5. Right-click the third element of the array and select **Data Operations»Make Current Value Default** from the shortcut menu to set the scalar default value of the third element to 1.0, whereas it was originally 0.0. Right-click the element itself, not the array index display or array shell.
6. Set the value of the sixth element of the array to 2.0. LabVIEW makes the fourth and fifth elements part of the array and sets their values to 1.0, which you set as the new scalar default for the array in step 5. LabVIEW does not change the values of the first two elements because they were added to the array when the scalar default value was still 0.0. When you add an element to an array by entering its value,

LabVIEW sets the values of the elements between the current end of the array and the element you added to the scalar default.

7. Right-click the array index display or shell border and select **Data Operations»Make Current Value Default** from the shortcut menu to set the default value for the entire array. LabVIEW sets the current values in each element as the default value for those elements. For example, 2 . 0 is now the default value for the sixth element.
8. Right-click the first element of the array and select **Data Operations»Reinitialize to Default Value** from the shortcut menu to change the value of the first element from 0 . 0 to 1 . 0, which is the scalar default value of the entire array.
9. Right-click the array index display or shell border and select **Data Operations»Reinitialize to Default Value** from the shortcut menu. Notice that the value of the first element returns to 0 . 0, which was the value of this element when you set the default value of the entire array.



NOTE

In practice, you rarely need to change the scalar default. Typically you set or restore default values for the entire array.

Changing Array Dimensions

A new array has one dimension and one index display. Use one of the following ways to add more dimensions:

- Right-click the index display and select **Add Dimension** from the shortcut menu to add dimensions one at a time.
- Use the Positioning tool to resize the index display until you have as many dimensions as you want.

Use one of the following ways to delete dimensions:

- Right-click the index display and select **Remove Dimension** from the shortcut menu to delete dimensions one at a time.
- Use the Positioning tool to resize the index display to delete dimensions.

Clearing an Array

Complete the following steps to clear an array control, indicator, or constant of data.

1. Right-click the index display.
2. Select **Data Operations»Empty Array** from the shortcut menu.

Creating Array Controls and Indicators

Create an array front panel control or indicator by placing an array shell and dragging a data object or element, such as a numeric, Boolean, string, path, refnum, or cluster control or indicator, into the shell. The element cannot be another array or a chart. If the element is a graph, its data type cannot be an array, but it can be a cluster that contains arrays.

Complete the following steps to create an array control or indicator.

1. Add an array shell to the front panel. An array shell includes an index display on the left, an element display on the right, and an optional label.
2. If the **Controls** palette is not already visible, right-click the element display or the front panel window to display the **Controls** palette.
3. Drag a control or indicator into the array shell. For example, drag a numeric control into the array shell. If you attempt to drag an invalid control or indicator into the array shell, you are unable to add the control or indicator to the array shell. An example of an invalid control or indicator is an XY graph. You can copy or drag an existing array control from the front panel window to the block diagram window.
4. [Resize the array](#) to add elements to the array.



TIP

You can [export data](#) from arrays to locations such as Microsoft Excel. Right-click an array and select **Export** from the shortcut menu to view available export options.

Default Sizes and Values of Arrays

You cannot limit the size of an array to a fixed number of elements. However, when you [set the default values of an array](#) control, you can set the default size. You set the default values and array size when you create the array control.



NOTE

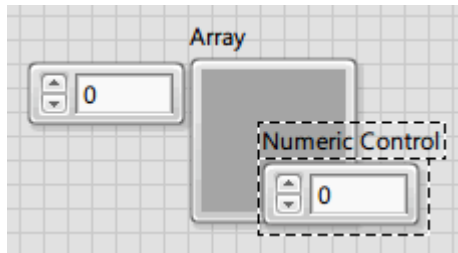
Do not make the default size of the array larger than necessary. If you set a large default size for an array, the default data for each element of the array saves with the VI and increases the size of the VI on disk.

Configuring an Array Control for Use in a VI

An array shell without a data object is undefined and has no data type. The VI will remain broken until you assign a data type by placing a control or indicator in the array shell.

When you place a control or indicator in the array shell, the empty array assumes the data type of the control or indicator. After you assign a data type to an array, you can use it in a VI regardless of whether the array is empty. The length of the empty array is 0, and the array elements are dimmed. You might place an empty array in a top-level VI that calls a subVI to populate the array.

For example, the following image displays an empty array shell about to receive a numeric data type.



Default Value Types of an Array

A front panel array control has the following two default values:

- **Default value for defined elements**– This value is the value of the array when the VI is loaded and is similar to the default value of any other front panel control. The values of the elements in the array compose the default value. The default array value also provides subVIs with a value when the calling VI does not specify a value for the array.
- **Default value for undefined elements**– This value is used to pad the array when the array expands. For example, if you set the array index beyond the defined portion of the array and enter a value for an undefined element, the elements between the defined portion of the array and the new elements you define are set to the default value for undefined elements.

Determining the Size of Arrays

Complete the following steps to determine the size of an array.

1. Right-click the array index display.
2. Select **Advanced»Show Last Element** from the shortcut menu. The index display shows the dimension of the array, thus giving you the size of the array. For example, if you have an array of 15 rows and 15 columns and you complete this step, the index display shows 14 in the upper display and 14 in the lower display.

You also can [change the data size of an array](#).

Displaying an Array in a Table

The index display of a 1D array identifies the left-most visible element.

By default, a new array is a 1D array. The element display shows the single element the index display contains. However, you also can display the array as a table of elements by resizing the array shell.

Complete the following steps to create a 2D numeric array, display it in a table, and use the index displays to alter the display.

1. Add an array shell to the front panel window. An array shell includes an index display on the left, an element display on the right, and an optional label.
2. If the **Controls** palette is not already visible, right-click the element display to display the **Controls** palette.
3. Add a numeric control to the array shell.
4. Right-click the index display and select **Add Dimension** from the shortcut menu to create a 2D numeric array in single-element form. A single-element array displays only one element. Keep both index values at 0 for now. The top index refers to rows, and the bottom index refers to columns.
5. Move the Positioning tool over one of the four corners surrounding the element in the single-element array. The Positioning tool changes shape and resizing handles appear.
6. Use the Positioning tool to resize the element display to add or subtract elements in the array. Resize the array to have two rows and three columns. You also can resize an individual element display. When you resize an element display, all other elements resize to match the one you changed.
7. Use the Labeling tool or the Operating tool to assign values to each numeric control in the array. Assign values 1 through 3 to the first row, 4 through 6 to the second row, and so on.
8. An index display starts at zero. You use the index display to identify the coordinates of the upper left-visible element. Manipulate the index displays the same way you operate a [numeric control](#). Increment the row and column index controls to see how they specify a particular element in the array. The specified element always appears in the upper-left element display.

Think of a 2D array as a row of columns. The upper display of the two boxes on the left is the row index and the lower display is the column index. The combined display to the right of the row and column displays shows the value at the specified position.

The number of indexes required to locate an item of data determines the dimension of an array.

You operate an element of the array in the same manner as you operate a control or indicator of the same type.

Displaying Elements in Arrays

An array can display more or fewer elements than are actually in the array. You can change the number of elements that an array displays at once by resizing the array.

Complete the following steps to resize an array:

1. Move the cursor over the array shell border. Resizing handles appear at the points where you can resize the array.

2. Move the cursor over a resizing handle to change the cursor to the resizing cursor.
3. Use the resizing cursor to drag the resizing handles vertically or horizontally until the dashed border outlines the number of elements you want to display in the array.

**NOTE**

To change which elements the array displays, use the increment and decrement buttons on the index display. You also can enter the index of the first element you want the array to display.

To cancel a resizing operation, continue dragging the resizing handle outside the window until the dashed border disappears or press the <Esc> key before releasing the mouse button.

Displaying or Hiding Index Displays

Complete the following steps to display or hide the index displays.

1. Right-click the index display or the front panel object where you want to display the index.
2. Select the **Visible Items»Index Display** item from the array shortcut menu to display or hide the index display.

You also can use the Index Visible property to show and hide the index display programmatically.

Selecting, Cutting, Copying, and Pasting Array Cells

Complete the following steps to define the cells for selection using the array index.

1. Add an array shell to the front panel.
2. Add a numeric control into the array shell.
3. **Resize the array** so that it contains four rows and four columns.
4. Add a value in the 4, 4 cell to initialize the array.
5. Right-click the array and select **Advanced»Show Selection** from the shortcut menu. Enabling this option shows selected cells, if any exist.
6. Right-click the border of the array shell and select **Add Element Gap** from the shortcut menu to separate the cells by a thin open space. Although it is not necessary to add the element gap, this open border makes it easier to select cells. When you select cells, a thick border to denote the cells you selected fills this open space.
7. Set the array index display to 1, 1. Right-click the index display and select **Advanced»Start Selection** from the shortcut menu.

8. Set the array index display to 3, 3. Right-click the index display and select **Advanced»End Selection** from the shortcut menu. The selection includes the cells denoted by the lower index numbers but not the cells denoted by the higher index numbers. Steps 7 and 8 select the cells from 1, 1 to 2, 2. If you want to include the cells from 1, 1 to 3, 3, set the array index to 4, 4 right-click the index display and select **Advanced»End Selection** from the shortcut menu.



NOTE

Cutting the selected cells removes the entire set of rows and columns spanned by the selection but places only the selected cells into the clipboard. You lose the information contained in the rows and columns surrounding the selection.

1. Right-click and select **Data Operations»Cut Data** from the shortcut menu to cut data or **Data Operations»Copy Data** to copy data.
2. Repeat steps 7 and 8, using the appropriate index display values, to define an area of equal size in the array where you want to paste the data.
3. Right-click the array shell border and select **Data Operations»Paste Data** from the shortcut menu to paste data. After you have pasted the selection, notice that one border remains highlighted around the pasted data. This is an insertion point and remains in the array. To hide this line, right-click the array and select **Advanced»Show Selection** from the shortcut menu to remove the checkmark from this menu item.

Complete the following steps to unmark the cells for selection.

1. Set the index display to 0 in all dimensions.
2. Right-click the index display and select **Advanced»Start Selection** from the shortcut menu.
3. Right-click the index display again and select **Advanced»End Selection** from the shortcut menu.

When in run mode, you can access these menu items when you right-click an array.



Note You cannot *reinitialize, cut, or paste the data* of an indicator in run mode. The **Reinitialize to Default Value**, **Cut Data**, and **Paste Data** shortcut options are available only for controls in run mode.

Setting Array Default Values

LabVIEW arrays have shortcut menu items called **Make Current Value Default** and **Reinitialize to Default Value**. These items are context sensitive. You can use them in two different circumstances—setting the default values of each element in the entire array or setting the default value of one array element.

Complete the following steps to set the default value for an array.

1. Add an array shell to the front panel.
2. Right-click the index display or the array shell border.
3. Select **Data Operations»Make Current Value Default** from the shortcut menu to set the default value of the array to the values that are currently contained in each element.
4. Select **Data Operations»Reinitialize to Default Value** from the shortcut menu to set the default value of the array back to the default value.

Complete the following steps to set the default value for an array element.

1. Add an array shell to the front panel.
2. Right-click an element in the array and select **Data Operations»Make Current Value Default** or **Data Operations»Reinitialize to Default Value** from the shortcut menu. This sets the value of the element to the scalar value that is used for padding when the array expands.

Creating Array Constants

Create an array constant on the block diagram by combining an array with a valid constant, which can be a numeric, Boolean, string, path, refnum, or cluster constant. The element cannot be another array.

1. Add an array constant to the block diagram. An array shell appears, with an index display on the left, an empty element display on the right, and an optional label.
2. Add a constant to the array shell. For example, you can add a numeric constant to the array shell. You can copy or drag an existing array from the front panel window to the block diagram window to create a constant of the same data type. The array shell automatically resizes to accommodate the object you place in the array shell. When you place an object in the array shell, you define the data type of the array constant.

Default Data in Arrays

The **default value** is the value used to pad the array when the array expands. For example, if you set the array index to a value beyond the defined portion of the array and enter a value in an element greater than one row past the defined portion of the array, the elements between the previous end of the array and the element you added are set to the default value.

You can use the Array Size function to determine the size of the array. You can index beyond the bounds of an array inadvertently by indexing an array past the last element using a While Loop, by supplying too large a value to the **index** input of an Index Array function, or by supplying an empty array to an Index Array function.

Deleting Elements, Rows, and Columns within Arrays

You can delete an element within a 1D array and a row or column within a 2D array. To delete an element in a 1D array, switch to the front panel, right-click the array element, and select **Data Operations»Delete Element** from the shortcut menu. To delete a row or column in a 2D array, right-click the array row or column and select **Data Operations»Delete Row** or **Delete Column**.

You also can *programmatically delete elements, rows, columns, and pages* within arrays.

Inserting Elements, Rows, and Columns into Arrays

You can insert an element into a 1D array and a row or column into a 2D array. To add an element to a 1D array, switch to the front panel, right-click the array, and select **Data Operations»Insert Element Before**. To add a row or column to a 2D array, right-click the array and select **Data Operations»Insert Row Before** or **Insert Column Before**.

You also can *programmatically insert elements, rows, columns, and pages* into arrays.

Programmatically Deleting Elements, Rows, Columns, and Pages within Arrays

You can delete an element, row, column, or page within an array programmatically. What you can delete depends on how many dimensions the array has. For example, you can delete a row or a column from an array of two or more dimensions. You can delete a page from an array of three or more dimensions.

Complete the following steps to delete elements, rows, columns, or pages in an array.

1. Add the Delete From Array function to the block diagram.
2. Wire an array of any dimension to the **n-dim array** input of the Delete From Array function. The function automatically resizes based on the dimensions of the array.
3. Determine which operation you want to perform from the following table and complete the associated steps.



NOTE

The **index** input specifies from which element, row, column, or page you want to start deleting, with 0 being the first. The **length** input specifies the number of elements, rows, columns, or pages you want to delete. In the following table, *n* is the total number of elements, rows, columns, pages, or dimensions in the array.

Array wired to n-dimension array	Deleting	Complete these steps.
1D array	Element(s)	Wire a value 1 through n to length . Wire a value 0 through n to index .
2D array	Row(s)	Wire a value 1 through n to length . Wire a value 0 through n to index (row) .
Column(s)	Wire a value 1 through n to length . Wire a value 0 through n to index (column) .	
3D through n D array	Page(s)	Wire a value 1 through n to length . Wire a value 0 through n to index (page) .

1. Run the VI.

Programmatically Inserting Elements, Rows, Columns, and Pages into Arrays

You can insert an element, row, column, or page into an array programmatically. What you can insert depends on how many dimensions the array has. For example, you can insert a row or a column into an array of two or more dimensions, but you cannot insert an element. You can insert a page into an array of three or more dimensions, but you cannot insert an element, row, or column.

Complete the following steps to insert elements, rows, columns, or pages in an array.

1. Add an Insert Into Array function to the block diagram.
2. Wire an array of any dimension to the **n-dim array** input of the Insert Into Array function. The function automatically resizes based on the dimensions of the array.
3. Determine which operation you want to perform from the following table and complete the associated steps.



NOTE

The **index** input specifies the element, row, column, or page where you want to insert the element or array, with 0 being the first. Elements and arrays are added before the value you wire to **index**. In the following table, n is the total number of elements, rows, columns, pages, or dimensions in the array.

Array wired to n-dimension array	Inserting	Complete these steps.
1D array	Element	Wire a value 0 through n to index . Wire a value to new element/subarray .

Array wired to n-dimension array	Inserting	Complete these steps.
1D array to row or column	Wire a value 0 through n to index . Wire a 1D array to new element/subarray .	
2D array	Row	Wire a value 0 through n to index (row) . Wire a 1D array to new element/subarray .
Column	Wire a value 0 through n to index (column) . Wire a 1D array to new element/subarray .	
2D array (rows and columns)	Wire a value 0 through n to index (row) or index (column) . Wire a 2D array to new element/sub array .	
3D through n D array	Page	Wire a value 0 through n to index (page) . Wire a 2D array to new element/sub array .
3D array (pages)	Wire a value 0 through n to index (page) . Wire a 3D array to new element/sub array .	

1. **Resize** the Insert Into Array function to insert another element(s), row(s), column(s), or page(s) in an array and repeat steps 2 and 3.
2. Run the VI.

If the dimension size of the array you insert is smaller than the dimension size of the array you wired to the **n-dim array** input of the Insert Into Array function, the function pads the array you insert with default data, such as zeros in the case of an array of numeric values.

If the dimension size of the array you insert is larger than the dimension size of the array you wired to the **n-dim array** input of the Insert Into Array function, the function crops the array from the end.

Replacing Elements, Rows, Columns, and Pages within Arrays

You can replace an element, row, column, or page in an array. What you can replace depends on how many dimensions the array has. For example, in an array of two or more dimensions, you can replace a row or a column with a 1D array. In an array of three or more dimensions, you can replace a page with a 2D array.

Complete the following steps to replace elements, rows, columns, or pages in an array.

1. Add the Replace Array Subset function to the block diagram.

- 2. Wire an array of any dimension to the **n-dimension array** input of the Replace Array Subset function. The function automatically resizes based on the dimensions of the array.
- 3. Determine which operation you want to perform from the following table and complete the associated steps.



NOTE

The **index** input specifies which element, row, column, or page to replace, with 0 being the first. The **new element/subarray** input specifies the value you want to replace an element, or the array you want to replace a row, column, or page.

Array wired to n-dimension array	Replacing	Complete these steps.
1D array	Element	Wire a value 0- n to index . Wire a value to new element/subarray .
2D array	Row	Wire a value 0- n to index (row) . Wire a 1D array to new element/subarray .
Column	Wire a value 0- n to index (column) . Wire a 1D array to new element/subarray .	
Element	Wire a value 0- n to index (row) . Wire a value 0- n to index (column) . Wire a value to new element/subarray .	
3D- n D array	Page	Wire a value 0- n to index (page) . Wire a 2D array to new element/subarray .
Row	Wire a value 0- n to index (page) . Wire a value 0- n to index (row) . Wire a 1D array to new element/subarray .	
Column	Wire a value 0- n to index (page) . Wire a value 0- n to index (column) . Wire a 1D array to new element/subarray .	
Element	Wire a value 0- n to index (page) . Wire a value 0- n to index (row) . Wire a value 0- n to index (column) . Wire a value to new element/subarray .	

- 1. **Resize** the Replace Array Subset function to replace another element, row, column, or page within an array and repeat steps 2 and 3.
- 2. Run the VI.

Clusters

Clusters

Clusters group data elements of mixed types. An example of a cluster is the LabVIEW error cluster, which combines a Boolean value, a numeric value, and a string. A cluster is similar to a record or a struct in text-based programming languages.



RELATED INFORMATION

[Bundling Elements into Clusters](#) on page 1111

[Converting a Cluster to an Array](#) on page 1115

[Creating Cluster Constants](#) on page 1112

[Replacing Cluster Elements](#) on page 1112

[Unbundling Elements from Clusters](#) on page 1113

Changing Cluster Controls to Indicators and Cluster Indicators to Controls

The first element you place in the element display determines if the cluster is a control or an indicator. Subsequent elements you add convert to match the first element. A cluster can contain controls or indicators, but not a combination. If you try to mix controls and indicators, LabVIEW converts them to match the first element of the cluster.

Complete the following steps to change cluster elements from controls to indicators and vice versa.

1. Right-click any cluster element.
2. Select **Change to Control** or **Change to Indicator** from the shortcut menu to change the cluster and all its elements from indicators to controls or from controls to indicators.

Creating Cluster Controls and Indicators

Create a cluster front panel control or indicator by combining a cluster shell with a valid data object or element, such as Boolean, string, graph, array, numeric, and so on.

Complete the following steps to create a cluster control or indicator.

1. Add a [cluster](#) shell to the front panel window.
2. If the **Controls** palette is not already visible, right-click the element display or the front panel window to display the **Controls** palette.

3. Drag a control or indicator into the cluster shell. For example, drag a numeric control into the cluster shell.

You also can copy or drag an existing **cluster constant** from the block diagram window to the front panel window to create a cluster control.

Modifying Cluster Element Order

Cluster elements have a logical order unrelated to their position in the shell. The first object you place in the cluster is element 0, the second is element 1, and so on. If you delete an element, the order adjusts automatically. The cluster order determines the order in which the elements appear as terminals on the Bundle and Unbundle functions on the block diagram.

Complete the following steps to view and modify the cluster order.

1. Right-click the cluster border and select **Reorder Controls in Cluster** from the shortcut menu. The toolbar and cluster display the following changes:
 - The cluster shows a numbered set of white and black boxes at each element. The white boxes on the elements show their current places in the cluster order.
 - The numbers in the black boxes match the numbers in the white boxes. If you change the order, the black boxes show the new place in the order for an element.
 - The toolbar also displays a number in a black box. The number on the toolbar affects the next element you change.
 - The cursor changes to the cluster order cursor.
2. Use any of these methods to change the order:
 - Click an element in the cluster order. The number of the element in the black box changes to match the number on the toolbar.
 - Type the new order number in the **Click to set to** field and click the element.
 - Click the elements in the order you want to set and the order number automatically increments.
3. Click the **OK** button on the toolbar to set the cluster order and exit the cluster order edit mode. Click the **Cancel** button to revert to the original order.

Resizing Clusters

Complete the following steps to shrink clusters to fit their contents.

1. Right-click a cluster shell border and select **AutoSizing»Size to Fit** from the shortcut menu.

2. Add another element to the cluster shell. When **AutoSizing»Size to Fit** is selected, the cluster resizes itself if necessary when you add another element to the cluster shell.
3. Right-click the cluster shell border and select **AutoSizing»Arrange Horizontally** or **AutoSizing»Arrange Vertically** from the shortcut menu to arrange the elements in the cluster horizontally or vertically.

Setting Cluster Default Values

Complete the following steps to set the default value of the entire cluster to the current value of each individual element.

1. In the front panel window, right-click the cluster shell border.
2. Select **Data Operations»Make Current Value Default** from the shortcut menu.

Complete the following steps to reset all elements to their individual default configurations

1. In the front panel window, right-click the cluster shell border.
2. Select **Data Operations»Reinitialize to Default Value** from the shortcut menu.

Complete the following steps to change the default value of a single element in a cluster.

1. In the front panel window, select the element whose default value you want to set.
2. Enter the new default value for the element.
3. Right-click the element and select **Data Operations»Make Current Value Default** from the shortcut menu.

Bundling Elements into Clusters

Bundling elements into clusters lets you create clusters programmatically.

Complete the following steps to bundle elements into a cluster.

1. Add the Bundle function to the block diagram.
2. If necessary, [resize](#) the Bundle function to include the number of inputs you intend to use as elements in the cluster. You cannot leave an input unwired.
3. Wire front panel control terminals or outputs from VIs and functions to the **element** inputs of the Bundle function. The order in which you wire the inputs determines the cluster element order.
4. Right-click the **output cluster** terminal and select **Create»Indicator**.

LabVIEW returns the bundled cluster in the **output cluster** output.

**TIP**

The Bundle By Name function is an effective tool for creating self-documenting block diagrams.

Creating Cluster Constants

Create a cluster constant on the block diagram by combining a cluster with a valid constant, which can be an array, numeric, Boolean, string, path, refnum, or cluster constant.

1. Add the cluster constant to the block diagram.
2. Add a constant to the cluster shell. For example, you can add a numeric constant to the cluster shell.

When you add objects to the constant shell, you define the data type of the cluster constant.

You also can copy or drag an existing **cluster control** from the front panel window to the block diagram window to easily create a constant with the same element data types.

**NOTE**

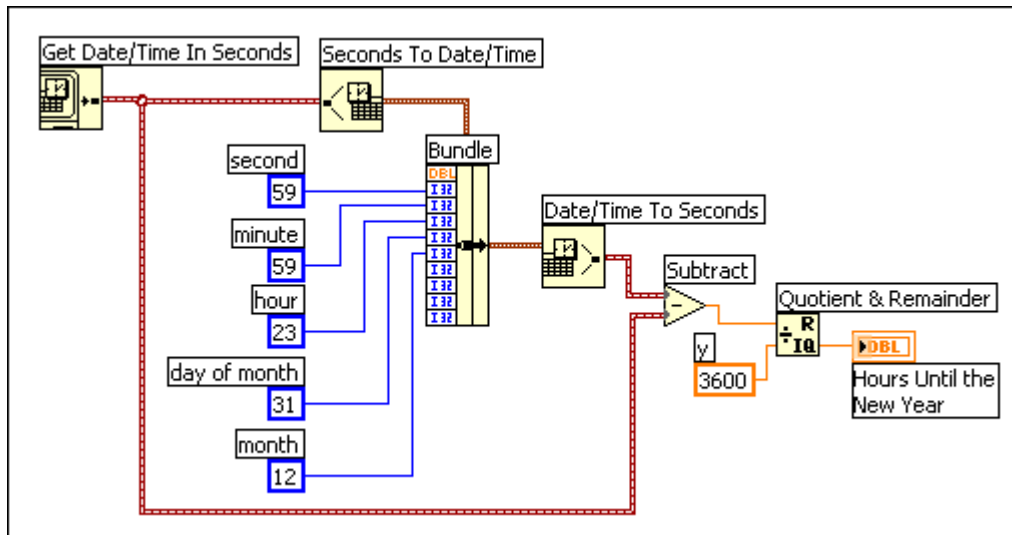
To reduce the size of cluster constants on the block diagram, right-click the cluster constant and select **View Cluster As Icon** from the shortcut menu. To undo this change, select **View Cluster As Icon** again from the shortcut menu. You also can double-click the frame of the cluster constant to view it as an icon. To undo this change, double-click the cluster constant icon.

Replacing Cluster Elements

Complete the following steps to replace or change the value of one or two elements of a cluster without affecting the other elements.

1. Add the Bundle function to the block diagram.
2. Wire the existing cluster to the cluster input of the Bundle function. The Bundle function expands to display the elements of the cluster you wired.
3. Wire the replacement values to the Bundle function.

The following block diagram computes the number of hours until January 1 using the date-time cluster. Because the middle terminal of the Bundle function is wired, the only element input terminals that must be wired are those with replacement values. The values of the unwired elements do not change.



Unbundling Elements from Clusters

Unbundling elements from clusters accesses and arranges all elements in a cluster in their cluster element order. To unbundle the elements, you first must populate the cluster. If you wire an empty cluster control to the Unbundle function, the wire will be void until you populate the cluster. After you unbundle elements from clusters, you can wire each element to VIs, functions, and indicators. This way of unbundling a cluster is useful if you need to access all the elements in a cluster. You also can unbundle all the elements from a cluster *by name*.

Complete the following steps to unbundle elements from a cluster.

1. Add the Unbundle function to the block diagram.
2. Wire a populated cluster to the Unbundle function. The data type representation of every element appears as **element** outputs.



TIP

The Unbundle By Name function is an effective tool for creating self-documenting block diagrams.

Unbundling Elements from Clusters by Name

Unbundling elements from clusters by name accesses and arranges the elements in a cluster by name in their cluster element order. To unbundle a cluster element by name, the element must have a label. After you unbundle an element from a cluster by name, you can wire the element to a VI, a function, or an indicator. This way of unbundling a cluster is useful if you need to access one element from a cluster that includes elements of the same data type. You also can unbundle all the elements from a cluster *without using the name*.

Complete the following steps to unbundle elements from a cluster by name.

1. Add the Unbundle By Name function to the block diagram.
2. Wire a cluster to the Unbundle By Name function. The first element in the cluster element order appears as an **element** output.
3. Select an individual element from the function in one of the following ways:
 - *Resize the function* until the name of the element you want appears.
 - Right-click an **element** output, select **Select Item** from the shortcut menu, and select the element you want.
 - Use the Operating tool to click an **element** output and select the element you want from the shortcut menu.

**NOTE**

When you use the Unbundle By Name function with nested clusters, you can display both the element name and the owning cluster name for elements in the nested cluster by right-clicking the function and selecting **Show Full Names**. This is useful to avoid confusion on the block diagram when element names in nested clusters are identical or similar.

Converting an Array to a Cluster

You can convert an array to a cluster and use Cluster functions to display the contents of the array.

Complete the following steps to convert an array to a cluster.

1. Create a 1D array of numeric controls and add five elements. Assign nonzero values to each element.
2. Add the Array To Cluster function to the block diagram.
3. Wire the array to the Array To Cluster function.
4. Double-click the Array To Cluster function to open the Cluster Size dialog box.
 1. Change the value to 5 to make the cluster size match the size of the array.
 2. Click **OK**.
5. Right-click the right side of the Array To Cluster node and select **Create»Indicator** from the shortcut menu to create a cluster indicator.
6. Run the VI from the front panel. The cluster indicator displays the contents of the array.
7. Modify the values in the array and run the VI again. The cluster indicator displays the new values. Change the index display to view different sections of a large or multidimensional array. To set the value of an element in an array, bring the element into view with the index display and then set the value of the element.

You also can *convert a cluster to an array*.

Converting a Cluster to an Array

You can convert a cluster with elements of the same type to an array and use Array functions to manipulate the contents.

Complete the following steps to convert a cluster to an array.

1. Add a **cluster** to the front panel.
2. Add the Cluster To Array function to the block diagram.
3. Wire the cluster to the Cluster To Array function.
4. Right-click the Cluster To Array function and select **Create»Indicator** from the shortcut menu to create an array indicator.
5. Run the VI from the front panel. The array indicator displays the values of the cluster.

You also can **convert an array to a cluster**.

Moving Arrays and Clusters

Complete the following steps to move an array or a cluster.

1. Use the Positioning tool to click the array shell border, the index display, or the cluster shell border to select the entire array or cluster, not just the element display.
2. Drag the array or cluster to a new location.



NOTE

You also can select the array or cluster by dragging the Positioning tool diagonally across it. After you have selected the array or cluster in this manner, you can move it using the Positioning tool.

If you move only the element in the array or cluster instead of the entire array or cluster, the element separates from the array or cluster because the array shell/cluster shell and its elements are not locked together. Select **Edit»Undo** to cancel the operation or press the <Esc> key before you release the mouse button.

Tabbing through Elements of an Array or Cluster

Complete the following steps to use the <Tab> key to move through elements within a single array or cluster when you run a VI.

1. Display the front panel. While the VI is running, press the <Tab> key until the array or cluster is highlighted.

2. Press the <Ctrl> key and the down arrow key to move the key focus inside the array or cluster. (macOS) Press the <Command> key. (Linux) Press the <Alt> key.
3. Press the <Tab> key to sequence through the array or cluster elements and the array indexes.
4. To return to tabbing among front panel controls, press the <Ctrl> key and the up arrow key. (macOS) Press the <Command> key. (Linux) Press the <Alt> key.

Matrices

Matrices

Matrices group rows or columns of real or complex scalar data for some math operations, such as linear algebra operations. A real matrix contains double-precision elements, and a complex matrix contains complex elements with double-precision components. A matrix can have only two dimensions. You cannot create an array of matrices. Use the Bundle function to combine two or more matrices and create a cluster. Matrices have the same restrictions as arrays.



RELATED INFORMATION

[Converting an Array to a Matrix](#) on page 1117

[Converting a Matrix to an Array](#) on page 1117

[Grouping Data with Arrays and Clusters](#) on page 460

Creating Matrix Controls, Indicators, and Constants

Use matrix front panel controls, indicators, and constants with matrices to group rows or columns of real or complex scalar data for some math operations, such as linear algebra operations.

Complete the following steps to create a matrix control, indicator, or constant.

1. Add one of the following controls to the front panel window:
 - Real matrix
 - Complex matrix



NOTE

When you create a front panel real matrix control or a complex matrix, LabVIEW configures the data type of the control to match the matrix control you select. You can change the type of the matrix by replacing the matrix control.

1. (Optional) You can convert the matrix control to a matrix indicator by right-clicking the control and selecting **Change to Indicator** from the shortcut menu. Matrix controls and indicators operate like array controls and indicators.



Note By default, matrix controls and indicators show more than one element and *display scrollbars* for both dimensions. Because the element type is configured by the matrix control, you can only change the matrix type by replacing the matrix control.

1. (Optional) You also can create a matrix constant. Right-click a matrix control or indicator on the block diagram and select **Change to Constant** from the shortcut menu.

Converting an Array to a Matrix

You can convert a 1D or 2D array of double-precision or complex numbers to a *matrix* and continue to use array functions to manipulate the matrix values. Complete the following steps to convert an array to a matrix.

1. Add a 1D or 2D array of floating-point values to the front panel.
2. Add the Array To Matrix function to the block diagram.
3. Wire the array to the Array To Matrix function.
4. Right-click the Array To Matrix function and select **Create»Indicator** from the shortcut menu to create a matrix indicator.
5. Run the VI. The matrix indicator displays the contents of the original array.

If you wire a 1D array to the Array to Matrix function, the function copies the array elements to the first column of the matrix. The function converts arrays that contain complex elements to complex matrices and converts all other arrays to real matrices. LabVIEW stores matrix elements with double-precision numerical values. You also can *convert a matrix to an array*.

Converting a Matrix to an Array

You can convert a matrix to a 2D array and use Array functions to manipulate the matrix values. Complete the following steps to convert a matrix to a 2D array.

1. Add a real matrix or complex matrix control to the front panel.
 - Real matrix
 - Complex matrix
2. Add the Matrix To Array function to the block diagram.
3. Wire a matrix to the Matrix To Array function.
4. Right-click the Matrix To Array function and select **Create»Indicator** from the shortcut menu to create a 2D array indicator.
5. Run the VI from the front panel. The array indicator displays the contents of the array.

You can store a matrix of a single row or column in a 1D array. However, LabVIEW cannot determine the dimensions of a matrix until run time. The 2D array the Matrix To Array function returns has the same element type, real or complex, as the matrix you wired to the Matrix To Array function. You also can [convert an array to a matrix](#).

Formula Nodes, Expression Nodes, and Script Nodes

Formula Node and Expression Node Functions

The following list contains the names of the Formula Node and Expression Node functions, and their descriptions. Except for the binary functions, you also can use these functions with the Formula VIs and the Nonlinear Curve Fit VI. All function names must be lowercase. You also can use [operators](#) and the `pi` constant in Formula and Expression Nodes.



NOTE

Formula Node and Expression Node functions might not return the same value as functions with the same name in Microsoft Excel, the MATLAB® software, or other text-based math environments.



Note You can use Formula Node most effectively if you are familiar with the C programming language. LabVIEW Help assumes you are familiar with C and does not contain any topics related to programming in C. Refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie or *Teach Yourself C* by Herbert Schildt for more information.

Function	Corresponding LabVIEW Function	Description
absx	Absolute Value	Returns the absolute value of x.
acosx	Inverse Cosine	Computes the inverse cosine of x in radians.
acoshx	Inverse Hyperbolic Cosine	Computes the inverse hyperbolic cosine of x.
asinx	Inverse Sine	Computes the inverse sine of x in radians.
asinhx	Inverse Hyperbolic Sine	Computes the inverse hyperbolic sine of x.
atanx	Inverse Tangent	Computes the inverse tangent of x in radians.
atan2y, x	Inverse Tangent (2 Input)	Computes the arctangent of y / x in radians.
atanhx	Inverse Hyperbolic Tangent	Computes the inverse hyperbolic tangent of x.

Function	Corresponding LabVIEW Function	Description
ceilx	Round Toward +Infinity	Rounds x to the next higher integer (smallest integer $\geq x$).
cosx	Cosine	Computes the cosine of x , where x is in radians.
coshx	Hyperbolic Cosine	Computes the hyperbolic cosine of x .
cotx	Cotangent	Computes the cotangent of x ($1/\tan x$), where x is in radians.
cscx	Cosecant	Computes the cosecant of x ($1/\sin x$), where x is in radians.
expx	Exponential	Computes the value of e raised to the x power.
expm1x	Exponential (Arg) - 1	Computes one less than the value of e raised to the x power ($e^x - 1$).
floorx	Round To -Infinity	Truncates x to the next lower integer (largest integer $\leq x$).
getexpx	Mantissa & Exponent	Returns the exponent of x .
getmanx	Mantissa & Exponent	Returns the mantissa of x .
intx	Round To Nearest	Rounds x to the nearest integer.
intrzx	—	Rounds x to the nearest integer between x and zero.
lnx	Natural Logarithm	Computes the natural logarithm of x (to the base of e).
lnp1x	Natural Logarithm (Arg +1)	Computes the natural logarithm of $x + 1$.
logx	Logarithm Base 10	Computes the logarithm of x (to the base of 10).
log2x	Logarithm Base 2	Computes the logarithm of x (to the base of 2).
maxx, y	Max & Min	Compares x and y and returns the larger value.
minx, y	Max & Min	Compares x and y and returns the smaller value.
modx, y	Quotient & Remainder	Computes the remainder of x / y , when the quotient is rounded toward -Infinity.
powx, y	Power of X	Computes x raised to the y power.
rand	Random Number (0 - 1)	Produces a floating-point number between 0 and 1 exclusively.

Function	Corresponding LabVIEW Function	Description
remx, y	Quotient & Remainder	Computes the remainder of x / y , when the quotient is rounded toward $-\infty$.
secx	Secant	Computes the secant of x , where x is in radians ($1/\cos x$).
signx	Sign	Returns 1 if x is greater than 0, returns 0 if x is equal to 0, and returns -1 if x is less than 0.
sinx	Sine	Computes the sine of x , where x is in radians.
sincx	Sinc	Computes the sine of x divided by x ($\sin x / x$, where x is in radians).
sinhx	Hyperbolic Sine	Computes the hyperbolic sine of x .
sizeofDimary, di	—	Returns the size of the dimension d_i specified for the array ary .
sqrtox	Square Root	Computes the square root of x .
tanx	Tangent	Computes the tangent of x , where x is in radians.
tanhx	Hyperbolic Tangent	Computes the hyperbolic tangent of x .

Precedence of Operators in Formula Nodes and Expression Nodes

The precedence of operators is as follows, from highest to lowest. Operators on the same line all have the same precedence.

**	exponentiation
+, -, !, ~, ++, and --	unary plus, unary negation, logical not, bit complement, pre- and post-increment, pre- and post-decrement ++ and -- are not available in Expression Nodes.
*, /, %	multiplication, division, modulus (remainder)
+ and -	addition and subtraction
>> and <<	arithmetic shift right and shift left
>, <, >=, and <=	greater, less, greater or equal, and less or equal
!= and ==	inequality and equality
&	bit and

<code>^</code>	bit exclusive or
<code> </code>	bit or
<code>&&</code>	logical and
<code> </code>	logical or
<code>? :</code>	conditional evaluation
<code>= op =</code>	assignment, shortcut operate and assign op can be +, -, *, /, >>, <<, &, ^, , %, or **. = op= is not available in Expression Nodes.

The assignment operator = is right associative (groups right to left), as is the exponentiation operator **. All other binary operators are left associative.

The numeric value of TRUE is 1, and FALSE is 0 for output. The logical value of 0 is FALSE, and any nonzero number is TRUE. The logical value of the conditional expression

`<lexpr> ? <texpr>: <fexpr>`

is `<texpr>` if the logical value of `<lexpr>` is TRUE and `<fexpr>` otherwise.



NOTE

You can use Formula Node most effectively if you are familiar with the C programming language. LabVIEW Help assumes you are familiar with C and does not contain any topics related to programming in C. Refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie or *Teach Yourself C* by Herbert Schildt for more information.

Array Manipulation in Formula Nodes

Input arrays and input-output arrays take their type from the array to which they are wired. These arrays do not require you to declare them inside the Formula Node. However, you must declare local arrays and output arrays in the Formula Node.

Arrays are zero-based, as they are in C. Unlike C, LabVIEW treats an assignment to an array element that is out of range as a non-operation, and no assignment occurs. Also unlike C, if you make a reference to an array element that is out of range, LabVIEW returns a value of zero. You must declare array outputs in the Formula Node unless they correspond to an array input, in which case the two terminals must share a name.

If an input terminal is an array, you can use the following functions to determine the shape of the array.

```
numDim = numberOfDims(inputArr);
```

```
firstrowsize = sizeOfDim(inputArr, 0);
```

Array Manipulation Example

Take a typical array declaration, such as

```
int16 a[4][7]
```

After declaring such an array, you can make assignments, such as

```
a[3][2]=455;
```

You also can use array elements in expressions, such as

```
a[0][5]=a[1][6]*3+a[5][0];
```

Changing a Formula Node Terminal Type

Complete the following steps to change a Formula Node terminal from an output to an input or vice versa.

1. Right-click the output or input terminal.
2. Select **Change to Input** or **Change to Output** from the shortcut menu.

Creating Formula Nodes

Complete the following steps to create a Formula Node.

1. Add a Formula Node to the block diagram.
2. Review the available [functions](#) and [operators](#) you can use.
3. Use the Labeling tool or the Operating tool to enter the equations you want to calculate inside the Formula Node. Each assignment must have only a single variable on the left side of the assignment =. Each assignment must end with a semicolon ;. Confirm that you are using the correct [Formula Node syntax](#). If a syntax error occurs, click the broken **Run** button to display the Error list window. LabVIEW marks the syntax error with a # symbol.



TIP

Add comments to the text in a Formula Node by enclosing them inside a slash-asterisk pair /*comment*/, or after a double-slash //comment.

1. Create an input terminal for each input variable by right-clicking the Formula Node border and selecting **Add Input** from the shortcut menu. Type the variable name in the terminal that appears. You can edit the variable name at any time using the Labeling tool or the Operating tool, except when the VI is running. Variable terminals are case sensitive. There is no limit to the number of terminals or equations in a Formula Node. You can [change a terminal type](#) or [remove a terminal](#).

2. Create an output terminal for each output variable by right-clicking the Formula Node border and selecting **Add Output** from the shortcut menu. Type the variable name in the terminal that appears. You can edit the variable name at any time using the Labeling tool or the Operating tool, except when the VI is running. Output variables have thicker borders than input variables.

**NOTE**

No two inputs and no two outputs can have the same name. However, an output can have the same name as an input.

1. (Optional) The default data type for output terminals is double-precision, floating-point. To change the data type, create an input terminal with exactly the same name as the output terminal and wire a data type to that input terminal. Doing so also provides a default value for the terminal. You also can use the [Formula Node syntax](#) to define the variable inside the Formula Node. For example, `int32 y;` changes the data type of the output terminal `y` to 32-bit integer.
2. Wire the input and output terminals of the Formula Node to their corresponding terminals on the block diagram. All input terminals must be wired. Output terminals do not have to be wired.

Formula Node Syntax

The Formula Node syntax is similar to the syntax used in text-based programming languages. Remember to end assignments with a semicolon (;) as in C. Use [scope rules](#) to declare variables in Formula Nodes. Also, keep in mind the [allowed functions](#) and the [allowed operators and their precedence](#) in the Formula Node.

**NOTE**

You can use Formula Node most effectively if you are familiar with the C programming language. *LabVIEW Help* assumes you are familiar with C and does not contain any topics related to programming in C. Refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie or *Teach Yourself C* by Herbert Schildt for more information.

The Formula Node syntax is summarized below using Backus-Naur Form (BNF notation). The summary includes non-terminal symbols: compound-statement, identifier, conditional-expression, number, array-size, floating-point-type, integer-type, left-hand-side, assignment-operator, and function. Symbols in red bold monospace are terminal symbols given exactly as they should be reproduced. The symbol # denotes any number of the term following it.

statement-list:

```
statement ;
statement ; statement-list
```

statement:

assignment
 compound-statement
 control-statement
 conditional-statement
 iterative-statement
 switch-statement
 variable-declaration

assignment:

expression
 left-hand-side assignment-operator assignment

expression:

expression binary-operator expression
 unary-operator expression
 expression unary-operator
 expression ? expression : expression
 (expression)
 identifier
 constant
 function-name (argument-list)

left-hand-side:

identifier
 identifier array-subscription

array-subscription:

[assignment]
 [assignment] array-subscription

assignment-operator: one of the following

= += -= *= /= >>= <<= &= ^= |= %= **=

binary-operator: one of the following

+ - * / ^ != == > < >= <= && || & | % **

unary-operator: one of the following

+ - ! ++ -- ~

argument-list:

expression
expression , argument-list

constant:

pi
number

compound-statement:

{ statement-list }

The following table provides the Formula Node syntax for control, conditional, iterative, and switch statements.

Statement Type	Construct	Grammar	Description/Example
Control Statement	Break Statement	<code>break</code>	Use the break keyword to exit the nearest Do, For, or While Loop or switch statement in the Formula Node.
	Continue Statement	<code>continue</code>	Use the continue keyword to transfer control to the next iteration of the nearest Do, For, or While Loop in the Formula Node.
Conditional Statement		conditional-statement: if-statement if-else-statement	<pre> if (y==x && a[2][3]<a[0][1]) { int32 temp; temp = a[2][3]; a[2][3] = y; y=temp; } else x=y; </pre>
	If Statement	if-statement: <code>if (assignment) statement</code>	
	If-Else Statement	if-else-statement: if-statement <code>else</code> statement	

Statement Type	Construct	Grammar	Description/Example
Iterative Statement		iterative-statement: do-statement for-statement while-statement	
	Do Loops	<code>do statement while (assignment)</code>	<pre>do { int32 temp; temp = --a[2]+y; y=y-1; } while (y!=x && a[2]>=a[0]);</pre>
	For Loops	<code>for ([assignment] ; [assignment] ; [assignment]) statement</code>	<pre>for (y=5; y<x; y*=2) { int32 temp; temp = --a[2]+y; x-=temp; }</pre>
	While Loops	<code>while (assignment) statement</code>	<pre>while (y!=x && a[2]>=a[0]) { int32 temp; temp = --a[2]+y; y=y-1; }</pre>
Switch Statement		<code>switch (assignment) { case-statement-list }</code>	<pre>switch(month) { case 2: days = evenyear? 29: 28; break; case 4:case 6:case9: days = 30; break; default: days = 31;</pre>
	Case Statement List	case-statement-list: case-statement case-statement-list case-statement	

Statement Type	Construct	Grammar	Description/Example
	Case Statement	<pre> case-statement: case number : statement-list default : statement-list </pre>	<pre> break; } </pre>

variable-declaration:

```

type-specifier identifier
type-specifier identifier array-index-list
type-spec identifier = assignment

```

array-index-list:

```

[integer-constant]
[integer-constant] array-index-list

```

type-specifier:

```

floating-point-type
integer-type

```

floating-point-type:

```

float
float32
float64

```

integer-type:

```

int
int8
int16
int32
uInt8
uInt16
uInt32

```

non-digit: one of the following

```

_ a~z A~Z

```

digit: one of the following

```

0 1 2 3 4 5 6 7 8 9

```

nonzero-digit: one of the following

1 2 3 4 5 6 7 8 9

binary-digit: one of the following

0 1

octal-digit: one of the following

0 1 2 3 4 5 6 7

hex-digit: one of the following

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

identifier:

non-digit [non-first-character]

non-first-character:

non-digit [non-first-character]
digit [non-first-character]

number:

integer-constant
float-constant

integer-constant:

decimal-constant
binary-constant
octal-constant
hex-constant

decimal-constant:

nonzero-digit #digit

binary-constant:

0b #binary-digit
0B #binary-digit

octal-constant:

0 #octal-digit

hex-constant:

0x #hex-digit
0X #hex-digit

float-constant:

fraction exponent-part
 decimal-constant exponent-part

fraction:

#digit . digit #digit

exponent-part:

e [sign] #digit
E [sign] #digit

sign: one of the following

+ **-**

comment:

//comment
 /* comment */

Removing a Terminal from a Formula Node

Complete the following steps to remove a terminal from a Formula Node.

1. Right-click the input or output terminal you want to remove.
2. Select **Remove** from the shortcut menu.

You also can select the terminal and press the <Delete> or <Backspace> key.

Scope Rules for Declaring Variables in Formula Nodes

Formula Nodes use the same scope rules for declaring variables as C.

- All variables you declare in bracketed blocks are only accessible within the bracketed block.
- All input terminals are considered variables at the outermost block (not enclosed in brackets), and cannot be declared again in the outermost block.
- LabVIEW tries to match variables you declare at the outermost block (not enclosed in brackets) to output terminals with the same name.
- You can declare output terminals that have no corresponding input terminal and are not declared at the outermost block.

Use the correct Formula Node syntax when declaring variables.



NOTE

You can use Formula Node most effectively if you are familiar with the C programming language. LabVIEW Help assumes you are familiar with C and does not contain any topics related to programming in C. Refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie or *Teach Yourself C* by Herbert Schildt for more information.

Related Information

[Formula Node Syntax](#)

Creating Expression Nodes

Use the Expression Node to calculate expressions that contain a single variable. Expression Nodes are useful when an expression has only one variable but is otherwise complicated.

Expression Nodes use the value you pass to the input terminal as the value of the variable. The output terminal returns the value of the calculation.

Complete the following steps to create an Expression Node.

1. Add an Expression Node to the block diagram.
2. Use the Labeling tool or the Operating tool to type the expression inside the Expression Node. You cannot have more than one variable in the expression. Use a [syntax](#) similar to the syntax used in most text-based programming languages for arithmetic expressions. If a syntax error occurs, click the broken **Run** button to display the Error list window. LabVIEW marks the syntax error with a # symbol.
3. Wire the VI, function, control, or constant terminal that provides the value for the variable to the input terminal of the Expression Node. You also can create a control or constant by right-clicking the input terminal of the Expression Node and selecting **Create»Control** or **Create»Constant** from the shortcut menu.
4. Wire the output terminal of the Expression Node to the VI, function, or indicator terminal you want to receive the value of the expression. You also can create an indicator by right-clicking the output terminal of the Expression Node and selecting **Create»Indicator** from the shortcut menu.

Expression Node Syntax

The following summary of the Expression Node syntax uses Backus-Naur Form (BNF notation). The Expression Node uses C operations, but adds the exponent operator, **.

```

<expr> : = <expr> <binaryoperator> <expr>
        | <unaryoperator> <expr>
        | <expr> ? <expr> : <expr>
        | ( <expr> )
        | <identifier>
        | <const>
        | <function> ( <arglist> )
<binaryoperator>: = + | - | * | / | ^ | != | == | > | < |
                 >= | <= | && | || | & | | | ^ | **
<unaryoperator>: = + | - | ! | ~
  
```



NOTE

To include optional items in the Expression Node syntax, use square brackets to enclose the items.

Script Nodes

Use script nodes to execute math scripts in LabVIEW. LabVIEW provides a script node, called the MATLAB® script node, that facilitates the processing of your scripts written in the MATLAB language syntax.



NOTE

You must have a licensed copy of the MATLAB software version 6.5 or later installed on your computer to use the MATLAB script node because this script node invokes the MATLAB software script server to execute the scripts. The MATLAB script node is available only on Windows.

Adding Scroll Bars to Script Nodes

Right-click inside a script node and select **Visible Items»Scrollbar** from the shortcut menu. A scroll bar appears on the right side of the script node.

Calling the MATLAB® Software from LabVIEW

You must have a licensed copy of the MATLAB® software version 6.5 or later installed to use MATLAB script nodes because the script nodes invoke the MATLAB software script server to execute scripts written in the MATLAB language syntax. The MATLAB script nodes are available only on Windows.

Complete the following steps to create and run a VI that uses a script you wrote in the MATLAB language syntax.

1. Add a MATLAB script node to the block diagram.

**NOTE**

You can create MATLAB script nodes only in the LabVIEW Full and Professional Development Systems. If a VI contains a MATLAB script node, you can run the VI in all LabVIEW packages.

1. Use the Operating or Labeling tool to enter the following script in the MATLAB script node: `a=rand(50) surf(a)`
2. Add an output to the MATLAB script node and create an indicator for the output.
 1. Right-click the right side of the MATLAB script node frame and select **Add Output** from the shortcut menu.
 2. Enter `a` in the output terminal to add an output for the `a` variable in the script.
 3. Verify the data type of the output terminal. In the MATLAB script node, the default data type for any new input or output is Real. Right-click the `a` output and select **Choose Data Type»2-D Array of Real** from the shortcut menu.
 4. Right-click the `a` output terminal and select **Create»Indicator** from the shortcut menu to create a 2D array of numeric indicators labeled **2-D Array of Real**.
3. Right-click the **error out** output terminal and select **Create»Indicator** from the shortcut menu to create an **error out** indicator labeled error out.
4. Resize the **2-D Array of Real** front panel indicator so you can see the numbers the script generates when the VI runs.
5. Run the VI. LabVIEW invokes the [MATLAB software script server](#), creates a matrix of random values, displays the matrix in the MATLAB software, plots that information to a graph, and displays the values that make up the matrix in the **2-D Array of Real** front panel indicator.
6. (Optional) [Debug](#) your script written in the MATLAB language syntax.

You can [display a scroll bar](#) within a MATLAB script node.



Note A script inside a MATLAB script node behaves as it does in the MATLAB software environment.

Changing Script Servers in Script Nodes

The MATLAB® software installs a script server engine that runs your scripts written in the MATLAB language syntax. Other third-party software also might install a script server engine that runs scripts. LabVIEW communicates with these script server engines through a corresponding script node. You change the script server engine with which LabVIEW communicates by converting the script node.

Complete the following steps to change the script server engine with which LabVIEW communicates.

1. Right-click the border of a MATLAB script node or third-party script node.
2. Select **Choose Script Server** from the shortcut menu and select the script server engine with which you want LabVIEW to communicate.

Clearing Scripts

Complete the following steps to clear the contents of a script node.

1. Right-click a script node.
2. Select **Clear Script** from the shortcut menu.

Configuring the Data Type of Script Node Terminals

You can configure the data type of *MATLAB script* node terminals.

MATLAB Script Nodes

The MATLAB language syntax is a loosely-typed script language that does not determine the data type of a variable until after the script runs. Therefore, LabVIEW cannot determine the type of a variable in edit mode. LabVIEW queries the script server to find out possible data types. You can select the LabVIEW data type for each terminal.



NOTE

If you do not correctly configure a variable data type, LabVIEW might produce an error, return incorrect information at run time, or both.

Complete the following steps to change the data type of an input or output terminal on a MATLAB script node.

1. Right-click the terminal of the input or output and select **Choose Data Type** from the shortcut menu. A list of the available data types appears.
2. Select the data type you want to use.

Debugging Scripts

Use *script nodes* to execute math scripts in LabVIEW. LabVIEW provides a script node called the MATLAB® script node, that facilitates the processing of your scripts written in the MATLAB language syntax.

To make debugging a script easier, use the following programming techniques:

- Write the script and run it in the native program for testing and debugging purposes before you import it into LabVIEW.

- **Verify data types.** When you create a new input or output, make sure that the data type of the terminal is correct. Also, create controls and indicators for inputs and outputs so you can monitor what values are being passed between LabVIEW and the script server engine. This allows you to pinpoint where a script node calculates a value incorrectly, if necessary.
- Take advantage of the error-checking parameters for debugging information. Create an indicator for the **error out** terminal on a MATLAB script node before you run any VI so you can view the generated error information at run time.

Importing or Exporting Your Scripts

If you already have a script you wrote in another text-based syntax, such as other .m file language syntaxes, you can import it into LabVIEW. You also can save a script from LabVIEW to a text file.

Complete the following steps to import a script into another **script node** in LabVIEW.

1. Select a MATLAB script Node and drag out a region to add the script node to the block diagram.



NOTE

You can create MATLAB script nodes only in the LabVIEW Full and Professional Development Systems. If a VI contains a MATLAB script node, you can run the VI in all LabVIEW packages. You must have a licensed copy of the MATLAB® software version 6.5 or later installed on your computer to use a MATLAB script node.

1. Right-click the script node and select **Import** from the shortcut menu to display a file dialog box.
2. Select the file you want to import and click the **OK** button. The script text appears in the node.

Complete the following steps to save a script to a text file.

1. Right-click the script node and select **Export** from the shortcut menu to display a file dialog box.
2. Enter the name to use for the text file or select a file to overwrite.
3. Click the **OK** button.

Showing and Hiding Line Numbers in Script Nodes

You can show and hide line numbers in the Formula Node and the MATLAB script node.

Complete the following steps to show or hide line numbers in a script node.

1. Right-click inside the node.

2. Select **Visible Items»Line Numbers** from the shortcut menu.

Variables

Read and Write Variables

After you create a local or global variable, you can read data from a variable or write data to it. By default, a new variable receives data. This kind of variable works as an indicator and is a write local or global variable. You also can configure a variable to behave as a data source, or a read local or global variable.

On the block diagram, you can distinguish read local or global variables from write local or global variables the same way you distinguish controls from indicators. A read local or global variable has a thick border similar to a control. A write local or global variable has a thin border similar to an indicator.



TIP

Write a value to a global variable before reading from it so that it has a known initial value when you access it.



RELATED INFORMATION

[Creating Local Variables](#) on page 1136

[Creating Global Variables](#) on page 1137

[Changing the Data Direction of a Local Variable](#) on page 1135

[Changing the Data Direction of a Global Variable](#) on page 1137

Local Variables

Changing the Data Direction of a Local Variable

Use the **Change To Read** and **Change To Write** shortcut menu items to change a read local to a write local or vice versa, which changes the data direction of the local variable.

Complete the following steps to change the configuration of a local variable so that it acts as a read local and then change it back to a write local.

1. Add a string control to the front panel.
2. Use the Labeling tool to label the control Name.
3. Right-click the control and select **Create»Local Variable** from the shortcut menu. A local variable for the string control appears on the block diagram. By default, the local variable is a write local and has a thin border.

4. On the block diagram, right-click the local variable node and select **Change To Read** from the shortcut menu to configure the local variable to work as a control. The local variable changes to a read local and has a thick border.
5. Right-click the local variable node again and select **Change To Write** from the shortcut menu to make the variable a write local.

Creating Local Variables

You might need to use a local variable when *writing to one front panel object* from two block diagram locations or when reading from one front panel object to two block diagram locations.

Right-click an existing front panel object or block diagram terminal and select **Create»Local Variable** from the shortcut menu to create a local variable that you can add to the block diagram.

Complete the following steps to create a local variable and associate it with a front panel object.

1. Add a local variable to the block diagram. The local variable node is not yet associated with a control or indicator.
2. Right-click the local variable node and select a front panel object from the **Select Item** shortcut menu. The shortcut menu lists all the front panel objects that have owned labels. You also can use the Operating tool or Labeling tool to click the local variable node and select the front panel object from the shortcut menu.



NOTE

LabVIEW uses owned labels to associate local variables with front panel objects, so label front panel controls and indicators with descriptive owned labels.



[Data Communication Methods Home](#)

Memory Considerations when Using Local Variables

Local variables make copies of data buffers. When you read from a local variable, you create a new buffer for the data from its associated control.

If you use local variables to transfer large amounts of data from one place on the block diagram to another, you generally use more memory and, consequently, have slower execution speed than if you transfer data using a wire. If you need to store data during execution, consider using a *shift register*.

Global Variables

Changing the Data Direction of a Global Variable

Use the **Change To Read** and **Change To Write** shortcut menu items to change a read global to a write global or vice versa, which changes the data direction of the global variable.

Complete the following steps to change the configuration of a global variable so that it acts as a read global and then change it back to a write global.

1. Add a global variable to the block diagram. By default, the global variable is a write global and has a thin border.
2. Right-click the global variable node and select **Change To Read** from the shortcut menu to configure the global variable to work as a control. The global variable changes to a read global and has a thick border.
3. Right-click the global variable node again and select **Change To Write** from the shortcut menu to make the variable a write global.

Creating Global Variables

You can create several single global VIs, each with one front panel object, or if you want to group similar variables together, you can create one global VI with multiple front panel objects.

Complete the following steps to create a global VI with multiple front panel objects.

1. Add a global variable to the block diagram.
2. Use the Operating tool or Positioning tool to double-click the global variable node to display the front panel window of the global VI.
3. Add controls and indicators to the front panel window the same way you do in a standard front panel window.



NOTE

LabVIEW uses **owned labels** to identify global variables, so label the front panel controls and indicators with descriptive owned labels.

1. Save and close the global VI.
2. Return to the block diagram of the original VI and right-click the global variable node and select a front panel object from the **Select Item** shortcut menu. The shortcut menu lists all the front panel objects that have owned labels. You also can use the Operating tool or Labeling tool to click the global variable node and select the front panel object from the shortcut menu.

Complete the following steps to use a global variable in other VIs.

1. Click the **Select a VI** icon or text on the **Functions** palette, navigate to the directory where you saved the global VI, and double-click the VI.
2. Add the global variable to the block diagram. By default, the global variable is associated with the first front panel object with an owned label you added to the global VI.
3. Right-click the global variable node you added to the block diagram and select a front panel object from the **Select Item** shortcut menu to associate the global variable with the data from another front panel object.



Note Use a global variable only when you cannot route a wire through multiple VIs to share data. For example, if a file path is changed in one VI and another VI must open that path, there might not be a logical way to wire the path through numerous calling VIs. Using a global variable to hold the path data might be the best choice.



[Data Communication Methods Home](#)

Creating Template Global Variables

Complete the following steps to create a template you can use to create global variables.

1. [Create a global variable](#) to include only the front panel objects you want to reuse.
2. Select **File»Save As** to display a file dialog box.



NOTE

If you have already saved the global variable, selecting **File»Save As** displays the Save As dialog box instead. Select the **Copy** and **Substitute copy for original** radio buttons and click the **Continue** button to display the file dialog box.

1. (Windows) Select **Template VIs** from the **Save as type** pull-down menu in the file dialog. (macOS and Linux) Include the template VI file extension `.vit`.
2. Enter the name of the template global variable and click the **OK** button.



Caution Do **not** save your own VIs and other files in [.llb files installed with LabVIEW](#).

Memory Considerations when Using Global Variables

When you read from a global variable, LabVIEW creates a copy of the data stored in that global variable.

When you manipulate large arrays and strings, the time and memory required to manipulate global variables can be considerable. Manipulating global variables is especially inefficient when dealing with arrays because if you modify only a single array element, LabVIEW stores and modifies the entire array. If you read from the global variable in several places in an application, you create several memory buffers, which is inefficient and slows performance.

Shared Variables

Configuring Firewalls and Network Address Translating Routers for Shared Variables

You must configure firewalls and Network Address Translating (NAT) routers if you want to transmit network-published [shared variables](#) through them. If you do not correctly configure the firewall or router, network-published shared variables might not work.

Network-published shared variable servers use the TCP port 2343 and the UDP port 2343, in addition to one TCP port for each application that you run. The [NI Publish-Subscribe Protocol \(NI-PSP\)](#) protocol begins looking for available TCP ports at port 59110 and increments upward until it finds an available port. The number of available ports you need to configure depends on the number of applications you run. You manually can [configure the range of TCP ports the NI-PSP protocol uses](#) by creating a LogosXT.ini file and entering new values in the LogosXT_BasePort and LogosXT_NumPortsToCheck parameters. (Phar Lap ETS, VxWorks) For Phar Lap ETS and VxWorks targets, configure the range of TCP ports in the ni-rt.ini configuration file.

If possible, use TCP for transmitting network-published shared variables.



NOTE

(Optional) You can still use UDP ports for the network-published shared variables servers. In addition to the required UDP port 2343, the servers use a range of UDP ports beginning with port 6000. The number of ports above 6000 that the network-published shared variable servers use depends on the number of servers running on the computer. Network-published shared variable clients use a range of UDP ports beginning with port 5000. The number of ports above 5000 that the network-published shared variable clients use depends on the number of client applications running on the computer. To find out the number of ports the server or client uses, first use [TCP](#) without the firewall configured to find out how many ports you need.

Complete the following steps to allow clients outside a firewall to connect to a server inside the firewall. The exact steps for configuring the firewall depend on the specific firewall you use. Refer to the documentation for the firewall for specific instructions about opening ports.

1. Open TCP port 2343 and UDP port 2343 for incoming packets, allowing server-discovery broadcasts.
2. Starting at TCP port 59110, open one TCP port for each application you run.
3. **(Optional)** Open the UDP port range from 6000 to 6010 for incoming packets.

You also can allow clients inside a firewall to connect to a server outside the firewall. Some firewalls automatically transmit incoming packets without any configuration. **(Optional)** If the firewall does not open the ports automatically, open the UDP port range from 5000 to 5010 for incoming packets.

Complete the following steps to allow clients outside a NAT router to connect to a server inside the router. The exact steps for configuring the router depend on the specific router you use. Refer to the documentation for the router for specific instructions about forwarding ports. Only one computer inside the router can function as a server.

1. Forward TCP port 2343 and UDP port 2343 to the computer running the server.
2. Starting at TCP port 59110, forward one TCP port for each application you run.
3. **(Optional)** Forward the UDP port range from 6000 to 6010 to the computer running the server.

You also can allow clients inside a NAT router to connect to a server outside the router. Some routers automatically forward ports without any configuration. **(Optional)** If the router does not forward the ports automatically, forward the UDP port range from 5000 to 5010 to the computer running the client.



Note If you deploy a network variable to a Windows machine with the Windows firewall enabled, you must add `\National Instruments\Shared\Tagger\tagsrv.exe` and `\windows\SysWOW64\lkads.exe` to the Windows firewall exception list in order to read or write to the variable hosted on the machine.

Scaling Shared Variables

You can scale the value of a shared variable with the following data type:

- Numeric
- Boolean
- U32 Bit Field



NOTE

You can use scaling only for **network-published** shared variables, I/O variables, or I/O aliases. When the shared variable you want to scale is a network-published shared variable, it must have a data source. Configure a network-published shared variable to have a data source by placing a checkmark in the **Enable Aliasing** checkbox on the Variable page of the Shared Variable Properties dialog box.

Scaling Numeric Shared Variables

Complete the following steps to scale numeric shared variables.

1. Right-click a shared variable in the *Project Explorer* window and select **Properties** from the shortcut menu to display the **Shared Variable Properties** dialog box.
2. Select **Scaling** to display the Scaling page.
3. Place a checkmark in the **Enable Scaling** checkbox.
4. Select one of the following scaling options from the **Scale Type** pull-down menu.
 - **Linear** —Enables a linear $mx + b$ conversion between raw and engineering ranges.
 - **Square Root** —Enables a square root conversion between the raw **Raw Zero Scale** and **Raw Full Scale** and engineering **Engineering Zero Scale** and **Engineering Full Scale** ranges, which is $b + m * \sqrt{\text{raw} - o}$ where $b = \text{Engineering Full Scale} - \text{Engineering Zero Scale}$, $m = \frac{\text{Engineering Full Scale} - \text{Engineering Zero Scale}}{\sqrt{\text{Raw Full Scale} - \text{Raw Zero Scale}}}$, and $o = \text{Raw Zero Scale}$.
5. (Optional) In the **Engineering Unit** text box, specify the unit of data measurement, such as degrees Celsius, pounds, grams, and so on.
6. Specify the raw and engineering ranges in the **Raw Full Scale**, **Raw Zero Scale**, **Engineering Full Scale**, and **Engineering Zero Scale** numeric controls.
7. (Optional) Place a checkmark in the **Coerce to Range** checkbox to coerce the data to the specified range.
8. Click the **OK** button to close the **Shared Variable Properties** dialog box.

Scaling Boolean Shared Variables

Shared variables with a data type of Boolean can have *inverted scaling*. Complete the following steps to scale Boolean shared variables.

1. Right-click a shared variable in the *Project Explorer* window and select **Properties** from the shortcut menu to display the **Shared Variable Properties** dialog box.
2. Select **Scaling** to display the Scaling page.
3. Place a checkmark in the **Enable Scaling** checkbox.
4. Place a checkmark in the **Invert** checkbox for the *Shared Variable Engine* (SVE) to invert the Boolean value when the SVE communicates with the device server.
5. Click the **OK** button to close the **Shared Variable Properties** dialog box.

Scaling U32 Bit Field Shared Variables

Shared variables with a data type of U32 Bit Field can have invert-mask and/or select-mask scaling. Complete the following steps to scale bit array shared variables.

1. Right-click a shared variable in the *Project Explorer* window and select **Properties** from the shortcut menu to display the **Shared Variable Properties** dialog box.
2. Select **Scaling** from the list of this dialog box to display the Scaling page.
3. Place a checkmark in the **Enable Scaling** checkbox.
4. In the **Scaling Invert Mask** text box, set the bits you need to scale. LabVIEW masks the bits you do not set and reports these masked bits in an array as zeros.

5. In the **Scaling Select Mask** text box, set which bits are inverted between the device server and the *Shared Variable Engine*. You can invert only the bits specified in this text box.
6. Click the **OK** button to close the **Shared Variable Properties** dialog box.

(DSC Module) To scale shared variables programmatically, use the `SharedVariableIO` properties.

Resources Exposed by the Variable Web Service

The variable Web service exposes the following resources:

VariableObject — A collection of `VariableObject` types containing the following properties:

- `Url [string]` — The dynamic API URL of the `VariableObject`. The `Url` property uniquely identifies the `VariableObject`.
- `Type [string]` — The type of the `VariableObject`. The available types are "container" and "variable".
- `Name [string]` — The name of the `VariableObject`.
- `Children [VariableObject []]` — An array of `VariableObjects` representing the children of the current `VariableObject`. This property is deferred by default. Use the `$expand` parameter to represent this property inline.

Variable — Inherits `VariableObject` and adds the following properties:

- `Access [string]` — The Access permission of the `Variable`. The available permissions are "Read", "Write", and "Read/Write".
- `DataType [string]` — The data type of the `Variable`. The variable Web service supports primitive types and arrays of primitive types. For complex data types, use the serialized XML string or JSON string as the value of the `Value` property.
- `VariableValue [string]` — The additional properties of the current `Variable`. This property is deferred by default. Use the `$expand` parameter to represent this property inline.

VariableValue — Contains the following additional `Variable` properties:

- `Url [string]` — The dynamic API URL of the corresponding `Variable`. The `Url` property uniquely identifies the `VariableValue`.
- `Access [string]` — The Access permission of the `Variable`. The available permissions are "Read", "Write", and "Read/Write".
- `Quality [int64]` — The raw quality bits of the `Variable`.

- **ErrorStatus** [int32] — The error status code of the **Variable**.
- **DataType** [string] — The data type of the **Variable**. The variable Web service supports primitive types and arrays of primitive types. For complex data types, use the serialized XML string or JSON string as the value of the **Value** property.
- **Value** [string] — The string version of the **Value** of the **Variable**.
- **Timestamp** [string] — The string representation of the **Variable** timestamp in **Date Time** format.
 - **JSON** — A serialized string representing milliseconds after 1970/1/1 0:0:0, UTC time.
 - **XML** — A string formatted according to RFC 3339. For example, 2011-05-25T17:04:30

VariableValueGroup — Used to get variable updates and batch read variables, this user-created group contains the following properties:

- **Id** [string] — The ID of the group. The **Id** property uniquely identifies the **VariableValueGroup**.
- **Timeout** [int32] — The number of milliseconds of inactivity that closes the variable connection. The value of this property is -1, representing no timeout, by default.
- **VariableValues** [**VariableValue** []] — The list of **VariableValues** included in the **VariableValueGroup**. This property is deferred by default. Use the **\$expand** parameter to represent this property inline.

Finding Out Basic Information about a Shared Variable

Required permission: VariableBrowse

To find out the name, type, and URL of a shared variable, you can browse the corresponding **VariableObject**. To browse a **VariableObject**, use the following URI format: `http://[host]:[port]/nivariable/VariableObjects('[dynamic api url]')`

For example, GET `http://my_server:3580/nivariable/VariableObjects('ni.var.psp://localhost')?$format=json`



NOTE

The example commands return JSON responses only. To return XML responses, remove `$format=json` from the commands.

Follow the deferred children URI to view the **Children** property of the **VariableObject**.

For example, GET `http://my_server:3580/nivariable/VariableObjects('ni.var.psp://localhost')/Children?$format=json`

Use the `$expand` parameter to get the Children property represented inline without having to make a separate request to get the children. For example, GET `http://my_server:3580/nivariable/VariableObjects('ni.var.psp://localhost')?$format=json&$expand=children`

To get only one property of a VariableObject, use the following URI format:

GET `http://[host]:[port]/nivariable/VariableObjects('[dynamic api url]')/[specific property]`

The available properties are `Url`, `Type`, and `Name`. For example, to get the `type` property of `ni.var.psp://localhost`, use the following URI:

GET `http://my_server:3580/nivariable/VariableObjects('ni.var.psp://localhost')/type?$format=json`

Reading the Value of a Variable

Required permission: VariableRead

To read the value of a shared variable, you must read the VariableValue. To read a VariableValue, use the following URI format:

GET `http://[host]:[port]/nivariable/VariableValues('[dynamic api url]')`

For example, use the following URI to read the value of `ni.var.psp://localhost/hvac/t01`:

GET `http://my_server:3580/nivariable/VariableValues('ni.var.psp://localhost/hvac/t01')?$format=json`



NOTE

In the JSON response, the timestamp is represented as milliseconds after 1970/1/1 0:0:0, UTC time.

To read the value property without the timestamp, use the following URI:

GET `http://my_server:3580/nivariable/VariableValues('ni.var.psp://localhost/hvac/t01')/Value?$format=json`

Writing to a Variable

Required permission: VariableWrite

To modify the variable value, issue a PUT request to the variable with a request body that uses the following data format:

```
{ "Value" : "[new value]" }
```

For example, to modify the value of `ni.var.psp://localhost/hvac/t01`, use the following URI:

```
PUT http://my_server:3580/nivariable/
VariableValues('ni.var.psp://localhost/hvac/t01')/Value
```



NOTE

The Value property of a variable must be a serialized string value, even if the variable is string type or another type. When writing to a string type variable, use the following format: `"\"test\""`

An empty response body with a header status code of 204 indicates that the modification was a success. If the PUT request data is invalid or the variable is not writable, a 500 Internal Error code or a 400 Bad Request code will be returned. In the case of an error, the response body contains the standard OData error object serialized to the format specified in the request.

Writing to Multiple Variables Simultaneously

Required permission: VariableWrite

Use the following format to update `variable1` and `variable2` of `process1`.

```
POST http://my_server:3580/nivariable/BatchWrite
```

POST JSON data

```
[ {
  "Url"
  : "ni.var.psp://localhost/process1/variable1", "Value"
  : "false"
}, {
  "Url"
  : "ni.var.psp://localhost/process1/variable2", "Value"
  : "\"this is a new string value\""
```

```
}}
```

POST XML data

```
<?xml version="1.0" encoding="utf-8"?>
<BatchWriteArguments>
  <BatchWriteArgument>
    <Url>ni.var.psp://localhost/process1/variable1</Url>
    <Value>>false</Value>
  </BatchWriteArgument>
  <BatchWriteArgument>
    <Url>ni.var.psp://localhost/process1/variable2</Url>
    <Value>this is a new string value</Value>
  </BatchWriteArgument>
</BatchWriteArguments>
```

If all variable writes succeed, the server returns the following response with a response body of an empty array:

HTTP/1.1 200 OK

Response body (JSON)

```
[]
```

Response body (XML)

```
<?xml version="1.0" encoding="utf-8"?>
<BatchWriteErrors>
</BatchWriteErrors>
```

If one or more variable writes fail, the response includes error descriptions. For example, if `variable2` is not writable, the server returns the following:

HTTP/1.1 200 OK

Response body (JSON)

```
[ {
  "Url"
  : "ni.var.psp://localhost/process1/variable2", "Error"
  : {
    "code"
    : "", "message"
```

```

: {
  "lang"
: "en-us", "value"
: "ni.var.psp://localhost/process1/variable2 is not
writable"
}
}]

```

Response body (XML)

```

<?xml version="1.0" encoding="utf-8"?>
<BatchWriteErrors>
<BatchWriteError>
<Url><ni.var.psp://localhost/process1/variable2</Url>
<error xmlns="http://schemas.microsoft.com/ado/2007/08/
dataservices/metadata">
<code>0</code>
<message xml:lang="en-US">"ni.var.psp://localhost/
process1/variable2 is not writable"</message>
</error>
</BatchWriteError>
</BatchWriteErrors>

```



NOTE

Each batch writes to all selected writable variables and returns errors for all failed write attempts. A single failed write attempt does not stop the entire batch write.

Creating a Variable Group

Required permission: VariableCreateDeleteGroup

A `VariableValueGroup` references a customized list of variable values. You can read the values of these variables using a single request. You can also use the `$delta` token feature to get only updated values for the variables.

To use a variable group, you must first create the group. For example, POST `http://my_server:3580/nivariable/VariableValueGroups`

POST JSON data:

```

HTTP Header - Content-Type : application/json
HTTP Body - {
  "VariableValues"
: [{ "__metadata" : {
    "uri"

```



```

      : "http://my_server:3580/nivariable/
VariableValues('ni.var.psp://localhost/process1/
variable1')"} }, { "__metadata" : {

      "uri"

      : "http://my_server:3580/nivariable/
VariableValues('ni.var.psp://localhost/process2/
variable2')"} } ]}

```

POST XML data:

```

HTTP Header - Content-Type : application/xmlHTTP Body -<?
xml version="1.0" encoding="utf-8"?><entry
xmlns:d="http://my_server:3580/nivariable/"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/
dataservices/metadata" xmlns="http://www.w3.org/2005/
Atom"> <title type="text"></title>
<updated>2010-02-27T21:36:47Z</updated> <author> <name />
</author> <category term="nivariable.VariableValueGroup"
scheme="http://schemas.microsoft.com/ado/2007/08/
dataservices/scheme" /> <link rel="http://
schemas.microsoft.com/ado/2007/08/dataservices/related/
VariableValues" href="VariableValues('ni.var.psp://
localhost/process1/variable1')"/> <link rel="http://
schemas.microsoft.com/ado/2007/08/dataservices/related/
VariableValues" href="VariableValues('ni.var.psp://
localhost/process1/variable2')"/> <content
type="application/xml"> <m:properties> <d:Timeout>15000</
d:Timeout> </m:properties> </content></entry>

```



NOTE

You do not need to include required properties when creating resources for VariableValueGroups. The ID property of a VariableValueGroup is normally a required property, and resources created in OData must include required properties. However, there is no way to determine which IDs are available to use. If your OData client requires a value for the Id property in order to send a request, you can assign a random value. The variable Web service replaces the user-supplied Id property value with a unique value.

If the VariableValueGroup is created, the server returns a 201 response and adds a Location header indicating the newly created resource's URI. The response body contains the ID of the newly created group.

If the VariableValueGroup is not created, the server returns one of the following responses:

- 500 Internal Server Error - The error occurred on the server side.
- 400 Bad Request - The requested POST data is invalid.

The response body uses the standard OData error object to indicate the reason of failure.

Deleting a Variable Group

Required permission: VariableCreateDeleteGroup

To delete a group, send a DELETE request to the ID of the variable group. For example,
 DELETE `http://my_server:3580/nivariable/VariableValueGroups('1587dcd4a87d118ae3ddd0fce09246f6')`

If the DELETE request succeeds, a 204 status code is returned with an empty response body. If the DELETE request fails, the OData error object detailing the error is returned in the requested format (XML or JSON).

Reading a Variable Group

Required permission: VariableReadGroup

After you create the VariableValueGroup, you can use the group ID to read the group like any other resource. For example, GET `http://my_server:3580/nivariable/VariableValueGroups('1587dcd4a87d118ae3ddd0fce09246f6')?$format=json`

To read the VariableValues contained in the group, you can request the deferred VariableValues URI:

GET `http://my_server:3580/nivariable/VariableValueGroups('1587dcd4a87d118ae3ddd0fce09246f6')/VariableValues?$format=json`

You can use the \$expand option to include the VariableValues in the group response. For example, GET `http://my_server:3580/nivariable/VariableValueGroups('1587dcd4a87d118ae3ddd0fce09246f6')?$expand=VariableValues&$format=json`

Getting Variable Updates

To receive updates, HTTP clients must poll manually. HTTP does not provide a push method for servers to notify HTTP clients of events or changes. To reduce unnecessary traffic when polling, OData provides a \$deltatoken parameter that returns only items that are changed or added.

In the case of VariableValueGroups, the \$deltatoken provides the updated variable values while skipping variable values that have not changed since the last request.

This process works in the following way:

Initially, the HTTP client reads the `VariableValues` property of a group.

```
GET http://my_server:3580/nivariable/
VariableValueGroups('1587dcd4a87d118ae3ddd0fce09246f6')/
VariableValues?$format=json
```

The HTTP client receives the initial data set. To request the delta change, the HTTP client uses the following URI, as given by the `_delta` property, with the timestamp from the `VariableValue` reading as the `$deltatoken`.

```
GET http://my_server:3580/nivariable/
VariableValueGroups('1587dcd4a87d118ae3ddd0fce09246f6')/
VariableValues?$format=json&$deltatoken=1305692428170
```

If there are no changes since the previous GET, the results are empty. To get the changes again, follow the delta URI as specified in the last response.

```
GET http://my_server:3580/nivariable/
VariableValueGroups('1587dcd4a87d118ae3ddd0fce09246f6')/
VariableValues?$format=json&$deltatoken=1305692428170
```



NOTE

Requesting a delta value that has not changed does not cause an error for the HTTP client.

Polling Variables Bound to External I/O or Other Shared Variables in Set Periods

If a shared variable is bound to an external I/O or another shared variable, a value update can come with a timestamp that is older than the `deltatoken`. For example, you request a delta change at time t , and a new value change arrives at $t + 1$, but with a timestamp of $t - 1$. The next time you get another delta change, the server no longer treats the $t - 1$ timestamp as a new value. To avoid this issue, poll variable values in set periods instead of using the `$delta` link.

Editing Multiple Shared Variables

Use the Multiple Variable Editor window to configure a large number of shared variables at one time. For example, you can change the data type of several shared variables at once. Complete the following steps to change several double-precision variables to single-precision variables.

1. Right-click a project library or shared variable and select **Multiple Variable Editor** from the shortcut menu to display the **Multiple Variable Editor** window.
2. Right-click the **Data Type** column and select **Sort Ascending** from the shortcut menu.

3. Select all of the cells with a **Data Type** of **Double**.
4. Click on **Double** and select **Single** from the menu.
5. Click the **Done** button to close the **Multiple Variable Editor** window.
6. In the **Project Explorer** window, save the library in which the shared variables reside to save the configuration.

Creating Bound Variables

Complete the following steps to create multiple bound variables simultaneously.

1. Right-click a project library and select **Create Bound Variables** from the shortcut menu to display the Create Bound Variables dialog box.
2. Use the **Browse Source** pull-down menu to determine the data source of the shared variable.
3. Navigate to an *NI Publish-Subscribe Protocol (NI-PSP)* data item in the **Project Items** tree or the **Network Items** tree.
4. Select the shared variables you want to add to the current project library and click the **Add** button. LabVIEW creates the new bound shared variables based on the shared variables you select.
5. Click the **OK** button to close the **Create Bound Variables** dialog box and display the Multiple Variable Editor window.
6. (Optional) Edit the shared variables in the **Multiple Variable Editor** window.
7. Click the **Done** button. The bound shared variables now appear in the **Project Explorer** window under the project library you selected.
8. Select **File»Save** to save the changes to the project library.

Importing Shared Variable Configuration from a Spreadsheet

Shared variables contain configuration information, such as the data type and variable type. Complete the following steps to import shared variable configuration information from a spreadsheet file.



NOTE

If you enable the initial value of a string shared variable, ensure that you specify the initial value in the spreadsheet file. LabVIEW returns an error if you import a string shared variable whose initial value is enabled but not set or empty.

1. *Export* shared variable configuration to a .csv file.
2. In a spreadsheet application, open the .csv file and modify the properties.
3. In LabVIEW, right-click the project library to which you want to import the configuration and select **Import Variables** from the shortcut menu.
4. Select the .csv file from the **CSV to import** dialog box and click the **OK** button.



Note (DSC Module) If the description of a shared variable contains line breaks and you export the configuration information for that shared variable to a spreadsheet, the line breaks are converted to spaces. When you import that shared variable configuration information back to a project library, the resulting description contains spaces instead of line breaks.

Exporting Shared Variable Configuration to a Spreadsheet

Shared variables contain configuration information, such as the data type and variable type. Complete the following steps to export shared variable configuration information to a spreadsheet file.

1. Right-click a project library and select **Export Variables** from the shortcut menu.
2. Enter the name of the .csv spreadsheet file you want to use to export the shared variable configuration.
3. In a spreadsheet application, open the .csv file you created.



NOTE

(DSC Module) If the description of a shared variable contains line breaks and you export the configuration information for that shared variable to a spreadsheet, the line breaks are converted to spaces. When you import that shared variable configuration information back to a project library, the resulting description contains spaces instead of line breaks.

Numeric Data

Numeric Conversion

If you wire an output to a destination that has a different numeric representation, LabVIEW converts the data according to the following rules:

- **Signed or unsigned integer to floating-point number** —LabVIEW converts the data to the nearest value possible. This conversion is exact when the floating-point number is more precise than the integer. If the integer is more precise than the floating-point number, LabVIEW maintains as much precision as possible in the conversion.
- **Floating-point number to signed or unsigned integer** —LabVIEW coerces out-of-range values to the minimum or maximum value of the integer. For example, if you convert a negative floating-point number to an unsigned integer, the result is 0.
- **LabVIEW treats enums as unsigned integers** —For example, if you convert -1, a floating-point number, to an [enum](#), LabVIEW coerces the value into the range of the enum. If the range of the enum is 0 to 25, LabVIEW converts the value to 0.
- **Integer to integer** —LabVIEW does not coerce out-of-range values to the minimum or maximum value of an integer. If the source is smaller than the destination,

LabVIEW extends the sign of a signed source and places zeros in the extra bits of an unsigned source. If the source is larger than the destination, LabVIEW copies only the least significant bits of the value.

- **Integer, floating-point, or fixed-point number to fixed-point number** —LabVIEW coerces values outside the desired range for the fixed-point number to within the fixed-point number's maximum and minimum values.

If you wire two different numeric data types to a numeric function that expects the inputs to be the same data type, LabVIEW converts one of the terminals to the same representation as the other terminal. LabVIEW chooses the representation that uses more bits. If the number of bits is the same, LabVIEW chooses unsigned over signed.



NOTE

(FPGA Module) Coercions that occur in FPGA VIs can consume significant FPGA resources, especially when a terminal is coerced to or from the SGL data type.

LabVIEW places a **coercion dot** on a block diagram node to alert you that LabVIEW coerced the value passed into the node to a different representation. In **certain coercions**, coercion dots may lead to an increase in memory usage and execution time and a decrease in accuracy of results. To eliminate coercion dots, match the output data type of the control or constant to the data type of the receiving terminal. You can right-click the control or constant on the block diagram and select **Representation** from the shortcut menu to change the representation of the input value to the representation of the terminal.

Changing Numeric Representation

Change the representation of numeric objects to conserve memory by limiting the number of times LabVIEW coerces data.

By default, the representation of a numeric constant automatically adapts to the value of the constant you enter. For example, the default representation of a numeric constant with a value of 1 is a 32-bit integer. If you change the value of the constant to 1.1, the representation of the constant changes to a double-precision, floating-point number.

Complete the following steps to change the representation of a numeric object.

1. Right-click the numeric object.
2. Select **Representation** from the shortcut menu.
3. Select the **data type** that best represents the data.



NOTE

Slide and rotary controls and indicators cannot represent complex numbers.

If you change the representation of an object by using the shortcut menu, the object retains the representation you specified regardless of the value you enter.

Some functions, such as Divide, Sine, and Cosine, always produce floating-point output. If you wire integers to the inputs of these functions, the functions convert the integers to double-precision, floating-point numbers before they perform the calculation.

To reset a constant to determine its type based on its value, right-click the constant and select **Adapt To Entered Data** from the shortcut menu.

Changing the Numeric Representation of a Control Refnum

Complete the following steps to change the numeric representation of a control refnum.

1. Right-click the control refnum front panel control and select **Show Control** from the shortcut menu to display the refnum control as its referenced object.
2. Right-click the control that appears in the front panel control refnum and select **Representation** from the shortcut menu.
3. Select the *data type* that you want to use.

To revert back to the control refnum icon, right-click the control refnum control and select **Show Icon** from the shortcut menu.

Configuring Output Settings for Numeric Functions

The Numeric functions are *polymorphic functions*. When you wire two or more numeric inputs of different representations to a function, the function usually returns the data in the larger, or wider, representation. The functions coerce the smaller representations to the widest representation before execution and LabVIEW places a coercion dot on the terminal where the conversion takes place.

However, you can configure a Numeric function to return a data type you specify. If you select *fixed-point representation* as the output and remove the checkmark from the **Adapt to source** checkbox on the Data Type page, you manually can configure the encoding of the output and how the function handles overflow and/or rounding conditions. After you remove the checkmark from this checkbox, a *blue coercion dot* appears on the output terminal of the function. In this situation, the width and range of the output type determines whether data loss occurs.

Configuring Output Representation

Complete the following steps to configure the representation for the output of a Numeric function.

1. Right-click the function and select **Properties** from the shortcut menu to display the **Object Properties** dialog box.

2. On the Output Configuration page, click the data type icon in the **Representation** section and select the data type you want to use for the output from the shortcut menu. LabVIEW removes the checkmark from the **Adapt to source** checkbox if the data type you select does not match the data type of the input values.
3. If you set **Representation** to **FXPFixed-point**, continue to the following section. Otherwise, click the **OK** button to close the dialog box and apply the configuration settings.

Configuring Settings for a Fixed-Point Output

Complete the following steps to configure settings for a fixed-point output.

1. Right-click the Numeric function and select **Properties** from the shortcut menu to display the **Object Properties** dialog box.
2. On the **Output Configuration** page, click the data type icon in the **Representation** section and select the **FXPFixed-point** icon from the shortcut menu. LabVIEW removes the checkmark from the **Adapt to source** checkbox because the fixed-point data type does not match the data type of the input values.
3. Select the **Rounding mode** to specify how the function handles *rounding conditions*.
4. Select the **Overflow mode** to specify how the function handles *overflow conditions*.
5. (Optional) Complete the following steps to configure the **Encoding** of the output.
 1. Select **Signed** or **Unsigned** to specify whether you want to represent a signed or unsigned number.
 2. In the **Word length** field, specify the total number of bits you want to use to represent the value of the output.
 3. In the **Integer word length** field, specify the number of integer bits you want to use to represent the value of the output.



NOTE

Specifying **Encoding** has a lower impact on performance than specifying **Range**. The performance impact of specifying **Range** is minimal. However, the performance impact can be significant if you run a VI that coerces values to a desired range on a target such as FPGA.

1. (Optional) Complete the following steps to configure the **Range** of the output. If you select **Wrap** as the **Overflow mode**, you cannot configure the **Range** of the output because LabVIEW automatically sets the range to fit the encoding.
 1. In the **Minimum** field, enter the minimum value to which you want the output to conform.
 2. In the **Maximum** field, enter the maximum value to which you want the output to conform.
 3. In the **Delta** field, enter the increment between numbers within the **Range**.



Note You cannot configure both **Encoding** and **Range** because the settings depend on each other. The settings you configure last are the settings that LabVIEW uses for the fixed-point number.

1. (Optional) Place a checkmark in the **Include overflow status** checkbox to include an **overflow status** in the fixed-point output. If one of the inputs of the Numeric function includes an overflow status, the fixed-point output also includes an overflow status, regardless of whether you place a checkmark in this checkbox.
2. Click the **OK** button to close the dialog box and apply the configuration settings.

Numeric Data Types

LabVIEW represents numeric data types as floating-point numbers, fixed-point numbers, integers, unsigned integers, and complex numbers. The difference among the numeric data types is the number of bits they use to store data and the data values they represent.

(Windows) LabVIEW can process data only in the ranges shown in the Numeric Data Types Table, but it can display in text format a range of $\pm 9.999999999999999\text{E}999$.



RELATED INFORMATION

Floating-Point Numbers on page 1157

Fixed-Point Numbers on page 1160

Integers on page 1157

Unsigned Integers on page 1159

Complex Numbers on page 1156

Fixed-Point Numbers on page 1160

Numeric Data Types Table on page 1158

Complex Numbers

Complex numbers are concatenated versions of floating-point numbers with a real and an imaginary part. Refer to the Numeric Data Types Table for more information about numeric data type bits, digits, and range. There are three types of complex numbers.



Complex single-precision, floating-point (CSG) —Same as single-precision, floating-point for each (real and imaginary) part.



Complex double-precision, floating-point (CDB)—Same as double-precision, floating-point for each (real and imaginary) part.



Complex extended-precision, floating-point (CXT)—Same as extended-precision, floating-point for each (real and imaginary) part.

Use the Complex functions to create complex numbers.



RELATED INFORMATION

[Floating-Point Numbers](#) on page 1157

[Numeric Data Types Table](#) on page 1158

[Floating-Point Numbers](#) on page 1157

[Floating-Point Numbers](#) on page 1157

[Floating-Point Numbers](#) on page 1157

Floating-Point Numbers

Floating-point numbers in LabVIEW conform to the ANSI/IEEE Standard 754-1985. Not all real numbers can be represented in the ANSI/IEEE standard floating-point numbers. Because of this, comparisons using floating-point numbers may yield results you do not expect because of rounding errors. To avoid inaccurate results, you can round floating-point numbers to integers. For example, if you want the result of a calculation to contain two digits of precision, multiply the floating-point number by 100 and then round the product to an integer before you complete the calculation. You also can check to see whether two floating-point numbers are close to each other instead of equal to each other. For example, if the absolute value of the difference of two floating-point numbers is smaller than a defined tolerance, assume the numbers are equal.

Refer to the [Numeric Data Types Table](#) for more information about numeric data type bits, digits, and range. There are three types of floating-point numbers.

	Single-precision (SGL) — Single-precision, floating-point numbers have 32-bit IEEE single-precision format. Use single-precision, floating-point numbers when memory savings are important and you will not overflow the range of the numbers.
	Double-precision (DBL) — Double-precision, floating-point numbers have 64-bit IEEE double-precision format. Double-precision is the default format for numeric objects. For most situations, use double-precision, floating-point numbers.
	Extended-precision (EXT) — When you save extended-precision numbers to disk, LabVIEW <i>stores</i> them in a platform-independent 128-bit format. In memory, the size and precision vary depending on the platform. Use extended-precision, floating-point numbers only when necessary. The performance of extended-precision arithmetic vary among platforms.

Data Acquisition VIs often return arrays of floating-point numbers.

Integers

Integers represent whole numbers and can be positive or negative. Refer to the [Numeric Data Types Table](#) for more information about numeric data type bits, digits, and range. There are four types of integers.

	Byte (I8) —Byte integer numbers have 8 bits of storage.
	Word (I16) —Word integer numbers have 16 bits of storage.
	Long (I32) —Long integer numbers have 32 bits of storage. In most cases, it is best to use a 32-bit integer.
	Quad (I64) —Quad integer numbers have 64 bits of storage.

When LabVIEW converts floating-point numbers to integers, the VI rounds to the nearest even integer. For example, LabVIEW rounds 2.5 to 2 and rounds 3.5 to 4. Refer to [Numeric Conversion](#) for more information about how LabVIEW converts numeric representations.

Numeric Data Types Table

The following table displays the [numeric data types](#) available in LabVIEW. LabVIEW [stores](#) each data type in a different way.

Terminal	Numeric Data Type	Bits of Storage on Disk	Approximate Number of Decimal Digits	Approximate Range
	Single-precision, floating-point	32	6	Minimum positive number: 1.40e-45 Maximum positive number: 3.40e+38 Minimum negative number: -1.40e-45 Maximum negative number: -3.40e+38
	Double-precision, floating-point	64	15	Minimum positive number: 4.94e-324 Maximum positive number: 1.79e+308 Minimum negative number: -4.94e-324 Maximum negative number: -1.79e+308
	Extended-precision, floating-point	128	varies from 15 to 20 by platform	Minimum positive number: 6.48e-4966 Maximum positive number: 1.19e+4932 Minimum negative number: -6.48e-4966 Maximum negative number: -1.19e+4932
	Complex single-precision, floating-point	64	6	Same as single-precision, floating-point for each (real and imaginary) part

Terminal	Numeric Data Type	Bits of Storage on Disk	Approximate Number of Decimal Digits	Approximate Range
	Complex double-precision, floating-point	128	15	Same as double-precision, floating-point for each (real and imaginary) part
	Complex extended-precision, floating-point	256	varies from 15 to 20 by platform	Same as extended-precision, floating-point for each (real and imaginary) part
	Fixed-point	64, or 72 if you <i>include an overflow status</i>	varies by user configuration	varies by user configuration
	Byte signed integer	8	2	–128 to 127
	Word signed integer	16	4	–32,768 to 32,767
	Long signed integer	32	9	–2,147,483,648 to 2,147,483,647
	Quad signed integer	64	18	–1e19 to 1e19
	Byte unsigned integer	8	2	0 to 255
	Word unsigned integer	16	4	0 to 65,535
	Long unsigned integer	32	9	0 to 4,294,967,295
	Quad unsigned integer	64	19	0 to 2e19
	128-bit time stamp	128	19	Minimum time: 01/01/1600 00:00:00 UTC maximum time: 01/01/3001 00:00:00 UTC

Unsigned Integers

Unsigned integers represent only non-negative integers and have a larger range of positive numbers than signed integers because the number of bits is the same for both representations. Refer to the [Numeric Data Types Table](#) for more information about numeric data type bits, digits, and range. There are four types of unsigned numbers.

	Byte (U8) —Byte unsigned integer numbers have 8 bits of storage.

	Word (U16) —Word unsigned integer numbers have 16 bits of storage.
	Long (U32) —Long unsigned integer numbers have 32 bits of storage.
	Quad (U64) —Quad unsigned integer numbers have 64 bits of storage.

Fixed-Point Data

Fixed-Point Numbers

Fixed-point numbers represent rational numbers within a user-specified range and with a user-specified precision. You can specify any size between 1 and 64 bits, inclusive, for a fixed-point number. You can configure fixed-point numbers as signed or unsigned.

Refer to the Numeric Data Types Table for more information about numeric data type bits, digits, and range.

	Fixed-point —Fixed-point numbers have up to 64 bits of storage.



RELATED INFORMATION

- [Numeric Data](#) on page 133
- [Configuring Fixed-Point Numbers](#) on page 1160
- [Numeric Data Types Table](#) on page 1158

Configuring Fixed-Point Numbers

You can configure the [encoding](#) for a [fixed-point number](#) in a numeric control, constant, or indicator.

After you configure a fixed-point control, constant, or indicator, that object cannot display a number that does not conform to the settings you specify. If the object you configure is an indicator, LabVIEW coerces any input value to the indicator to conform to the fixed-point configuration settings of the indicator.

Configuring Controls

Complete the following steps to configure a fixed-point number in a numeric control.

1. Right-click the control and select **Properties** from the shortcut menu to display the **Numeric Properties** dialog box.

2. On the Data Type page, click the data type icon in the **Representation** section and select **FXPFixed-point** from the shortcut menu. The **Fixed-Point Configuration** section displays default values for the **Encoding** and **Range** options.
3. (Optional) Complete the following steps to configure the **Encoding** of the fixed-point number.
 1. Select **Signed** or **Unsigned** to specify whether you want to represent a signed or unsigned number.
 2. In the **Word length** field, specify the total number of bits you want to use to represent the value of the fixed-point number.
 3. In the **Integer word length** field, specify the number of integer bits you want to use to represent the value of the fixed-point number.

**NOTE**

LabVIEW calculates the **Range** of the fixed-point number according to the values you specify for the **Encoding** of the fixed-point number.

1. (Optional) Place a checkmark in the **Include overflow status** checkbox to include an **overflow status** in the fixed-point number.
2. Click the **OK** button to close the dialog box and apply the configuration settings.

Configuring Constants

Complete the following steps to configure a fixed-point number in a numeric constant.

1. Right-click the constant and select **Properties** from the shortcut menu to display the **Numeric Constant Properties** dialog box.
2. On the Data Type page, remove the checkmark from the **Adapt to entered data** checkbox.



Note When you select **Adapt to entered data**, LabVIEW displays any value you enter with the shortest possible word length and integer word length. You must remove the checkmark from this checkbox if you want LabVIEW to retain any changes you make to the **Fixed-Point Configuration** settings.

1. Click the data type icon in the **Representation** section and select **FXPFixed-point** from the shortcut menu. The **Fixed-Point Configuration** section displays default values for the **Range** and **Encoding** options.
2. (Optional) Complete the following steps to configure the **Encoding** of the fixed-point number.
 1. Select **Signed** or **Unsigned** to specify whether you want to represent a signed or unsigned number.
 2. In the **Word length** field, specify the total number of bits you want to use to represent the value of the fixed-point number.

3. In the **Integer word length** field, specify the number of integer bits you want to use to represent the value of the fixed-point number.



Note LabVIEW calculates the **Range** of the fixed-point number according to the values you specify for the **Encoding** of the fixed-point number.

1. (Optional) Place a checkmark in the **Include overflow status** checkbox to include an **overflow status** in the fixed-point number.
2. Click the **OK** button to close the dialog box and apply the configuration settings.

Configuring Indicators

Complete the following steps to configure a fixed-point number in a numeric indicator.

1. Right-click the indicator and select **Properties** from the shortcut menu to display the **Numeric Properties** dialog box.
2. On the Data Type page, click the data type icon in the **Representation** section and select **FXPFixed-point** from the shortcut menu. The **Fixed-Point Configuration** section displays default values for the **Range** and **Encoding** options.
3. (Optional) Place a checkmark in the **Adapt to source** checkbox if you want the value to inherit the fixed-point configuration settings of an input fixed-point value. If you select this option, skip to step 7.
4. (Optional) Complete the following steps to configure the **Encoding** of the fixed-point number.
 1. Select **Signed** or **Unsigned** to specify whether you want to represent a signed or unsigned number.
 2. In the **Word length** field, specify the total number of bits you want to use to represent the value of the fixed-point number.
 3. In the **Integer word length** field, specify the number of integer bits you want to use to represent the value of the fixed-point number.



Note LabVIEW calculates the **Range** of the fixed-point number according to the values you specify for the **Encoding** of the fixed-point number.

1. (Optional) Place a checkmark in the **Include overflow status** checkbox to include an **overflow status** in the fixed-point number.
2. Click the **OK** button to close the dialog box and apply the configuration settings.

Caveats and Recommendations for Using Fixed-Point Numbers

The following list describes some of the caveats and recommendations to consider when you use fixed-point numbers:

- Adding a shift register to a loop that contains Numeric functions that increase the size of fixed-point data can create cycles in the propagation of type information. Overflow can occur as the growing data approaches the maximum word length that LabVIEW accepts or the maximum word length that you specify. If you do not configure how the Numeric functions handle overflow, LabVIEW converts the data to floating-point representation to avoid potential data loss and broken VIs.
- The word length you specify might exceed the maximum word length a target accepts. If this occurs, LabVIEW sets the word length to the maximum possible length and uses the default settings for overflow and rounding. By default, the word length retains the most significant bits. This word length reduction rounds the value but prevents overflow. Refer to the documentation for a target to determine the maximum word length the target accepts.
- LabVIEW and other applications encode fixed-point numbers in binary format, which is a base-2 number system. Because the decimal number system is base-10, you cannot represent all decimal values exactly in fixed-point. For example, you can represent 0.1 exactly in decimal format, but if you convert 0.1 to a fixed-point number with a 32-bit word length and a 16-bit integer word length, LabVIEW changes the rational number to 0.100006, which it can represent in base-2.



RELATED INFORMATION

[Numeric Data](#) on page 133

[Transferring Values between Loop Iterations](#) on page 415

[Adapting to Cycles of Fixed-Point Data in Loops with Shift Registers](#) on page 1163

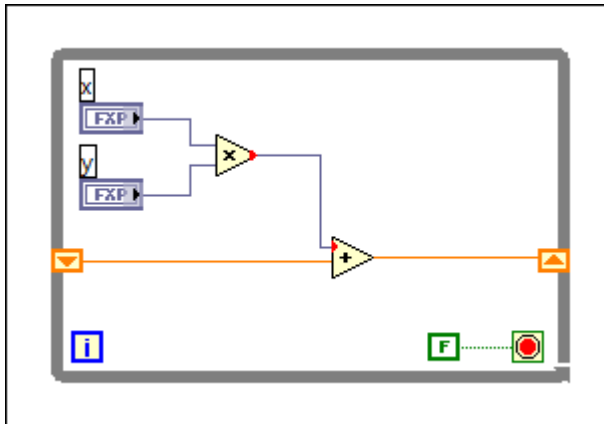
[Configuring Output Settings for Numeric Functions](#) on page 1154

[Numeric Data](#) on page 133

Adapting to Cycles of Fixed-Point Data in Loops with Shift Registers

If you perform an operation within a loop that increases the size of [fixed-point data](#), such as multiplication or incrementing, and pass the result of the operation into a shift register, LabVIEW converts the output data to floating-point representation to avoid potential data loss from [overflow of the fixed-point data](#). Overflow can occur because the word length of the fixed-point value can grow exponentially with each iteration of the loop, depending on the operation you perform within the loop.

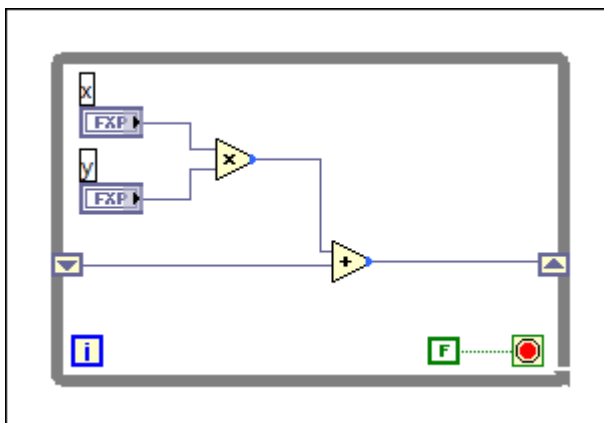
The following block diagram displays an orange wire where LabVIEW automatically converted fixed-point data to floating-point. The coercion dots indicate that LabVIEW is discarding bits of the output values of the Numeric functions.



If you do not want LabVIEW to convert the data to floating-point automatically, you must [configure the output settings](#) for the function that processes data from the shift register to specify the output data type and how the function handles overflow, should it occur.

For example, in the previous block diagram, you can configure the Add function to output only fixed-point values. After you configure the output data type to be fixed-point, the orange wire on the block diagram changes color to indicate that it carries fixed-point data.

The following image displays the block diagram appearance after you configure the Add function to output fixed-point values. The blue coercion dots on the Numeric functions indicate that you have configured output settings for those functions.



Displaying and Hiding the Overflow Status LED for Fixed-Point Numbers

After you [configure a fixed-point control, constant, or indicator](#) to include an overflow status, you can display and hide an overflow status LED on the control, constant, or indicator. This LED indicates whether the fixed-point number is the result of an operation that [overflowed](#). Complete the following steps to display or hide an overflow status LED.

1. [Configure the fixed-point control, constant or indicator](#) to include an overflow status.
2. Right-click the fixed-point control or indicator from the front panel, or right-click the fixed-point constant from the block diagram.

3. Select **Visible Items»Overflow Status LED** from the shortcut menu.



NOTE

If you do not configure the control, constant, or indicator to include an overflow status, the **Overflow Status LED** shortcut menu item does not appear.

VIs and Functions that Support the Fixed-Point Data Type

Not all VIs and functions in LabVIEW support the fixed-point data type. If you wire a fixed-point number to a VI or function that does not support the fixed-point data type, the VI appears with a broken **Run** button. The following VIs and functions support the fixed-point data type.

Array VIs and Functions

The following Array VIs support the fixed-point data type:

- Decrement Array Element
- Increment Array Element

Assert Type VIs and Functions

The following Assert Type VIs and functions support the fixed-point data type:

- Assert Fixed-Point Numeric Type
- Assert Fractional Numeric Type
- Assert Real Numeric or Waveform Type
- Assert Real Numeric Type
- Assert Scalar Numeric or Waveform Type
- Assert Scalar Numeric Type
- Assert Structural Type Match
- Assert Structural Type Mismatch

Comparison VIs and Functions

The following Comparison VIs and functions support the fixed-point data type:

- Equal?
- Equal To 0?
- Greater Or Equal?
- Greater Or Equal To 0?

- Greater?
- Greater Than 0?
- In Range and Coerce
- Is Value Changed
- Less Or Equal?
- Less Or Equal To 0?
- Less?
- Less Than 0?
- Max & Min
- Not Equal?
- Not Equal To 0?

Conversion Functions

The following Conversion functions support the fixed-point data type:

- Boolean Array To Number
- Coerce To Type
- Number To Boolean Array
- To Byte Integer
- To Double Precision Float
- To Extended Precision Float
- To Fixed-Point
- To Long Integer
- To Quad Integer
- To Single Precision Float
- To Unsigned Byte Integer
- To Unsigned Long Integer
- To Unsigned Quad Integer
- To Unsigned Word Integer
- To Word Integer

Data Manipulation Functions

The following Data Manipulation functions support the fixed-point data type:

- Flatten To String
- Logical Shift
- Rotate Left With Carry

- Rotate Right With Carry
- Type Cast
- Unflatten From String

LabVIEW Schema Functions

The following LabVIEW Schema functions support the fixed-point data type:

- Flatten to XML
- Unflatten from XML

Numeric Functions

The following Numeric functions support the fixed-point data type:

- Absolute Value
- Add
- Decrement
- Divide
- Increment
- Multiply
- Negate
- Reciprocal
- Round To Nearest
- Round Toward +Infinity
- Round Toward -Infinity
- Scale by Power of 2
- Sign
- Square
- Square Root
- Subtract

String/Number Conversion Functions

The following String/Number Conversion functions support the fixed-point data type:

- Decimal String To Number
- Fract/Exp String To Number
- Hexadecimal String To Number
- Number To Decimal String

- Number To Engineering String
- Number To Exponential String
- Number To Fractional String
- Number To Hexadecimal String
- Number To Octal String
- Octal String To Number

Timing VIs and Functions

The following Timing VI supports the fixed-point data type:

- Stall Data Flow



RELATED INFORMATION

[Fixed-Point Numbers](#) on page 1160

Variant Data

Variant Data

Sometimes you may need a VI to handle data of many different types in a generic way. You could write a different VI for each specific data type, but having copies of your VI is hard to maintain, since if you change one you have to remember to change all of them. LabVIEW provides the variant data type as one solution for these situations. The variant data type is a generic container for all other types of data in LabVIEW. When you convert other data to a variant, the variant stores the data and the original data type of the data, which allows LabVIEW to correctly convert the variant data back to the original data at a later time. For example, if you convert string data to a variant, the variant stores the text of the string and an indicator that says that this data was originally a string (as opposed to a path or an array of bytes, or other possible LabVIEW types).

Use the Variant functions to create and manipulate variant data. You can convert flattened data and convert LabVIEW data to variant data. You can convert variant data to LabVIEW data. Use the Data Type Parsing VIs to retrieve and compare the data types stored in a variant to other data types.

The variant data type allows you to store attributes of the data. An attribute is information about the data that you define and the variant data type stores. For example, if you want to know the time when a piece of data was created, you can store the data as variant data and

add an attribute called **Time** to store the time string. The attribute data can be of any type. You also can delete or retrieve attributes from variant data.



RELATED INFORMATION

[Converting Flattened Data to Variant Data](#) on page 1171

[Converting LabVIEW Data to Variant Data](#) on page 1171

[Converting Variant Data to LabVIEW Data](#) on page 1172

[Adding Attributes to Variant Data](#) on page 1169

[Deleting Attributes from Variant Data](#) on page 1173

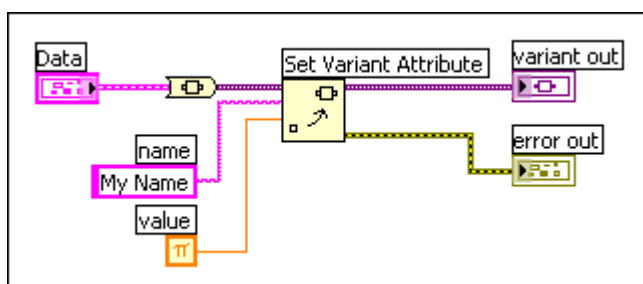
[Retrieving Attributes from Variant Data](#) on page 1173

[Handling Variant Data](#) on page 199

Adding Attributes to Variant Data

Complete the following steps to add attributes to **variant data**.

1. Add the Set Variant Attribute function to the block diagram.
2. Wire the variant data to the **variant** input of the Set Variant Attribute function.
3. Wire the **name** and **value** inputs of the Set Variant Attribute function.
4. Wire the **variant out** output of the Set Variant Attribute function to any VI or function that accepts variant data as an input.
5. Run the VI. The block diagram should appear similar to the following block diagram.



Checking Variants for Known Data Types

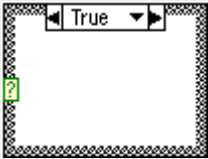
To check if a variant contains a specific data type, use the Data Type Parsing VIs to retrieve data type information. Unlike the Variant to Data function, the Data Type Parsing VIs do not retrieve the actual data stored in the variant. Instead, these VIs return information about the data type, so you do not have to manipulate the stored data to return information about it.

If you know the data type you want to check against the data type stored in a variant, you can use the Data Type Parsing VIs to compare these data types. For example, if an

application parses arbitrary front panels to retrieve the values of the controls, you can use the example in this topic to identify the variant controls that contain error clusters.

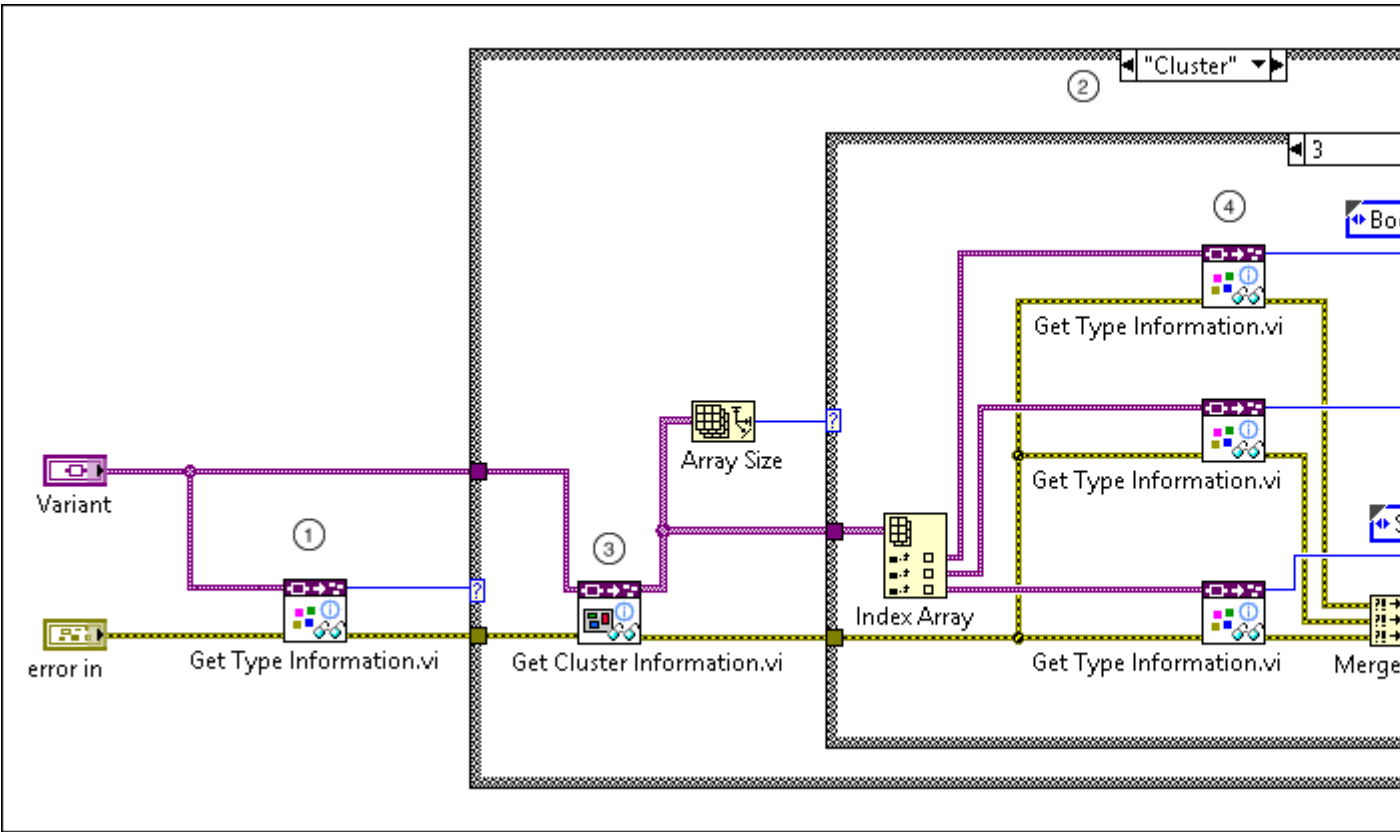
What to Use

Use the following block diagram objects to retrieve data type information stored in a variant.

Get Type Information VI	Case Structure	Get Cluster Information VI
		

What to Do

Create the following block diagram to check a variant for a known data type. This example checks and compares the data types stored in a variant to error cluster elements.



The following list describes important details about the previous diagram.

①	Use the Get Type Information VI to return an enum that specifies the data type stored in the variant. In the example, the data type is a cluster.
②	Edit the selector label of the Case structure to correspond to the known data type. In the example, the label and data type are both "Cluster."
③	Use the Get Cluster Information VI to return an array containing the cluster element data types. Once you know the data type contained in a variant, you can use data type specific VIs from the Data Type Parsing palette, such as the Get Numeric Information or Get Fixed-Point Information VIs, to return more specific data type information.
④	Edit the selector label of the inner Case structure to correspond with the known number of elements in the error cluster data type.
⑤	Use the Get Type Information VI again to return an enum for each of the three elements in the error cluster. You can then compare the data types of the individual elements and the standard order of these elements to the known data types and order of a LabVIEW error cluster.

Converting Flattened Data to Variant Data

Complete the following steps to convert data you flattened with the Flatten To String function to variant data.

1. Add the Flattened String To Variant function to the block diagram.
2. Wire the array of numeric integers that represents the data type of the flattened data to the **type string** input of the Flattened String To Variant function.
3. Wire a flattened data string to the **data string** input of the Flattened String To Variant function.
4. Wire the **variant** output of the Flattened String To Variant function to any VI or function that accepts variant data as an input.
5. Run the VI.

Converting LabVIEW Data to Variant Data

The variant data type is a generic container for all other types of data in LabVIEW. When you convert other data to a variant, the variant stores the data and the original data type. Use variant data when it is important to manipulate data independently of data type.

Complete the following steps to convert LabVIEW data to variant data.

1. Add the To Variant function to the block diagram.
2. Wire data to the **anything** input.
3. Wire the **variant** output of the To Variant function to any VI or function that accepts variant data as an input.

4. Run the VI.

After you convert LabVIEW data to variant data, you can *add attributes* to the data.

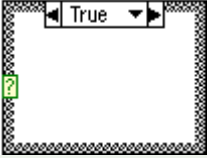
You also can *convert variant data to LabVIEW data*.

Converting Variant Data to LabVIEW Data

Use the Variant to Data function to convert *variant data* to LabVIEW data. You must determine the data type stored in a variant before you convert the variant.

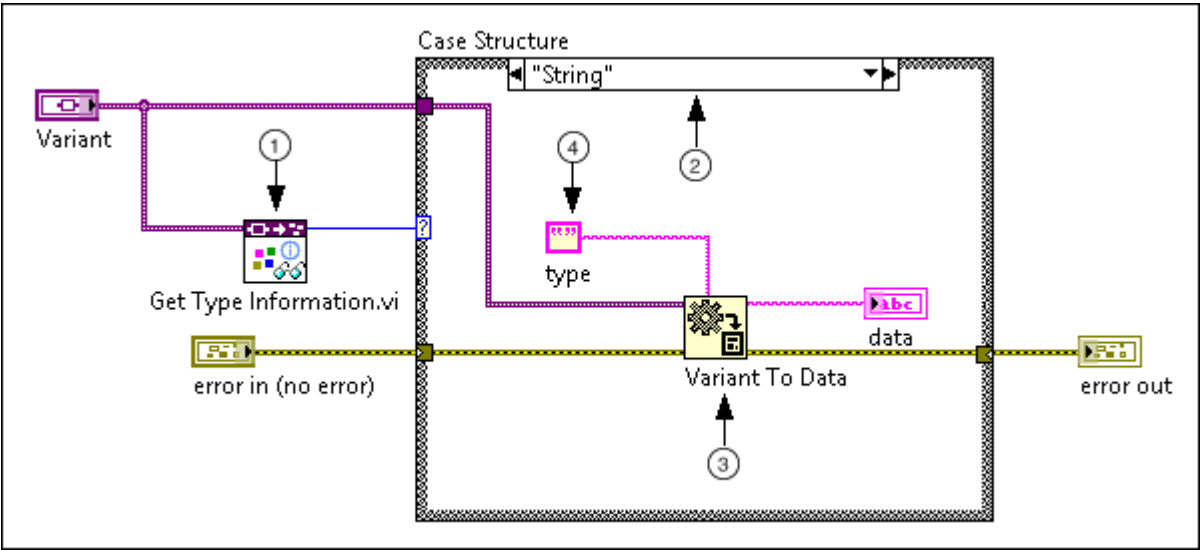
What to Use

Use the following block diagram objects to determine the data type stored in a variant and convert the variant to the data type.

Get Type Information VI	Case Structure	Variant To Data Function
		

What to Do

Create the following block diagram to determine the data type stored in a variant and convert the variant to the data type.



The following list describes important details about the previous diagram.

①	The Get Type Information VI retrieves the data type stored in the variant.
②	<i>Add a subdiagram</i> for each possible data type. A subdiagram executes only when the variant data type matches the data type that appears in the selector label.
③	In each subdiagram, the Variant to Data function converts the variant.
④	In each subdiagram, the data type wired to the type input of the Variant to Data function must match the data type in the selector label. LabVIEW returns an error if the data type of the type input is not compatible with the data type of the variant input. LabVIEW ignores the value of the type input.

Related Information

Handling Variant Data

Converting LabVIEW Data to Variant Data

Deleting Attributes from Variant Data

Complete the following steps to delete an attribute(s) from *variant data*.

1. Add the Delete Variant Attribute function to the block diagram.
2. Wire variant data with attributes to the **variant** input of the Delete Variant Attribute function.
3. To delete all the attributes, leave the **name (all attributes)** input of the Delete Variant Attribute function unwired. To delete a specific attribute, enter the name of the attribute in the **name (all attributes)** input.
4. Run the VI.

Retrieving Attributes from Variant Data

Complete the following steps to retrieve a specific attribute from *variant data*.

1. Add the Get Variant Attribute function to the block diagram.
2. Wire variant data to the **variant** input of the Get Variant Attribute function.
3. Wire the name of the attribute to the **name** input of the Get Variant Attribute function.
4. Wire the **default value** input of the Get Variant Attribute function. What you wire to the **default value** input determines the type of indicator that is created when you wire the **value** output.
5. Right-click the **found** output of the Get Variant Attribute function and select **Create»Indicator** from the shortcut menu.

6. Right-click the **value** output of the Get Variant Attribute function and select **Create»Indicator** from the shortcut menu to retrieve the value associated with the attribute.
7. Run the VI.

If **found** is TRUE, the **value** indicator displays the value of the attribute. If **found** is FALSE, the **value** indicator displays the value you wired to the **default value** input.

Complete the following steps to retrieve all the attribute names from variant data.

1. Add the Get Variant Attribute function to the block diagram.
2. Wire variant data to the **variant** input of the Get Variant Attribute function.
3. Leave the **name** input and the **default value** input of the Get Variant Attribute function unwired.
4. Right-click the **names** output and select **Create»Indicator** from the shortcut menu.
5. Right-click the **values** output of the Get Variant Attribute function and select **Create»Indicator** from the shortcut menu to retrieve the values associated with each attribute.
6. Run the VI.

The names of the attributes appear in the front panel **names** array indicator. The associated values appear in the **values** array indicator as a list of variants, which you then can **convert** for use in VIs and functions.

Units

Associating Units with a Numeric Control or Indicator

You can associate physical units of measure, such as degrees Fahrenheit or kilometers/second, with any numeric control or indicator that has floating-point representation.

Complete the following steps to associate units with a numeric control or indicator.

1. Right-click the front panel control or indicator and select **Visible Items»Unit Label** from the shortcut menu. A unit label appears.



NOTE

National Instruments recommends you do not use units with a numeric object formatted for absolute time.

1. Enter a unit using correct **abbreviations** and **syntax**. For example, use a space to indicate multiplication and use a forward slash / to indicate division. If the unit syntax is incorrect, a question mark ? appears in the label.
2. **Enter a value** in the numeric object.



Note When you create a unit label of any type, LabVIEW automatically converts all data on the block diagram to SI units.

To change units, use the Operating tool or the Labeling tool to click inside the unit label and change the unit string. Complete the following steps to change units and avoid unit syntax errors.

1. Right-click the unit label and select **Build Unit String** from the shortcut menu to display the Build Unit String dialog box.
2. Select the appropriate unit in the **Units** listbox.
3. Click the **OK** button.



Note To associate units with a graph or chart, wire an object that has an associated unit to the graph or chart. Right-click the graph or chart and select **Visible Items»Unit Label** from the shortcut menu to display the unit acquired through the connection. By default, LabVIEW associates an SI unit with the graph or chart based on the unit associated with the object you wire to the graph or chart. You can change the unit only to a compatible unit that measures the same phenomenon. For example, if an input to a chart has a unit of **mi**, for miles, you can edit the chart unit to **ft**, **in**, or **m** but not to **N**, **Hz**, **min**, **acre**, or **A** because they are not length units.

You also can use the Text property to change units programmatically. The Unit Label property creates a reference that you can use with the Text properties.

Available Units in LabVIEW

The following tables display the units available in LabVIEW. LabVIEW recognizes both SI units and units from other systems of measurement. The symbol * indicates the base units LabVIEW uses.

SI Base Units

The base units in LabVIEW are the SI base units.

Quantity Name	Unit	Abbreviation
length	meter*	m
mass	kilogram*	kg
time	second*	s
electric current	ampere*	A
thermodynamic temperature	kelvin*	K

Quantity Name	Unit	Abbreviation
amount of substance	mole*	mol
luminous intensity	candela*	cd

SI Derived Units

LabVIEW also recognizes units derived from the SI base units. The following table shows SI derived units that LabVIEW recognizes.

Quantity Name	Unit	Abbreviation
area	square meter	m ²
volume	cubic meter	m ³
mass	gram	g
velocity	meters per second	m/s
acceleration	meters per second squared	m/s ²
wave number	reciprocal meter	m ⁻¹
mass density	kilograms per cubic meter	kg/m ³
specific volume	cubic meters per kilogram	m ³ /kg
current density	amperes per square meter	A/m ²
magnetic field strength	amperes per meter	A/m
amount-of-substance concentration	moles per cubic meter	mol/m ³
luminance	candelas per square meter	cd/m ²
mass fraction	kilogram per kilogram	kg/kg
plane angle	radian*	rad
solid angle	steradian*	sr
frequency	hertz	Hz
force	newton	N
pressure, stress	pascal	Pa
energy, quantity of heat, work	joule	J
power, radiant flux	watt	W
electric charge	coulomb	C

Quantity Name	Unit	Abbreviation
electric potential difference	volt	V
capacitance	farad	F
electric resistance	ohm	Ohm
electric conductance	siemens	S
magnetic flux	weber	Wb
magnetic flux density	tesla	T
inductance	henry	H
temperature (absolute)	degree Celsius	degC
luminous flux	lumen	lm
illuminance	lux	lx
activity (of a radionuclide)	becquerel	Bq
absorbed dose, kerma, specific energy (imparted)	gray	Gy
dose equivalent	sievert	Sv

Other Units

LabVIEW also recognizes various units from other non-SI systems of measurement. The following table lists these other units.

Quantity Name	Unit	Abbreviation
activity	curie	Ci
area	acre	acre
barn	b	
hectare	ha	
dynamic viscosity	poise	P
electric power	horsepower (electric)	hp
energy	British thermal unit (mean)	Btu
calorie (thermal)	cal	
electron volt	eV	

Quantity Name	Unit	Abbreviation
erg	erg	
force	dyne	dyn
pound-force	lbf	
illuminance	foot-candle	fc
length	ångström	Angstrom
astronomical unit	AU	
chain	chain	
cubit	cubit	
fathom	fathom	
fermi	fermi	
foot	ft	
furlong	furlong	
hubble	hubble	
inch	in	
light year	ly	
mile	mi	
rod	rod	
yard	yd	
lens power	diopetre	dpt
magnetism	emu	emu
gauss	Gs	
maxwell	Mx	
oersted	Oe	
mass	carat	CM
dram	dr	
grain	gr	
metric tonne	t	
ounce	oz	

Quantity Name	Unit	Abbreviation
pound-mass	lb	
slug	slug	
stone	stone	
ton (UK)	Ton	
ton (US short)	ton	
unified atomic mass unit	u	
plane angle	degree	deg
minute	'	
second	"	
pressure	atmosphere	atm
bar	bar	
meter of mercury	m.Hg	
pound per square inch	psi	
torr	torr	
radiation dose	Rad	Rad
radiation exposure	roentgen	r
radiation exposure: man	roentgen equivalent man	rem
temperature (absolute)	degree Fahrenheit	degF
temperature difference (relative)	Celsius degree	Cdeg
Fahrenheit degree	Fdeg	
time	day	d
hour	h	
minute	min	
year	a	
velocity	knot	kn
volume	barrel	barrel
bushel	bushel	
fluid dram	fl.dr	

Quantity Name	Unit	Abbreviation
fluid ounce	fl.oz	
gallon (imperial)	UKgal	
gallon (US)	gal	
gill	gill	
liter	l	
minim	minim	
peck	peck	
pint	pint	
quart	quart	

Creating Polymorphic Units

Create a *polymorphic* unit by entering \$x in the unit label of a front panel numeric control or indicator, where x is a number, for example \$1. You can think of \$x as a placeholder for the actual unit. When the VI is called, LabVIEW substitutes the units you pass in for all occurrences of \$x in that VI.

If you need to use more than one polymorphic unit, you can use the abbreviations \$2, \$3, and so on.

Complete the following steps to create a polymorphic unit in a VI.

1. Right-click a front panel numeric object and select **Visible Items»Unit Label** from the shortcut menu.
2. Type \$x, where x is any number, 1 through 9.
3. Click outside the front panel numeric object to end the editing session.

Unit Syntax

Use the following syntax when you *associate units with a numeric control or indicator* :

- Any unit can have any SI prefix. The prefix applies only to the unit directly after it.
- Use a space to indicate multiplication. LabVIEW interprets multiple spaces as a single space.
- Use a forward slash / to indicate division. If you use spaces around the forward slash, LabVIEW ignores the spaces. The forward slash applies only to the unit directly after it.

- Use a carat ^ to indicate exponentiation. Exponents can be positive or negative integers, except zero 0. The carat applies only to the unit directly before it.

The following table displays examples of unit strings with correct syntax and the mathematical equivalent.

Unit String	Mathematical Equivalent
ft lb	ft * lb
m/s	m / s (meters divided by seconds)
m /s/s	m / (s * s)
s^2	s^2
kg m/s	kg * m * s ⁻¹

Cutting, Copying, and Pasting Digital Data

Complete the following steps to cut digital data.

1. Select the row or column in the digital data control or digital waveform control you want to cut.
2. Right-click the row or column and select **Data Operations»Cut Data** from the shortcut menu.

Complete the following steps to copy digital data.

1. Select the area in the digital data control or digital waveform control you want to copy.
2. Right-click the area and select **Data Operations»Copy Data** from the shortcut menu.

Complete the following steps to paste digital data.

1. Select an area in the digital data control or digital waveform control that consists of the same number of rows or columns you copied.
2. Right-click the area and select **Data Operations»Paste Data** from the shortcut menu.

You can cut only whole rows or columns of data. You cannot create a new row or column with the digital data you cut. You must paste digital data into an area that is the same dimension as the area you cut or copied from. For example, if you copy four bits of data from one row, you must select four existing bits of data to paste over in the same row or in a different row. If you copy four bits of data from an area of two rows by two columns, you must paste the data into an area of two rows by two columns.

Exporting Data and Images

You can export data from graphs, charts, tables, and arrays. You also can export an image of a graph, chart, or table.

Exporting Data from Graphs, Charts, Tables, and Arrays

Complete the following steps to export data from a graph, chart, table, or array.

1. Right-click a graph, chart, table, or array.
2. Select **Export** from the shortcut menu, and then select one of the following options:
 - **Export Data To Clipboard**
 - **Export Data To Excel**
 - **Export Data To DIAdem**

You also can use the Export Data to Excel method to export data from a graph, chart, table, array or digital data control programmatically.

You can use the Export Data to Clipboard method to export data from a graph, chart, table, array or digital data control programmatically.

You can use the Export Data to DIAdem method to export data from a graph or chart programmatically.

Exporting Images of Graphs, Charts, and Tables

Complete the following steps to export an image of a graph, chart, table, picture control, digital data, or digital waveform control or indicator.

1. Right-click a graph, chart, table, picture control, digital data, or digital waveform control or indicator and select **Export»Export Simplified Image** from the shortcut menu to display the Export Simplified Image dialog box.



NOTE

The **Export Simplified Image** menu item is not available for intensity graphs, intensity charts, 3D graphs, or 3D picture controls.

1. In the **Export Simplified Image** dialog box, select the format of the file.
2. Select whether you want to save to the clipboard or to disk.
3. (Optional) Place a checkmark in the **Hide Grid** checkbox to hide the grid of a graph or chart.
4. Click the **Save** button to save the image to disk.

You also can use the Export Simplified Image method to export an image of a graph, chart, table, or digital data control programmatically.

You can use the Get Image method to export the image of a control or indicator programmatically.



RELATED INFORMATION

[Customizing Graphs and Charts](#) on page 493

[Customizing Graphs and Charts](#) on page 493

[Customizing Graphs and Charts](#) on page 493

Inserting and Deleting Rows and Columns in Digital Data Controls

To insert a row, right-click a sample in the sample column and select **Insert Row Before** from the shortcut menu.

To delete a row, right-click a sample in the sample column and select **Delete Row** from the shortcut menu.

To insert a column, right-click a signal in the signal column and select **Insert Column Before** from the shortcut menu.

To delete a column, right click a signal in the signal column and select **Delete Column** from the shortcut menu.

Comparing functions in LabVIEW

Comparing Arrays

Some Comparison functions have two modes for comparing arrays. In Compare Aggregates mode, the function returns a single Boolean value. In Compare Elements mode, the function compares the elements individually and returns an array of Boolean values.

In Compare Aggregates mode, the string comparison and array comparison operations follow exactly the same process—the Comparison function treats the string as an array of ASCII characters.



NOTE

The Max & Min function behaves differently than other comparison functions. The Max & Min function returns either the maximum element or array or the minimum element or array, depending on which mode you specify and which output you specify.



RELATED INFORMATION

[Setting Comparison Functions to Compare Elements or Aggregates](#) on page 1187

[Using Comparison Functions](#) on page 184

Comparing Boolean Values

Comparison functions treat the Boolean value TRUE as greater than the Boolean value FALSE.

Comparing Characters

Use the character Comparison functions Decimal Digit?, Hex Digit?, Octal Digit?, Printable?, White Space?, and Lexical Class to compare characters to determine the type of a character.

If the input to the Comparison function is a string, the functions test the first character. If the input is an empty string, the result is FALSE. If the input is a number, the functions interpret it as an ASCII character code. If the input is a floating-point number, the function rounds to the nearest integer.

Comparing Clusters

Clusters you compare must include the same number of elements, each element in the clusters must be of compatible types, and the elements must be in the same cluster order. For example, you can compare a cluster of a DBL and a string to a cluster of an I32 and a string.

In Compare Elements mode, Comparison functions return a cluster of Boolean elements, one for each corresponding element in the input clusters.

In Compare Aggregates mode, comparison functions return a single Boolean result after comparing the elements in a cluster. Comparison functions compare elements sequentially until a decision can be made, much like the alphabetical ordering of English words, where letters of each word are compared until they are not equal. Therefore, the function performs the following steps to determine the result of the comparison:

- The function compares corresponding elements in each input cluster, starting at the beginning of the cluster.
- If the corresponding elements are not equal, the function stops—it returns the result of this comparison.
- If the corresponding elements are equal, the comparison function processes the next pair of values, until it finds an inequality or reaches the end of the input clusters.

Comparing Numeric Values

Most of the Comparison functions test one input or compare two inputs and return a Boolean value. The functions convert numbers to the same representation before comparing them. The functions convert each input to the largest representation needed to make an accurate comparison. Comparisons with one or two inputs that have the value not a number NaN return a result that indicates that the comparison was not equal.

Comparing Strings

Comparison functions *compare strings* according to the values of the ASCII character codes. The Comparison functions compare strings one element at a time starting with the 0th element until they encounter an inequality or until one string ends. The functions treat the longer string as greater than the shorter string.

For example, a (with a decimal value of 97) is greater than A (65), which is greater than the numeral 0 (48), which is greater than the space character (32). LabVIEW compares characters one by one from the beginning of the string until an inequality occurs, at which time the comparison ends. For example, LabVIEW evaluates the strings abcd and abef until it finds c, which is less than the value of e. The presence of a character is greater than the absence of one. Thus, the string abcd is greater than abc because the first string is longer.

The functions that test the category of a string character, such as the Decimal Digit? and Printable? functions, evaluate only the first character of the string.

Determining Range Values and Coercing Data

Complete the following steps to determine if a value falls within a range specified by the **upper limit** and **lower limit** inputs of the In Range and Coerce function and optionally to coerce the value to fall within the range.



NOTE

The following steps use numeric values as an example. You can *determine if a value* in a string or Boolean object falls within a range and optionally coerce the value to fall within the range.

1. Add the In Range and Coerce function to the block diagram.

2. Wire a numeric value to the **x** input of the function.
3. Wire the maximum value for the range to the **upper limit** input of the function. If **x** is greater than this value, the function coerces the number to the **upper limit**. The default value is positive infinity.
4. Wire the minimum value for the range to the **lower limit** input of the function. If **x** is less than this value, the function coerces the number to the **lower limit**. The default value is negative infinity.
5. (Optional) Wire the **coerced(x)** output of the function to a subVI, function, or indicator that accepts a number. If the number in **x** is within the range set by the **upper limit** and **lower limit** inputs, the value in **x** remains unchanged. If the number in **x** is not in range, the function coerces the number to the **upper limit** or **lower limit**.
6. Wire the **In Range?** output of the In Range and Coerce function to a Boolean indicator to determine if the number in **x** is within the range set by the **upper limit** and **lower limit** inputs, or wire the **In Range?** output to a subVI or function that accepts a Boolean value as an input. If the number in **x** is within the range, **In Range?** returns a TRUE value. If the number is not within the range, **In Range?** returns a FALSE value.
7. Run the VI.

Determining Range Values and Coercing Data in Arrays

Complete the following steps to determine if each value in an array falls within a range specified by the **upper limit** and **lower limit** inputs of the In Range and Coerce function and to coerce the values to fall within the range.



NOTE

The following steps use numeric values as an example. You can *determine if a value* in an array of strings or an array of Boolean values falls within a range and optionally coerce the value to fall within the range.

1. Add the In Range and Coerce function to the block diagram.
2. Wire an array to the **x** input of the function.
3. Wire the maximum value for the range to the **upper limit** input of the function. If a value in **x** is greater than this value, the function coerces the number to the **upper limit**. The default value is positive infinity.
4. Wire the minimum value for the range to the **lower limit** input of the function. If a value in **x** is less than this value, the function coerces the number to the **lower limit**. The default value is negative infinity.



Note You can wire a scalar value or an array of numbers to **upper limit** or **lower limit**. If you wire a scalar value to **upper limit** or **lower limit**, LabVIEW checks each element in the array you wire to **x** against the scalar value limit. If you wire an array to **upper limit** or **lower limit**, LabVIEW checks each element in the array you wire to **x** against the corresponding element in the array you wired as a limit.

1. Wire the **coerced(x)** output of the function to a subVI, function, or indicator that accepts an array of numbers. If each element of the array you wired to **x** is within the range set by the **upper limit** and **lower limit** inputs, the value in **x** remains unchanged. If the element of the array you wired to **x** is not in range, the function coerces the element to the **upper limit** or **lower limit**.
2. Wire the **In Range?** output of the In Range and Coerce function to an array of Boolean values. Each Boolean value in the array corresponds to an element in the array you wired to **x** and indicates if the element is within the range you specify.
3. Run the VI.

Setting Comparison Functions to Compare Elements or Aggregates

Some Comparison functions have *two modes for comparing* arrays or clusters of data. In Compare Aggregates mode, the function returns a single Boolean value. In Compare Elements mode, the function compares the elements individually and returns an array or cluster of Boolean values.



NOTE

The Max & Min function behaves differently than other comparison functions. The Max & Min function returns either the maximum element or array or the minimum element or array, depending on which mode you specify and which output you specify.

To set a Comparison function that has two or more inputs to compare elements or aggregates in arrays and clusters, right-click the function and select **Comparison Mode»Compare Elements** or **Comparison Mode»Compare Aggregates** from the shortcut menu.

Showing and Hiding Labels and Digital Displays

Although you can delete most objects, you cannot delete owned labels or digital displays.

Complete the following steps to show or hide control, indicator, or wire labels and digital displays.

1. Right-click the object.
2. Select **Visible Items»Label** or **Visible Items»Digital Display** from the shortcut menu.

Importing Background Images for Panes

Complete the following steps to import a background image for a pane:

1. Right-click the scroll bar of the pane and select **Properties** from the shortcut menu to display the **Pane Properties** dialog box.
2. Navigate to the Background tab and select an image from the **Background** list. **Image Preview** displays a preview of the image, and the image appears in the pane. You also can complete the following steps to select an image that does not appear in the **Background** list.
 1. Click the **Browse** button to display a file dialog box.
 2. Navigate to the image and click the **OK** button. The image appears in the **Background** list. LabVIEW supports BMP, JPEG, and PNG graphic formats for background images. If you attempt to load an image of an unsupported graphic format, LabVIEW returns an error.

**NOTE**

If you select an image that does not appear in the **Background** list, LabVIEW does not add that image to the **Background** list permanently. To add an image to the list permanently, you must save the image in the `labview\resource\backgrounds` directory.

3. Use the **Position** pull-down menu to set the positioning of the image to stretch, center, or tile.
4. Click the **OK** button to close the **Pane Properties** dialog box and apply the background.

When you save a VI that contains a pane with a background image, LabVIEW saves the pane background image with the VI.

You also can use the Background Image property to set a pane background image programmatically. You can use the Background Mode property to set the positioning of a pane background image programmatically.

Importing Graphics on Linux

Complete the following steps to import an image (JPG, PNG, or an X Window System Dump XWD) into LabVIEW. You can create an XWD file using the `xwd` command.

1. Select **Edit»Import Picture to Clipboard**.
2. Use the dialog box that appears to navigate to the image file you want to import. Select the file and click the **OK** button.
3. (Optional) Click the pane or block diagram where you want to paste the image. If you do not click where you want to paste the image, LabVIEW pastes it in the center of the front panel or block diagram.
4. Select **Edit»Paste** in LabVIEW.

Complete the following steps to import an image (GIF, JPG, XWD, BMP, PNG or animated MNG) using the LabVIEW **Import Picture to Clipboard** feature.

1. Select **Edit»Import Picture to Clipboard**.

2. Use the dialog box that appears to navigate to the image file you want to import. Select the file and click the **OK** button.
3. (Optional) Click the pane or block diagram where you want to paste the image. If you do not click where you want to paste the image, LabVIEW pastes it in the center of the front panel or block diagram.
4. Select **Edit»Paste** in LabVIEW.

Importing Graphics on macOS

You can import an image from a graphics application, drag and drop an image file into LabVIEW, or use the **Edit»Import Picture to Clipboard** menu item.

Complete the following steps to import an image into LabVIEW.

1. In a graphics editing application or Web browser, copy an image to the clipboard.
2. Switch to LabVIEW. The image is automatically available on the LabVIEW clipboard.
3. (Optional) Click the pane or block diagram where you want to paste the image. If you do not click where you want to paste the image, LabVIEW pastes it in the center of the front panel or block diagram.
4. Select **Edit»Paste** to place the image in LabVIEW.



NOTE

If you import an image by copying and pasting it, the image loses any transparency.

Complete the following steps to drag a closed graphic file from a folder and place it in LabVIEW.

1. In the folder, select the graphics file you want to import.
2. Drag the file into the LabVIEW window and place the cursor where you want to paste it.
3. Release the mouse button. The image from the file appears in LabVIEW.

Complete the following steps to import an image (GIF, JPG, PNG, or animated MNG) using the LabVIEW **Import Picture to Clipboard** feature.

1. Select **Edit»Import Picture to Clipboard**.
2. Use the dialog box that appears to navigate to the image file you want to import. Select the file and click the **OK** button.
3. (Optional) Click the pane or block diagram where you want to paste the image. If you do not click where you want to paste the image, LabVIEW pastes it in the center of the front panel or block diagram.
4. Select **Edit»Paste** in LabVIEW.

Importing Graphics on Windows

You can import an image from a graphics application, drag and drop an image file into LabVIEW, or use the **Edit»Import Picture to Clipboard** menu item.

Complete the following steps to import any image format, except animated images, from a graphics application.

1. In the graphics editing application or Web browser, copy an image to the clipboard.
2. Switch to LabVIEW. The image is automatically available on the LabVIEW clipboard.
3. (Optional) Click the pane or block diagram where you want to paste the image. If you do not click where you want to paste the image, LabVIEW pastes it in the center of the front panel or block diagram.
4. Select **Edit»Paste** to place the image in LabVIEW.



NOTE

If you import an image by copying and pasting it, the image loses any transparency.

Complete the following steps to drag a graphics file (GIF, JPG, CLP, EMF, WMF, BMP, PNG, or animated MNG) from Windows Explorer and place it in LabVIEW.

1. In Windows Explorer, select the graphics file you want to import.
2. Drag the file into the LabVIEW window and place the cursor where you want to paste it.
3. Release the mouse button. The image from the file appears in LabVIEW.

Complete the following steps to import an image (GIF, JPG, CLP, EMF, WMF, BMP, PNG, or animated MNG) using the LabVIEW **Import Picture to Clipboard** feature.

1. Select **Edit»Import Picture to Clipboard**.
2. Use the dialog box that appears to navigate to the image file you want to import. Select the file and click the **OK** button.
3. (Optional) Click the pane or block diagram where you want to paste the image. If you do not click where you want to paste the image, LabVIEW pastes it in the center of the front panel or block diagram.
4. Select **Edit»Paste** in LabVIEW.

Editing the Labels of Buttons

Boolean controls and indicators have Boolean text labels in addition to their *owned labels*. Boolean text labels change depending on the value of the control or indicator. Initially, most Boolean controls and indicators display the word **ON** in the TRUE state and the word

OFF in the FALSE state. When you click the button with the Operating tool, the control toggles to the opposite state.



TIP

Label the Boolean object so that its name corresponds to the state of the control or indicator. For example, use ON for TRUE and OFF for FALSE. Also, indicate the default state in parenthesis.

By default, the label centers on the button. Complete the following steps to practice manipulating the label.

1. Add a push button to the front panel.
2. Right-click the button and select **Visible Items»Boolean Text** from the shortcut menu. The word **OFF** appears on the Boolean button.
3. With the Positioning tool, select the text and move it to the location you want. Also, you can select **Lock Text in Center** from the shortcut menu to return the text to its default location. To unlock the text, right-click the Boolean button and select **Release Text** from the shortcut menu.
4. Use the Labeling tool to change the text in either state. For example, change ON to YES.
5. Select the Boolean text and use the **Text Settings** pull-down menu on the toolbar to change the font, size, and color of the text.
6. Hide the Boolean text from both states by right-clicking the button and selecting **Visible Items»Boolean Text** from the shortcut menu.

By default, you can change the label for each state of labeled buttons located on the **Controls** palette. To do so for labeled buttons such as **OK** and **Cancel** buttons, you must edit the button to allow different text strings for each state.

Complete the following steps to edit a labeled button to allow multiple text strings.

1. Verify that the Boolean text is visible. To display the Boolean text, right-click the control and select **Visible Items»Boolean Text** from the shortcut menu.
2. Right-click the button and select **Advanced»Customize** from the shortcut menu to display the *Control Editor* window.
3. In the Control Editor window, right-click the button and select **Multiple Text Strings** from the shortcut menu.
4. Select **File»Apply Changes**.
5. If you want to use the button in other VIs, save it as a custom control.

You also can use the Boolean properties to manipulate Boolean labels programmatically.

Graphs and Charts

Getting Started with Graphs and Charts

Graphs and charts differ in the way they display and update data. Use graphs when you want to display a complete set of data and export the graph. Graphs are best for fast processes that acquire data continuously.

Use charts when you want to collect and display the data simultaneously. Charts are best for slow processes that only plot a few data points per second. Refer to the following support document at ni.com for more information about the differences between graphs and charts.

Use the following topics to get started with graphs and charts.

Choosing and Creating a Graph or Chart	Modifying the Plot Legend of a Graph or Chart
• Choosing the Right Graph or Chart • Creating a Graph • Creating a Chart	• Adding a Plot to a Plot Legend • Setting the Number of Plots in a Plot Legend
Customizing Your Graph or Chart	Navigating a Graph or Chart
• Configuring a Graph or Chart • Customizing Graphs and Charts • Changing the Format and Precision of Graph and Chart Axes • Configuring the X- and Y-Axes on a Graph or Chart • Adding Markers to X- and Y-Scales on a Graph or Chart • Adding Multiple X- and Y-Scales on Graphs or Charts	• Scrolling through a Graph or Chart • Zooming and Moving the View of a Graph or Chart
Exporting and Clearing Graph or Chart Data	

- Exporting Data and Images
- Clearing the Data on Graphs and Charts



RELATED INFORMATION

[Graph Subpalette](#) on page 151

[Creating a Graph](#) on page 1204

[Creating a Chart](#) on page 1222

[Adding a Plot to a Plot Legend](#) on page 1193

[Setting the Number of Plots in a Plot Legend](#) on page 1198

[Configuring a Graph or Chart](#) on page 1196

[Customizing Graphs and Charts](#) on page 493

[Changing the Format and Precision of Graph and Chart Axes](#) on page 1195

[Configuring the X- and Y-Axes on a Graph or Chart](#) on page 1196

[Adding Markers to X- and Y-Scales on a Graph or Chart](#) on page 1194

[Adding Multiple X- and Y-Scales on Graphs or Charts](#) on page 1194

[Scrolling through a Graph or Chart](#) on page 1198

[Zooming and Moving the View of a Graph or Chart](#) on page 1199

[Exporting Data and Images](#) on page 1182

[Clearing the Data on Graphs and Charts](#) on page 1195

Adding a Plot to a Plot Legend

You can allocate, or set aside, a plot in the plot legend for data that LabVIEW will draw to the graph or chart, as well as [customize the allocated plot](#).

To add a plot to a plot legend of a graph or chart, move the Positioning tool over the plot legend border. When resizing handles appear at the corners of the legend and in the middle of each border, drag a handle up, down, left, or right and release the mouse button when the legend contains the correct number of plots.

You also can click the increment arrow on a [plot legend scroll bar](#) to expose an empty plot. LabVIEW dims an empty plot in a plot legend. Click the empty glyph in the empty plot or enter a plot name to create a new plot.

Adding a Scroll Bar to a Plot Legend

Use a scroll bar to view plots in a plot legend without exposing all the plots at any one time.

To add a scroll bar to a plot legend, right-click the plot legend and select **Visible Items»Horizontal Scrollbar** or **Visible Items»Vertical Scrollbar** from the shortcut menu.

Adding Markers to X- and Y-Scales on a Graph or Chart

Complete the following steps to add a marker to an x- or y-scale on a [graph or chart](#).

1. Right-click the x- or y-scale and select **Marker Spacing»Arbitrary** from the shortcut menu to configure the scale to use arbitrary marker spacing. LabVIEW removes all markers on the scale except the two end markers.
2. Right-click the position on the scale where you want to add a new marker and select **X Scale»Add Marker** or **Y Scale»Add Marker** from the shortcut menu. LabVIEW adds a new marker to that location. You also can press the <Ctrl> key while you drag an existing marker to the location where you want to add a marker, or you can drag one of the end markers to the location where you want to add a marker. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.
3. (Optional) Use the Operating tool or the Labeling tool to change the value of the new marker.

To delete a marker, right-click the marker and select **Delete Marker** from the shortcut menu.

Adding Multiple X- and Y-Scales on Graphs or Charts

Waveform, XY, intensity, and digital waveform graphs support multiple x- and y-scales. Mixed signal graphs and all charts support only multiple y-scales. Use multiple scales on a graph or chart to display multiple plots that do not share a common x- or y-scale.

Complete the following steps to add a scale to a [waveform graph or chart](#).

1. Right-click the scale of the waveform graph or chart and select **Duplicate Scale** from the shortcut menu.
2. To move the new scale to the opposite side of the display, right-click the new scale and select **Swap Sides** from the shortcut menu.
3. Associate a plot to the new scale by right-clicking the plot in the plot legend and select from the **X Scale** or **Y Scale** shortcut menus. If the plot legend is not visible, right-click the graph or chart and select **Visible Items»Plot Legend** from the shortcut menu.



NOTE

You also can use the **Properties** dialog box to associate a plot with a scale. Right-click the waveform graph and select **Properties** from the shortcut menu. On the Plots page, use the Y-scale and X-scale pull-down menus to associate a plot with the correct scale.

1. To change the range of the new scale, use the Operating tool or the Labeling tool to highlight the end value(s) of the scale and enter a new value. If you set the scale manually, you should [disable autoscaling](#).

Changing the Format and Precision of Graph and Chart Axes

Change the format of an axis to display a different numeric or time format on the axis of a graph or chart. Change the precision to specify the number of digits the axis displays.

Complete the following steps to change the format and precision of a [graph or chart](#) axis.

1. Right-click the graph or chart and select **Properties**, **3D Plot Properties**, or **3D Graph Properties**.
2. Select the Display Format or Format tab.
3. Select the axis you want to configure.
4. Select a format from **Type**. If the [representation](#) of the numbers on the scale is floating-point, you cannot display the numbers on the axis as hexadecimal, octal, or binary.
5. Use **Digits** and **Precision Type** to change the precision of the numbers on the axis.



NOTE

Select **Advanced editing mode** to [enter format strings](#) and customize the precision of numbers on the axes.

1. Click **OK**.

Clearing the Data on Graphs and Charts

Complete the following steps to clear the data on a [graph or chart](#) manually.

1. Run a VI that is generating data and displaying that data on a graph or chart.
2. Right-click the graph or chart in a running VI and select **Clear Graph** or **Clear Chart** from the shortcut menu, respectively. LabVIEW clears the data on the graph or chart. Select **Data Operations»Clear Graph** or **Data Operations»Clear Chart** from the shortcut menu, respectively, in a VI that has completed running.

You also can [clear graphs and charts automatically](#) each time you run or call a VI.

You also can use the History Data property to clear a chart programmatically.

Refer to the Clearing Waveform Charts VI in the `labview\examples\Controls and Indicators\Graphs and Charts\Waveform Graphs and Charts` directory for an example of clearing graphs and charts programmatically.

Configuring a Graph or Chart

Right-click a graph or chart and select one of the following shortcut menu options to [configure a graph or chart](#).

- **Stack Plots** —Displays multiple plots on the chart by using multiple vertical scales. Remove the checkmark from this shortcut menu option to overlay the plots and display multiple plots on the chart by using a single vertical scale. This option is available only for charts.
- **Chart History Length** —Launches the Chart History Length dialog box. This option is available only for charts.
- **Transpose Array** —Switches the x- and y- data before plotting. You also can use the Transpose Array property to switch the x- and y- data before plotting programmatically.
- **Autosize Plot Legend** —Automatically resizes the plot legend to the width of the longest plot name visible in the legend.



NOTE

On a digital waveform graph, **Autosize Plot Legend** is only visible when the plot legend is in [standard view](#).

You also can [configure the x- and y-axes](#) on a graph or chart.

Configuring the X- and Y-Axes on a Graph or Chart

Right-click the x- or y-scale of a [graph or chart](#) and select one of the following shortcut menu options to configure the x- or y-axis. You also can access these options by right-clicking the graph or chart and selecting from the **X Scale** or **Y Scale** shortcut menus. You also can use the X Scale and Y Scale properties to select a scale programmatically.



NOTE

The following shortcut menu options do not apply to 3D graphs.

- **Marker Spacing** —Uses **Uniform** (default) or **Arbitrary** marker spacing. By default, scale markers are uniform, or evenly spaced. By selecting arbitrary marker spacing, you can indicate specific inner markers for a scale. [Arbitrary marker spacing](#) is useful for marking a few specific points on a slide or rotary control, such as a set point or threshold.
- **Add Marker** — [Adds a marker](#) to the scale. This option is not available if **Marker Spacing** is **Uniform**.
- **Delete Marker** — [Deletes a marker](#) from the scale. This option is not available if **Marker Spacing** is **Uniform**, and if the scale contains only the two end markers.
- **Formatting** —Displays a dialog box you can use to [format](#) the x- or y-axis.
- **Style** —Sets the style for the marks on the scale.
- **Mapping** —Select **Linear** (default) or **Logarithmic**.

- **AutoScale X** or **AutoScale Y** — *Autoscales* the x- or y-scales based on the data you wire to them.
- **Loose Fit** —Place a checkmark next to this option to round the end markers to a multiple of the increment used for the scale.
- **Visible Scale Label** —Displays the Scale Label.
- **Expand Digital Buses** —Displays digital waveform data as individual data lines. This option is available only for digital waveform graphs. You also can use the Expand Digital Buses property to set how the data is displayed programmatically.
- **Swap Sides** —Moves the scale to the opposite side of the display. This option is useful if you have two scales on one side of the graph or chart.



Note After you configure the x- or y-scale on a graph or chart, you can select **Advanced»Reset Scale Layout** from the shortcut menu to reset the position of the x- or y-scales.

- **Duplicate Scale** —Copies the scale and *adds it as a new scale* to the graph or chart.
- **Delete Scale** —Deletes the scale. This option is available only if the graph or chart has multiple x- or y-scales.

You also can use the GraphChart, GraphScale, and Scale properties to configure the X- or Y-scales of a graph or chart programmatically.

Customizing an Analog Plot Using the Plot Legend

Use the plot legend to customize how each plot appears in the plot area of a graph or chart. Click a glyph in the plot legend and select from the following options in the shortcut menu to customize how the plot appears in the plot area of a graph or chart:



NOTE

You also can configure a digital waveform plot to specify how numbers and plot lines appear on a digital waveform graph.

- **Plot Visible** —Sets whether to display the plot on the graph or chart.



Note If a graph or chart has multiple plots, right-click the plot legend and select **Visible Items»Plot Visibility Checkbox** to display a checkbox next to each plot. Remove checkmarks from the plots you do not want to display.

- **Common Plots** —Provides options for the plot type. You can select a line, scatter, line and symbol, fill, comb, or bar plot type.
- **Color** —Displays the color picker so you can select the color of the plot.
- **Line Style** —Provides solid and dashed line styles.
- **Line Width** —Provides line widths.

- **Anti-Aliased** —Place a checkmark next to this item to make line plots appear smoother. Anti-aliased line drawing can be computation intensive. Using anti-aliased line plots can slow performance.



Note Anti-aliased line drawing is not available on digital waveform graphs.

- **Bar Plots** —Provides bar plot options.
- **Fill Base Line** —Provides options for filling in space below the plot with the same color as the plot. If the graph or chart has multiple plots, you can select another plot at the bottom of the shortcut menu to fill the space below the plot up to the other plot.
- **Interpolation** —Provides interpolation options, including plotting only data points.
- **Point Style** —Provides point styles for the data points.



Note To change the size of the data points, you must use the **Line Width** option to change the line width for the plot.

- **X Scale** —Sets the variable to associate with the x-axis.
- **Y Scale** —Sets the variable to associate with the y-axis.

You also can use the Plot properties to customize how a plot appears in the plot area programmatically.

Scrolling through a Graph or Chart

Right-click a graph or chart and select **Visible Items»X Scrollbar** from the shortcut menu to display a scroll bar below the graph or chart. Use the Operating tool to click the scroll bar and scroll along the x-axis.

You also can use the X Scrollbar Visible property to display the x-axis scroll bar programmatically.



RELATED INFORMATION

[Types of Graphs and Charts](#) on page 477

Setting the Number of Plots in a Plot Legend

You can specify the number of plots to show in a plot legend at run time. You also can specify the number of plots to show in the plot legend regardless of the number of plots in the plot area of a graph or chart.

To specify the number of plots to show in the plot legend of a graph or chart, right-click the plot area of a graph or chart and select **Properties** from the shortcut menu. On the **Appearance** page, increase or decrease the minimum number of plots to show in the plot legend by changing the number in the **Plots shown** field. In the **Plots shown** field, enter the number of plots manually, or click the increment and decrement buttons to increase or decrease the number of plots to show.

You also can use the Legend: Number of Rows property to set the number of plots LabVIEW shows in the plot legend programmatically. Use the Legend: Plot Minimum property to set the minimum number of plots to show in the plot legend regardless of the number of plots in the plot area of a graph or chart.

Zooming and Moving the View of a Graph or Chart

To zoom in or out or to move the view to an area not visible in a graph or chart, right-click the *graph or chart* and select **Visible Items»Graph Palette** to display the graph palette, shown in the following image.



NOTE

You also can click  to move the *cursor* on a graph.

Zooming In or Out

Click , and select from the following images to zoom in or out.

	Zooms in an area of the graph or chart.
	Zooms in an area of the graph or chart on the x-axis.
	Zooms in an area of the graph or chart on the y-axis.
	Zooms in or out to <i>autoscale the graph or chart</i> .
	Zooms in. Hold <Shift> to zoom out while you are zooming in, and release <Shift> to zoom in again.
	Zooms out. Hold <Shift> to zoom in while you are zooming out, and release <Shift> to zoom out again.

Moving the View to an Area Not Visible

Complete the following steps to move the view to an area not visible in the graph or chart.

1. Click .
2. Click the graph or chart, without releasing the mouse.
3. Move the mouse to either side of the axes.

Adding Annotations to Graphs

Use an annotation on a *graph* to *highlight data points on the plot area*. Complete the following steps to add an annotation to a graph.

1. Right-click the graph and select **Data Operations»Create Annotation** from the shortcut menu to display the Create Annotation dialog box. Use the **Create Annotation** dialog box to specify the annotation name and how the annotation snaps to plots in the plot area.
2. Click the **OK** button to close the **Create Annotation** dialog box. The new annotation appears on the graph.
3. Right-click the annotation and use the following options from the shortcut menu to customize the annotation:
 - **Lock Style** —Sets the way in which the annotation snaps to plots and can be moved in the plot area. **Lock Style** includes the following options:
 - **Snap to All Plots** —Allows you to move the annotation to the nearest data point along any plot in the plot area.
 - **Snap to One Plot** —Allows you to move the annotation only along the specified plot.
 - **Free** —Allows you to move the annotation anywhere in the plot area. LabVIEW does not snap the annotation to any plots in the plot area.
 - **Attributes** —Customize the appearance and behavior of each annotation. **Attributes** includes the following options:
 - **Color** —Displays the *color picker* so you can select the color of the annotation, including its point, arrow, and name.
 - **Annotation Style** —Provides various styles for the annotation. LabVIEW draws the selected style, in combination with the **Point Style** option, at the focal point of the annotation.
 - **Point Style** —Provides various point styles for the annotation. LabVIEW draws the selected style, in combination with the **Cursor Style** option, at the focal point of the annotation.
 - **Line Style** —Provides various solid and dotted line styles for the annotation.
 - **Line Width** —Provides various line widths for the annotation.
 - **Show Name** —Displays the annotation name in the plot area.
 - **Allow Drag** —Allows you to drag the annotation.
 - **Lock Name** —Positions the annotation name absolutely so that when you move the annotation or scroll the plot area, the annotation name does not move in the plot area.

- **Show Arrow** —Displays the arrow that points from the annotation name to the annotated data point.
- **X Scale** —Allows you to choose the x-scale that maps a pixel location to a data value for the annotation. This option applies only to free annotations.
- **Y Scale** —Allows you to choose the y-scale that maps a pixel location to a data value for the annotation. This option applies only to free annotations.
- **Delete Annotation** —Deletes the annotation from the graph.

**NOTE**

Right-click the graph and select **Data Operations»Delete All Annotations** from the shortcut menu to delete all annotations in the plot area.

1. Click and drag the crosshairs of the annotation to move the annotation to a different data point in the plot area. Click and drag the annotation name to move it to a different location in the plot area.

Use the Annotation List property to get an array of information about all annotations in the plot area programmatically.

Adding Cursors to Graphs

Complete the following steps to add a *cursor* to a *graph*.

1. Right-click the graph and select **Visible Items»Cursor Legend** from the shortcut menu.
2. Right-click anywhere in the cursor legend, select **Create Cursor**, and select a cursor mode from the shortcut menu. The cursor position is defined by the cursor mode. The cursor includes the following modes:
 - **Free** —Moves the cursor freely within the plot area, regardless of plot positions.
 - **Single-Plot** —Positions the cursor only on the plot that is associated with the cursor. You can move the cursor along the associated plot. Right-click the cursor legend row and select **Snap To** from the shortcut menu to associate one or all plots with the cursor.
 - **Multi-Plot** —Positions the cursor only on a specific data point in the plot area. The multi-plot cursor reports values at the specified x-value for all of the plots with which the cursor is associated. You can position the cursor on any plot in the plot area. Right-click the cursor legend row and select **Snap To** from the shortcut menu to associate one or all plots with the cursor. This mode is only valid for mixed signal graphs.

**NOTE**

You cannot change the mode of a cursor after you create it. You must *delete the cursor* and create another cursor.

1. Click anywhere in the cursor legend row to select the cursor. Use the Operating tool or the Labeling tool to enter names and coordinates in the cursor legend.
2. Right-click the cursor legend row and select from the following options from the shortcut menu to customize the cursor:

- **Watch** —When the cursor mode is **Multi-Plot**, associates one or more specified plots or all plots with the cursor. The multi-plot cursor reports values at the specified x-value for all of the plots with which the cursor is associated.

If you initially select individual plots to watch and then select **Watch»All Plots** instead, the cursor begins to report values for all plots. If you then deselect **All Plots**, the cursor reports values for the plots you initially selected.

- **Snap To** —Locks the cursor to a plot. When the cursor mode is **Single-Plot**, you can snap the cursor to one or all plots. When the cursor mode is **Multi-Plot**, **Snap To** only affects how the cursor moves along the x-axis.

Note If no data exists in the plot area, the cursor snaps to only **All Plots** or the first plot listed in the plot legend. You must have data for more than one plot to snap a cursor to plots other than **All Plots** or the first plot in the plot legend. For example, you must have three data sets in the plot area before you can snap a cursor to the third plot.

- **X Scale** —Sets the x-scale of the cursor. This option is valid only if the cursor mode is **Free**.
- **Y Scale** —Sets the y-scale of the cursor. This option is valid only if the cursor mode is **Free**.
- **Attributes** —Customizes the appearance of each cursor. **Attributes** includes the following options:
 - **Color** —Displays the *color picker* so you can select the color of the cursor.
 - **Cursor Style** —Provides various cursor styles.
 - **Point Style** —Provides various point styles for the intersection of the cursor.
 - **Line Style** —Provides various solid and dotted line styles.
 - **Line Width** —Provides various line widths.
 - **Show Name** —Displays the name of the cursor on the graph. Use the Positioning tool to move the name in relation to the cursor.
 - **Allow Drag** —Allows the mouse to drag the cursor.
- **Bring to Center** —Centers the cursor on the graph without changing the x- and y-scales. When the cursor mode is **Single-Plot** or **Multi-Plot**, this option centers the cursor on the plot on which the cursor is currently positioned and updates the cursor coordinates in the cursor legend. When the cursor mode is **Free**, this option centers the cursor in the plot area and updates the cursor coordinates in the cursor legend.
- **Go to Cursor** —Changes the x- and y-scales to show the cursor at the center of the graph.

You also can use the Cursor properties to customize a cursor programmatically.

Adding Plot Areas to a Mixed Signal Graph

A mixed signal graph may have multiple plot areas. A given plot area can display only digital or analog plots, not both. The mixed signal graph automatically creates plot areas when necessary to accommodate analog and digital data. To add a plot area to a mixed signal graph, right-click the mixed signal graph and select **Add Plot Area** from the shortcut menu.

To remove a plot area from the graph, remove any data from the plot area, right-click the plot area you want to remove, and select **Remove Plot Area** from the shortcut menu.



NOTE

If you grow the graph so you can view multiple plot areas and then remove a plot area, LabVIEW resizes the remaining plot areas to fill the frame of the graph. You can reduce the size of the graph to reduce the size of the plot areas.

Changing the Plot Legend View in a Digital Waveform Graph

The digital waveform graph provides two views of the plot legend. Change the view of the plot legend to either the [standard view](#) or [tree view](#) to display digital line and bus information. The plot legend in tree view allows you to expand and contract digital buses both in the plot legend and in the plot area of the graph, while the plot legend in standard view allows you to view digital lines.

To change from the tree view of the plot legend to the standard view of the plot legend, right-click the digital waveform graph and select **Advanced»Switch Plot Legend to Standard View** from the shortcut menu. To change back to the tree view of the plot legend, right-click the digital waveform graph and select **Advanced»Switch Plot Legend to Tree View** from the shortcut menu.



NOTE

LabVIEW deletes any [allocated lines](#) when you change from the standard view of the plot legend to the tree view of the plot legend, including any [custom line settings](#). Additionally, you cannot allocate lines in the plot legend when the tree view is enabled.

To disable the plot legend regardless of the selected view, right-click the digital waveform graph and select **Advanced»Disable Plot Legend** from the shortcut menu to place a checkmark next to the menu item. You also can use the Legend:Disable property to disable the plot legend programmatically.

You also can use the Digital Tree View property to change the view of the plot legend on a digital waveform graph programmatically.

Creating a Graph

VIs with a graph usually collect the data in an array and then plot the data to the graph. This process is similar to a spreadsheet that first stores the data then generates a plot of it.

Complete the following steps to create a graph.

1. Add one of the following graphs to the front panel window.
 - Waveform graph
 - XY graph
 - Digital waveform graph
 - Mixed signal graph
 - Intensity graph
2. Right-click the graph and customize it using the shortcut menu items. You can configure the x- and y-scales, format the x- and y-scales, add a plot, zoom in or out, and add cursors.

3. Build the block diagram to display data on a waveform, XY, mixed signal, or digital waveform graph.

**TIP**

You can export data and images from graphs to locations such as Microsoft Excel. Right-click a graph and select **Export** from the shortcut menu to view available export options.

**RELATED INFORMATION**

[Graphs and Charts](#) on page 476

[Graphing Waveform Data](#) on page 1214

[Graphing XY Data](#) on page 1215

[Graphing Digital Data](#) on page 1213

[Types of Graphs and Charts](#) on page 477

[Types of Graphs and Charts](#) on page 477

[Configuring the X- and Y-Axes on a Graph or Chart](#) on page 1196

[Changing the Format and Precision of Graph and Chart Axes](#) on page 1195

[Adding a Plot to a Plot Legend](#) on page 1193

[Zooming and Moving the View of a Graph or Chart](#) on page 1199

[Adding Cursors to Graphs](#) on page 1201

[Graphing Waveform Data](#) on page 1214

[Graphing XY Data](#) on page 1215

[Graphing Multiple Data Types on a Mixed Signal Graph](#) on page 1213

[Graphing Digital Data](#) on page 1213

[Exporting Data and Images](#) on page 1182

Customizing a Plot on a Digital Waveform Graph Using the Plot Legend

You can specify how numbers and plot lines appear on a digital waveform graph by configuring each plot. To configure a digital waveform plot, click the glyph in the plot legend and select one of the following options from the shortcut menu:

**NOTE**

You also can configure a plot to customize how each plot appears in the plot area of a graph or chart.

- **Plot Visible** —Sets whether to display the plot on the graph or chart.
- **Color** —Displays the color picker so you can select the color of the plot.

- **Label Format** —Sets the format of the numbers in a bus. You can display numbers in **Hexadecimal**, **Decimal**, **Octal**, **Binary**, or **Fixed Point** formats, or you can select a format of **None**, which removes the labels from the line or bus. You also can select **Configure Fixed Point** to display the Configure Fixed Point dialog box, which allows you to configure **Fixed Point** label format options.
- **Transition Type** —Sets how LabVIEW distinguishes differing values in the plot. This setting affects only plots with more than one bit. You can select a square edge or sloped edge. Use the square edge to show simple state changes. Use the sloped edge to emphasize jitter or settling time between states.
- **Transition Location** —Sets whether to display high-to-low transitions at the previous point, in between points, or at the new point on the x-axis. The default displays high to low transitions at the new point on the x-axis.
- **Line Style** —Sets whether LabVIEW uses a thin or thick line in the plot to distinguish high or low values or to offset a particular plot line. Select the left-most option to keep the default line thickness.
- **X Scale** —Sets the variable to associate with the x-axis.
- **Y Scale** —Sets the variable to associate with the y-axis.

You also can use the Plot properties to customize how a plot appears in the plot area programmatically.

Customizing a Plot on a Mixed Signal Graph Using the Plot Legend

None

Customizing Digital Data Plots

You can specify how numbers and digital lines appear on a mixed signal graph by customizing each line or bus. To customize a digital line or bus on a graph, right-click a line or bus glyph in the plot legend and select from the following options on the shortcut menu:

- **Plot Visible** —Sets whether to display the plot on the graph or chart.
- **Color** —Displays the color picker so you can select the color of the plot.
- **Label Format** —Sets the format of the numbers in a bus. You can display numbers in **Hexadecimal**, **Decimal**, **Octal**, **Binary**, or **Fixed Point** formats, or you can select a format of **None**, which removes the labels from a line or bus. You also can select **Configure Fixed Point** to display the Configure Fixed Point dialog box, which allows you to configure **Fixed Point** label format options.



NOTE

LabVIEW displays the plot label only if there is enough room in the digital graph plot.

- **Transition Type** —Sets how LabVIEW distinguishes differing values in the plot. This setting affects only plots with more than one bit. You can select a square edge or sloped edge. Use the square edge to show simple state changes. Use the sloped edge to emphasize jitter or settling time between states.
- **Transition Location** —Sets whether to display high-to-low transitions at the previous point, in between points, or at the new point on the x-axis. The default displays high-to-low transitions at the new point on the x-axis.
- **Line Style** —Sets whether LabVIEW uses a thin or thick line in the plot to distinguish high or low values or to offset a particular plot line. Select the left-most option to keep the default line thickness.
- **X Scale** —Sets the variable to associate with the x-axis.
- **Y Scale** —Sets the variable to associate with the y-axis.

You also can use the Plot properties to customize how a digital data plot appears in the plot area programmatically.

Related information Plot Properties Configure Fixed Point Dialog Box [Adding Plot Areas to a Mixed Signal Graph](#) [Graphing Multiple Data Types on a Mixed Signal Graph](#) [Customizing a Digital Waveform Plot Using the Plot Legend](#)

Customizing Analog Data Plots

You can specify how numbers and analog plots appear on a mixed signal graph by customizing each plot. To customize an analog plot on a mixed signal graph, right-click a glyph in the plot legend and select one of the following options from the shortcut menu:

- **Plot Visible** —Sets whether to display the plot on the graph or chart.
- **Common Plots** —Sets the plot type. You can select a line, scatter, line and symbol, fill, comb, or bar plot type.
- **Color** —Displays the color picker so you can select the color of the plot.
- **Line Style** —Sets the line style of the selected plot line. You can select from solid and dashed line styles.
- **Line Width** —Sets the thickness of the selected plot line.
- **Anti-Aliased** —Sets whether to use anti-aliasing. Select this item to make line plots appear smoother. Anti-aliasing can be computation intensive and can slow performance.
- **Bar Plots** —Sets whether and how LabVIEW displays bar plots.
- **Fill Base Line** —Sets options for filling in space above or below the plot with the same color as the plot. If the graph or chart has multiple plots, you can select a second plot at the bottom of the shortcut menu to fill the space below the plot up to the first plot.
- **Interpolation** —Sets interpolation options, including plotting only data points.
- **Point Style** —Sets point styles for the data points of the selected plot line.



Note To change the size of the data points, you must use the **Line Width** shortcut menu option to change the line width for the plot.

- **X Scale** —Sets the variable to associate with the x-axis.
- **Y Scale** —Sets the variable to associate with the y-axis.

You also can use the Plot properties to customize how an analog data plot appears in the plot area programmatically.

Related information Plot Properties [Adding Plot Areas to a Mixed Signal Graph Graphing Multiple Data Types on a Mixed Signal Graph Customizing an Analog Plot Using the Plot Legend](#)

Deleting Cursors from Graphs

Complete the following steps to delete a *cursor* from a *graph*.

1. Right-click the graph and select **Visible Items»Cursor Legend** from the shortcut menu.
2. In the cursor legend that appears, right-click the row of the cursor you want to delete and select **Delete Cursor** from the shortcut menu.

Displaying Digital Lines and Buses in a Digital Waveform Graph

Customize the digital waveform graph to display only the data you want to view in the plot area. Complete the steps in the following sections to customize the digital waveform graph to display digital lines and buses, digital lines, or digital buses.

Displaying Digital Lines and Buses

Use the Group Digital Signals Express VI to organize and label digital lines and buses into a digital array. The Express VI returns the lines and buses in a digital array in the order you specify in the configuration dialog box for the Express VI.

By default, the digital waveform graph displays contracted digital lines and buses. To ensure this default behavior is enabled, right-click the digital waveform graph and confirm **Advanced»Show Buses With Lines** has a checkmark next to the menu item.

Displaying Digital Lines Only

By default, LabVIEW enables **Advanced»Show Buses With Lines**. To display only digital lines, right-click the digital waveform graph and select **Advanced»Show Buses With Lines** from the shortcut menu to remove the checkmark next to the menu item.

Displaying Digital Buses Only

By default, LabVIEW enables **Advanced»Show Buses With Lines** and **Y Scale»Expand Digital Buses**. To display only digital buses, disable **Advanced»Show Buses With Lines** by right-clicking the digital waveform graph and selecting **Advanced»Show Buses With Lines** from the shortcut menu to remove the checkmark next to the menu item. Right-click the digital waveform graph and select **Y Scale»Expand Digital Buses** to remove the checkmark next to the menu item.



NOTE

Y Scale»Expand Digital Buses is available only if **Advanced»Show Buses With Lines** is disabled and the plot legend is in *standard view*.

You also can use the Show Buses With Lines property to customize the digital lines and buses that appear in the plot area of the digital waveform graph programmatically.

Displaying Special Planes on the XY Graph

You can display *Nyquist planes*, *Nichols planes*, *S planes*, and *Z planes* on the *XY graph*.



NOTE

Selecting a plane from the **Optional Plane** submenu does not automatically scale the graph to display the plane. Certain values lie outside the visible range of these planes. The following steps contain values you can use to view the planes.

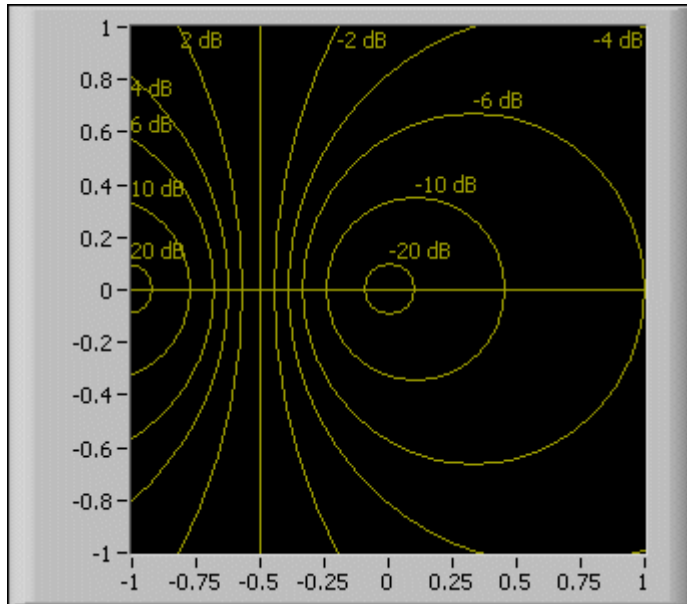
Displaying a Nyquist Plane

Complete the following steps to display a Nyquist plane.

1. Add an XY graph to the front panel.
2. Double-click the minimum and maximum values on the x- and y- scale and change both minimum values to -1 and both maximum values to 1.
3. Right-click the XY graph and select **Optional Plane»Nyquist** from the shortcut menu.
4. (Optional) To view the Nyquist plane without labels, right-click the XY graph and remove the checkmark from the **Optional Plane»Show Nyquist Labels** shortcut menu item.
5. (Optional) To view the Nyquist plane without the Cartesian lines, right-click the XY graph and remove the checkmark from the **Optional Plane»Show Cartesian Lines** shortcut menu item.

To remove the Nyquist plane, right-click the XY graph and select **Optional Plane»None** from the shortcut menu.

The following front panel shows a Nyquist plane without the Cartesian lines.



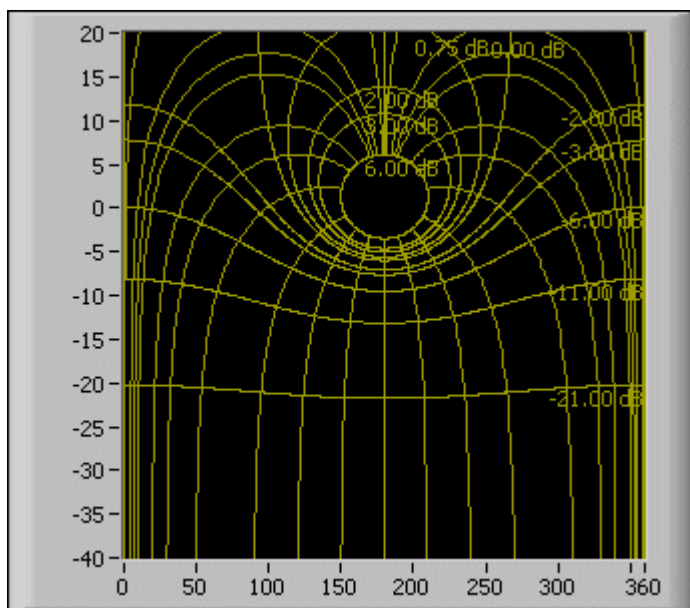
Displaying a Nichols Plane

Complete the following steps to display a Nichols plane.

1. Add an XY graph to the front panel.
2. Double-click the minimum and maximum values on the x-scale and change the minimum value to 0 and the maximum value to 360.
3. Double-click the minimum and maximum values on the y-scale and change the minimum value to -40 and the maximum value to 20.
4. Right-click the XY graph and select **Optional Plane»Nichols** from the shortcut menu.
5. (Optional) To view the Nichols plane without labels, right-click the XY graph and remove the checkmark from the **Optional Plane»Show Nichols Labels** shortcut menu item.
6. (Optional) To view the Nichols plane without the Cartesian lines, right-click the XY graph and remove the checkmark from the **Optional Plane»Show Cartesian Lines** shortcut menu item.

To remove the Nichols plane, right-click the XY graph and select **Optional Plane»None** from the shortcut menu.

The following front panel shows a Nichols plane without the Cartesian lines.



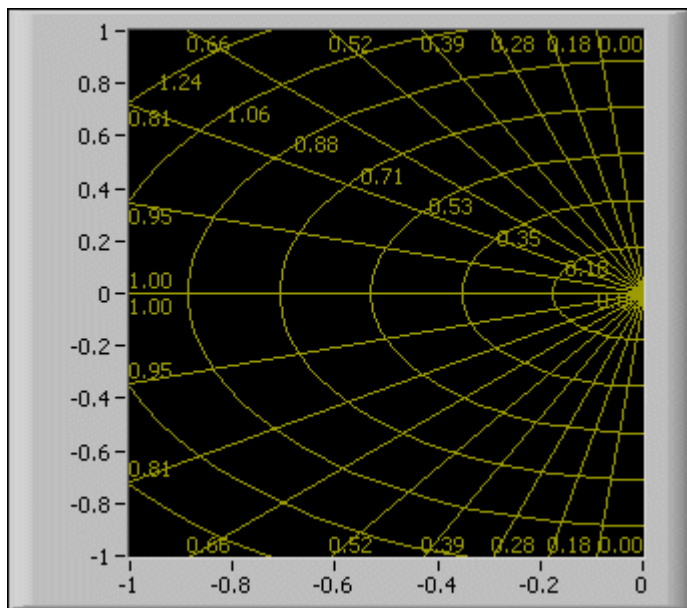
Displaying an S Plane

Complete the following steps to display an S plane.

1. Add an XY graph to the front panel.
2. Double-click the minimum and maximum values on the x-scale and change the minimum value to -1 and the maximum value to 0 .
3. Double-click the minimum and maximum values on the y-scale and change the minimum value to -1 and the maximum value to 1 .
4. Right-click the XY graph and select **Optional Plane»S Plane** from the shortcut menu.
5. (Optional) To view the S plane without labels, right-click the XY graph and remove the checkmark from the **Optional Plane»Show S Plane Labels** shortcut menu item.
6. (Optional) To view the S plane without the Cartesian lines, right-click the XY graph and remove the checkmark from the **Optional Plane»Show Cartesian Lines** shortcut menu item.

To remove the S plane, right-click the XY graph and select **Optional Plane»None** from the shortcut menu.

The following front panel shows an S plane without the Cartesian lines.



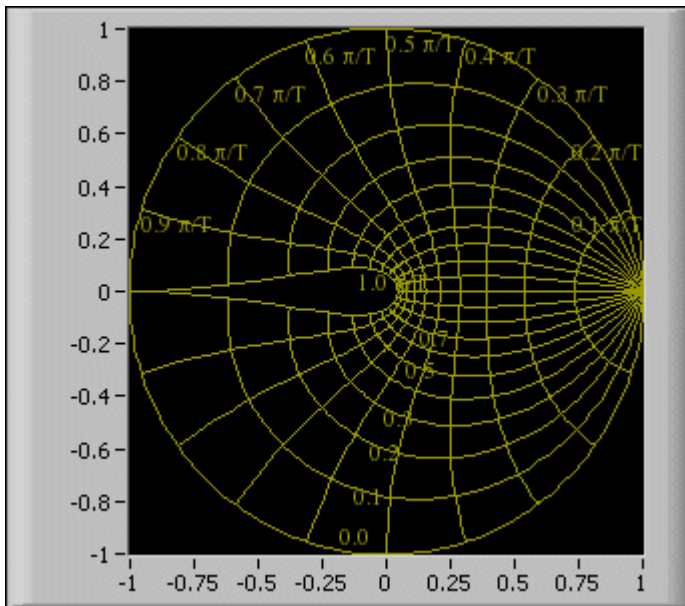
Displaying a Z Plane

Complete the following steps to display a Z plane.

1. Add an XY graph to the front panel.
2. Double-click the minimum and maximum values on the x- and y- scale and change both minimum values to -1 and both maximum values to 1.
3. Right-click the XY graph and select **Optional Plane»Z Plane** from the shortcut menu.
4. (Optional) To view the Z plane without labels, right-click the XY graph and remove the checkmark from the **Optional Plane»Show Z Plane Labels** shortcut menu item.
5. (Optional) To view the Z plane without the Cartesian lines, right-click the XY graph and remove the checkmark from the **Optional Plane»Show Cartesian Lines** shortcut menu item.

To remove the Z plane, right-click the XY graph and select **Optional Plane»None** from the shortcut menu.

The following front panel shows a Z plane without the Cartesian lines.



You also can use the Optional Plane:Plane Type property to display and hide these planes on the XY graph programmatically. Use the Optional Plane:Lines Visible, Optional Plane:Labels Visible, and Cartesian Lines:Visible properties to display and hide plane lines, plane labels, and Cartesian lines on the XY graph programmatically.

Graphing Digital Data

Complete the following steps to display a digital signal in a [digital waveform graph](#).

1. Acquire a digital signal or convert data into digital data.
2. [Add a digital waveform graph](#) to the front panel.
3. On the block diagram, wire the resulting digital data output or digital waveform output to the digital waveform graph terminal.
4. Run the VI.

Refer to the Create Digital Waveform VI in the `labview\examples\Controls and Indicators\Graphs and Charts\Digital Waveform Graph` directory for an example of graphing digital data.

Graphing Multiple Data Types on a Mixed Signal Graph

Graphing multiple data on a mixed signal graph is similar to graphing waveform data, XY data, and digital data on other graphs. You can wire any data type accepted by waveform, XY, or digital waveform graphs to the mixed signal graph.

Complete the following steps to graph multiple data types on a mixed signal graph.

1. Before wiring data directly to a graph, use the Bundle function to merge the data types into the mixed signal graph.
2. Wire the data to the elements of the Bundle function.
3. Wire the **output cluster** of the Bundle function to the mixed signal graph. LabVIEW automatically creates plot areas to accommodate combinations of analog and digital data. By default, the mixed signal graph displays digital data as digital lines and buses in the plot area.

Refer to the Mixed Signal Graph VI in the `labview\examples\Controls and Indicators\Graphs and Charts\Mixed Signal Graph` directory for an example of graphing multiple data types on a mixed signal graph.

Related information [Graphing Waveform Data](#) [Graphing XY Data](#) [Graphing Digital Data Types of Graphs and Charts](#) [Bundle Function](#)

Graphing Waveform Data

Complete the following steps to display waveform data on a single-plot [waveform graph](#).

1. [Add a waveform graph](#) to the front panel.
2. Build a block diagram with the following objects:
 - Divide function
 - Sine function
 - For Loop around all objects on the block diagram except the waveform graph terminal
3. Wire the **x/y** output of the Divide function to the **x** input of the Sine function.
4. Wire the **sin(x)** output of the Sine function to the waveform graph terminal. An [auto-indexing](#) tunnel appears where the wire intersects the loop border.
5. Wire the iteration terminal of the For Loop to the **x** input of the Divide function.
6. Right-click the **y** input terminal of the Divide function and select **Create»Constant** from the shortcut menu. A numeric constant appears.
7. Enter **10** in the constant and press the <Enter> key.
8. Right-click the count terminal of the For Loop and select **Create Constant** from the shortcut menu. A numeric constant appears.
9. Enter **100** in the constant and press the <Enter> key.
10. Run the VI.

Refer to the Waveform Graph Data Types VI in the `labview\examples\Controls and Indicators\Graphs and Charts\Waveform Graphs and Charts` directory for an example of graphing waveform data.

Graphing XY Data

Complete the following steps to display an array of points on a single-plot *XY graph*.

1. Add an *XY graph* to the front panel.
2. Build a block diagram with the following objects:
 - Divide function
 - Multiply function
 - Sine function
 - Bundle function
 - For Loop around all objects on the block diagram except the XY graph terminal
3. Wire the **x/y** output of the Divide function to the **x** input of the Sine function.
4. Wire the **sin(x)** output of the Sine function to the first (top) input terminal of the Bundle function.
5. Wire the **x*y** output of the Multiply function to the second (bottom) input terminal of the Bundle function.
6. Wire the iteration terminal of the For Loop to the **x** input of the Divide function and to the **x** input of the Multiply function.
7. Right-click the **y** input terminal of the Divide function and select **Create»Constant** from the shortcut menu. A numeric constant appears.
8. Enter **10** in the constant and press the <Enter> key.
9. Wire the constant to the **y** input terminal of the Multiply function.
10. Wire the output terminal of the Bundle function to the XY graph terminal. An *auto-indexing* tunnel appears where the wire intersects the loop border.
11. Right-click the count terminal of the For Loop and select **Create Constant** from the shortcut menu. A numeric constant appears.
12. Enter **100** in the constant and press the <Enter> key.
13. Run the VI.

Refer to the XY Graph Data Types VI in the `labview\examples\Controls and Indicators\Graphs and Charts\XY Graph` for an example of graphing XY data.

Ignoring Time Stamp and Plot Labels in Graphs

When you wire dynamic data or waveform data to a graph and run the VI, LabVIEW automatically formats the plot legend and x-scale time stamp for the graph based on the configuration of the dynamic or waveform data.

To ignore the plot legend label the dynamic or waveform data includes, right-click the graph and select **Ignore Attributes** from the shortcut menu. Selecting **Ignore Attributes** allows you to change the plot legend label programmatically.

To ignore the time stamp configuration the dynamic or waveform data includes, right-click the graph and select **Ignore Time Stamp** from the shortcut menu.



NOTE

If the data you display in the graph contains a time stamp formatted for absolute time, select **Ignore Time Stamp** to ignore the time stamp.

Mapping Graph Coordinates to Plot Image Coordinates

When you use the Plot Images properties, LabVIEW draws the image beginning at the origin of the graph plot area. If you want to use the Plot Images properties to draw an image that uses the graph data points as references rather than the plot area origin (0,0), you can use the Map XY To Coordinates method to map the graph coordinates of the data points to coordinates of the front panel. The Map XY To Coordinates method returns coordinates relative to the origin of the front panel. However, to draw the image correctly, you must make the coordinates relative to the origin of the plot image area located at the top, leftmost corner of the graph plot area. To account for the difference, you must subtract the offset of the plot area of the graph from the origin of the front panel from the coordinates that Map XY To Coordinates returns.

Complete the following steps to programmatically map graph coordinates to the coordinates of the owning pane.



NOTE

Using *splitter bars* creates multiple panes. You then account for the offset of the plot area of the graph from the origin of the owning pane.

1. Right-click the graph and select **Create»Invoke Node»Map XY To Coordinates** from the shortcut menu to create a Map XY To Coordinates Invoke Node.
2. Right-click the **XY** input of the Map XY to Coordinates method and select **Create»Constant** from the shortcut menu. Wire the graph coordinates of the data points to the **XY** input of the Map XY To Coordinates method.
3. Right-click the graph and select **Create»Property Node»Plot Area Bounds** from the shortcut menu to create a Plot Area Bounds Property Node.
4. Add the Unbundle By Name function to the block diagram.
5. Wire the **Plot Bounds** output of the Plot Area Bounds Property Node to the **input cluster** input of the Unbundle By Name function.
6. *Resize* the Unbundle By Name function so that both the **left** and **top** elements appear.
7. Add the Bundle function to the block diagram.
8. Wire the **left** and **top** elements of the Unbundle By Name function to the **element** inputs of the Bundle function.
9. Add the Subtract function to the block diagram.
10. Wire the **Coords** output of the Map XY To Coordinates Invoke Node to the **x** input of the Subtract function.
11. Wire the **output cluster** output of the Bundle function to the **y** input of the Subtract function.

12. Right-click the **x-y** output of the Subtract function and select **Create»Indicator** from the shortcut menu.

Moving Plots and Buses to Different Plot Areas in a Mixed Signal Graph

You can move plots and buses from one plot area to another in mixed signal graphs. For example, you might move a plot or a bus if you want to compare data more closely. Complete the following steps to perform this action.



NOTE

If a digital plot line is within a bus, you must move the bus to move the line.

1. In the tree control, select the icon next to the name of the plot or bus you want to move.
2. Drag the icon to another plot area in the plot legend.

You also can move a plot or bus in the mixed signal graph using the shortcut menu similar to how you can move plots in waveform graph, XY graph, and chart plot legends. Complete the following steps to move the plot or bus using the shortcut menu.

1. Right-click the plot or bus in the plot legend that you want to move.
2. Select **Y Scale** and the plot area to which you want to move the plot or bus from the shortcut menu.



Note LabVIEW dims the plot area in the shortcut menu if the data types do not match.

Related information [Adding Plot Areas to a Mixed Signal Graph](#)

Plotting Data on 2D Graphs

Use 2D graphs to plot data in two dimensions. When you add a 2D graph to the front panel, the graph appears on the block diagram with one of the following helper VIs:

- 2D Compass Plot Helper VI
- 2D Error Bar Plot Helper VI
- 2D Feather Plot Helper VI
- XY Plot Matrix Helper VI

Complete the following steps to plot data on a 2D graph.

1. Add a 2D graph to the front panel.
2. Select a polymorphic instance from the helper VI on the block diagram.
3. Configure the appearance of the graph with the 2D Graph properties. Use the Property Node to configure the appearance from the block diagram. Select **Properties** from the shortcut menu to configure the appearance with the Properties dialog box.
4. Wire data to each input to graph the plot.
5. Run the VI.



RELATED INFORMATION

[Types of Graphs and Charts](#) on page 477

[Manipulating Application and VI Settings](#) on page 261

Plotting Data on 3D Graphs

Use 3D graphs to plot data in three dimensions. When you add a 3D graph to the front panel, the graph appears on the block diagram with one of the following helper VIs:

- 3D Scatter Plot Helper VI
- 3D Stem Plot Helper VI
- 3D Comet Plot Helper VI
- 3D Surface Plot Helper VI
- 3D Contour Plot Helper VI
- 3D Mesh Plot Helper VI
- 3D Waterfall Plot Helper VI
- 3D Quiver Plot Helper VI
- 3D Ribbon Plot Helper VI
- 3D Bar Plot Helper VI
- 3D Pie Plot Helper VI

Complete the following steps to plot data on a 3D graph.



NOTE

The following steps do not apply to the 3D Surface Graph, 3D Parametric Graph, and 3D Line Graph.

1. Add a 3D graph to the front panel.



Note The 3D graphs are available only in the LabVIEW Full and Professional Development Systems.

1. Select a polymorphic instance from the helper VI on the block diagram.
2. Configure the appearance of the graph with the 3D Graph properties. Use the Property Node to configure the appearance from the block diagram. Select **3D Plot Properties** from the shortcut menu to configure the appearance with the 3D Plot Properties dialog box.
3. Wire data to each input to graph the plot.
4. Run the VI.



RELATED INFORMATION

[Types of Graphs and Charts](#) on page 477

[Plotting Data on a 3D Surface Graph](#) on page 1219

[Manipulating Application and VI Settings](#) on page 261

Plotting Data on a 3D Surface Graph

Complete the following steps to plot data on a [3D Surface Graph](#).

1. [Add a 3D Surface Graph](#) to the front panel window.



NOTE

The 3D graph controls are available only in the LabVIEW Full and Professional Development Systems.

1. On the block diagram, right-click the **x vector** input of the plot surface setup VI and select **Create»Control** from the shortcut menu to create a 1D array control.
2. Right-click the **y vector** input of the plot surface setup VI and select **Create»Control** from the shortcut menu to create a 1D array control.
3. Right-click the **z matrix** input of the plot surface setup VI and select **Create»Control** from the shortcut menu to create a 2D array control.
4. On the front panel, enter variant data into each of the array controls. Make sure the [array dimensions](#) match. The dimension size of the 2D array is the product of the 1D arrays.
5. Right-click the 3D graph control and select **3D Graph Properties** from the shortcut menu to display the 3D Graph Properties dialog box. Customize the appearance of the 3D graph using the options in the **3D Graph Properties** dialog box.
6. Run the VI. LabVIEW displays the data you input on the 3D graph surface.

You can right-click the 3D graph and select **Render Window** from the shortcut menu to view the 3D graph in a stand-alone window.

Plotting X, Y, and Z Data on an ActiveX 3D Surface Graph

Complete the following steps to plot x, y, and z data on an [ActiveX 3D surface graph](#).

1. [Add an ActiveX 3D surface graph](#) to the front panel.



NOTE

The 3D graph controls are available only in the LabVIEW Full and Professional Development Systems.

2. On the block diagram, delete the 3D Surface VI and the wire to the VI. This leaves only the ActiveX container terminal for the 3D graph.
3. Add an Invoke Node to the block diagram, and wire the 3D graph container terminal to the **reference** input of the Invoke Node.
4. Right-click the Invoke Node and select **Select Method»Plot3DMesh** from the shortcut menu. The Plot3DMesh method accepts [variant data](#) for arrays of x, y, and z data and plots triangulated surfaces.
5. Enter variant data to the **xVector**, **yVector**, and **zVector** inputs of the Plot3DMesh method. The values of **xVector** and **yVector** specify where the corresponding point in the **zVector** array is. The default values for **xVector** and **yVector** are 0, 1, 2, 3, and so on. The first point in **zVector** array (index 0,0) is located at (x vector[0], y vector[0]), or (0,0). If you change **xVector** to -1, 1, 2, 3, and so on, the first point in **zVector** moves to (-1,0).
6. (Optional) Wire magnitude data to the **wVector** input of the Plot3DMesh method to show the magnitude.
7. [Scale](#) the graph to ensure that [array dimensions](#) match. The dimension size of the 2D array is the product of the 1D arrays. For example, **xVector** and **yVector** correspond to the two indexes for the **zVector** 2D array. If the number of elements in **xVector** does not match the width of the **zVector** array in the first dimension, and the number of elements in **yVector** does not match the width of the **zVector** array in the second dimension, this method ignores the values of both **xVector** and **yVector**.
8. Run the VI.

Complete the following steps to show individual points instead of surfaces.

1. Right-click the front panel graph and select **CWGraph3D»Properties** from the shortcut menu.
2. Click the **Plots** tab and change the plot style to **Point**.
3. To avoid getting multiple colors for every point, select **None** from the **Color map style** pull-down menu.

Note You might have to change the default point color and size on the **Point** page for the data to be visible on the graph.

4. Click the **OK** button.
5. Run the VI.

You also can [plot x, y, and z data](#) on a 3D graph XControl.

Replaying the Animation for the 3D Comet Graph

Complete the following steps to replay the animation for the [3D Comet graph](#) after you run the VI.

1. Right-click the Comet graph.
2. Select **Play Animation** from the shortcut menu.



NOTE

The 3D graphs are available only in the LabVIEW Full and Professional Development Systems.

Refer to the 3D Comet Graph VI in the `labview\examples\Controls and Indicators\Graphs and Charts\Math Plots - 3D` directory for an example of using the Comet graph.

Charting Waveform Data

Complete the following steps to display waveform data on a single-plot, single-point waveform chart.

1. Add a [waveform chart](#) to the front panel.
2. Build a block diagram with the following objects:
 - Divide function
 - Sine function
 - While Loop —Drag the loop around all objects on the block diagram
3. Wire the **x/y** output of the Divide function to the **x** input of the Sine function, and wire the **sin(x)** output of the Sine function to the waveform chart terminal.
4. Wire the iteration terminal of the While Loop to the **x** input of the Divide function.
5. Right-click the **y** input terminal of the Divide function and select **Create»Constant** from the shortcut menu. A numeric constant appears.
6. Enter **10** in the constant and press the <Enter> key.
7. Wire a Boolean value to the conditional terminal of the While Loop. You also can right-click the terminal and select **Create Constant** or **Create Control** from the shortcut menu to create the Boolean object.
8. Run the VI.

Refer to the Waveform Chart Data Types and Update Modes VI in the `labview\examples\Controls and Indicators\Graphs and Charts\Waveform Graphs and Charts` directory for an example of charting waveform data.

Creating a Chart

A chart appends new data points to those points already in the display to create a history. On a chart, you can see the current reading or measurement in context with data previously acquired.

Complete the following steps to create a chart.

1. Add one of the following charts to the front panel window.
 - Waveform chart
 - Intensity chart
2. Right-click the chart and customize it using the shortcut menu items. You can configure the x- and y-scales, format the x- and y-scales, add a plot, zoom in or out, and scroll through the chart.
3. Build the block diagram to display data on a waveform chart.



TIP

You can export data and images from charts to locations such as Microsoft Excel. Right-click a chart and select **Export** from the shortcut menu to view available export options.



RELATED INFORMATION

[Graphs and Charts](#) on page 476

[Types of Graphs and Charts](#) on page 477

[Types of Graphs and Charts](#) on page 477

[Configuring the X- and Y-Axes on a Graph or Chart](#) on page 1196

[Changing the Format and Precision of Graph and Chart Axes](#) on page 1195

[Adding a Plot to a Plot Legend](#) on page 1193

[Zooming and Moving the View of a Graph or Chart](#) on page 1199

[Scrolling through a Graph or Chart](#) on page 1198

[Charting Waveform Data](#) on page 1221

[Exporting Data and Images](#) on page 1182

Changing the Numeric Representation of a Waveform

The waveform data **Y** of the waveform control can be an array of any numeric data type other than the fixed-point numeric data type. The interior of the waveform terminal appears orange if the data consists of floating-point or complex numbers and blue if the data consists of integers.

Complete the following steps to change the representation of the **Y** array of a waveform.

1. On the front panel or block diagram, right-click the waveform object.
2. Select **Y Array Representation** from the shortcut menu.
3. Select the *data type* that best represents the data.

Acquiring or Simulating a Signal

Use the DAQ Assistant Express VI to acquire a signal using a data acquisition device. Use the Simulate Signal Express VI to simulate a signal. After you acquire or simulate a signal, you can *graph*, *analyze*, or *write* the signal to a *text-based measurement file*.



NOTE

The DAQ Assistant Express VI is available on Windows only.

Different DAQ devices acquire different kinds of signals. In general, you can acquire two kinds of signals with a DAQ device—single-point, such as a temperature reading, and buffered, such as a waveform. You also can read data from a single channel or several channels.

Analyzing a Signal

Use the Signal Analysis Express VIs to analyze and measure any signal you acquire or simulate. For example, you can use the Spectral Measurements Express VI to take an FFT or power spectrum measurement of a signal.

Use the Arithmetic & Comparison Express VIs and the Numeric, Boolean, and Comparison functions to perform mathematical analysis of a signal. For example, you can use the Formula Express VI to analyze a signal based on a formula you enter.

Refer to the Context Help window for information about how each Express VI analyzes a signal or search the Functions palette for the correct Express VI for analyzing a signal.



RELATED INFORMATION

Acquiring or Simulating a Signal on page 1223

Finding an Object on the Palettes on page 904

Displaying a Signal

After you *acquire or simulate* a signal, you can display it in a graph, chart, or numeric indicator. Many Express VIs, such as the Simulate Signal, the DAQ Assistant, and Signal Analysis Express VIs, include a **Signals** output that you can wire to a graph, chart, or numeric indicator, such as a thermometer or digital display.

The indicator you choose depends on the type of signal. For example, if you acquire a single temperature, display the temperature in a thermometer or digital display. If you acquire several temperatures over a period of time, display those temperatures in a chart. If you acquire waveform signals, display those signals in a graph because waveforms, such as sine waves and sawtooth waves, include timing information and corresponding values.

You can configure the graph to automatically name the plot the same name as its signal type in the plot legend. For example, if you acquire or simulate a sine wave and wire the **Signals** output to a graph, the plot name for the signal appears as Sine in the plot legend.

Filtering a Signal

Use the Filter Express VI to filter a signal. You can display the signal in a graph, chart, or numeric indicator, send it to a hardware device, or write it to a file.



RELATED INFORMATION

[Displaying a Signal](#) on page 1223

[File I/O](#) on page 522

Merging Signals

Use the Merge Signals function to combine the data from two or more different acquired or simulated signals into one wire. You can wire the combined signals to a graph with multiple plots or save them to a file.

Complete the following steps to merge signals before you wire the signals to the input of another VI, function, graph, or chart.

1. Add the Merge Signals function to the block diagram.
2. Resize the Merge Signals function so it includes enough inputs for the signals you want to wire.
3. Wire each signal to an **input signal** input.
4. Wire the **combined signal** output to the **signals** input of another VI, function, graph, or chart.



NOTE

You can split any combined signals into the individual signals.



RELATED INFORMATION

[Resizing Nodes](#) on page 891

[Splitting Signals](#) on page 1225

Splitting Signals

Use the Split Signals function when you want to extract a single signal from a merged signal.

Complete the following steps to split a merged signal.

1. Add the Split Signals function to the block diagram.
2. To access more than one signal, resize the Split Signals function. For example, to split a merged signal into two signals, resize the function so two outputs appear. If you resize the function to include more outputs than are included in the merged signal, the extra outputs do not contain data.
3. Wire the merged signal to the **combined signal** input of the Split Signals function.



RELATED INFORMATION

[Merging Signals](#) on page 1224

[Resizing Nodes](#) on page 891

Instrument Driver Modification Instructions

Follow these instructions after creating an instrument driver using the Create New Instrument Driver Project wizard. Refer to the National Instruments Web site for more information about developing LabVIEW plug and play instrument drivers.

Modification Instructions

1. [Remove unsupported VIs](#)
2. [Modify common driver VIs](#)
3. [Modify instrument-specific VIs](#)
4. [Create new instrument-specific VIs](#)
5. [Create example VIs](#)
6. [Update the VI Tree VI](#)
7. [Update palette files](#)
8. [Submit the completed driver](#) to the Instrument Driver Network at ni.com/idnet.

Removing Unsupported VIs

1. Remove any instrument driver VIs in the project that the instrument does not support, including any common VIs such as Self-Test or Reset. You must identify the individual capabilities of your instrument to determine which VIs it supports. For example, the Digital Multimeter template includes VIs for making multipoint

measurements. If you write a driver for a Digital Multimeter that does not support multipoint measurements, delete these VIs from both the driver project and driver directory.

2. For each VI you removed, remove subVI calls to that VI from other instrument driver VIs in the project.
3. Save the project `.lvproj` and project library `.lvlib` after removing files.

Modifying Common Driver VIs

1. If you created the driver from a template, each common driver VI includes modification instructions in yellow text boxes on the front panel and/or block diagram. The common driver VIs include: Initialize, Close, Self-Test, Reset, Error Query, Revision Query, and Default Instrument Setup. Read the VI documentation, follow the modification instructions in each VI, and then delete the instructions.
2. Follow these additional instructions:
 - **Initialize VI**—Switch to the front panel and enter the default address information in the **VISA resource name** control. Refer to the **VISA resource name** control context help for the syntax for the device. Right-click the control and select **Data Operations»Make Current Value Default** from the shortcut menu. Update the context help of this control to show only information about supported buses for the instrument. Modify or remove serial controls in the **Serial Configuration** cluster to match the capabilities of the instrument. For example, if an instrument can only have 7 data bits and 1 stop bit, remove the **Data Bits** and **Stop Bits** controls and set these parameters as constants on the block diagram. If an instrument only supports 2 baud rates, remove unsupported rates from the **Baud Rate** control. Set the controls in the **Serial Configuration** cluster to the instrument defaults, right-click the controls, and select **Operate»Make Current Values Default** from the shortcut menu. You also can use the **Serial Configuration** cluster for other configuration settings such as a non-serial bus like TCP/IP. If necessary, update the cluster name and context help to reflect the controls within the cluster.
 - **Close VI**—If you need to configure the instrument to a certain state before it can be closed, add the necessary code to the block diagram before the VISA Close function.
 - **Self-Test VI**—Set the timeout constant on the block diagram to a value large enough for the instrument to finish its self-test. The default is 20,000 milliseconds, or 20 seconds.
 - **Error Query VI**—Designed to parse multiple SCPI errors. If the instrument implements error checking differently, refer to other instrument drivers, such as the Fluke 45 Multimeter, for alternatives to implement this functionality. The Error Query VI is included as a subVI in all driver VIs except low-level API VIs. For SCPI compliant instruments, use `:SYST:ERR?` for the error query command. For non-SCPI compliant instruments, parse the error query response to see if an error occurred. Based on the response, set the Boolean status element of the error cluster to the appropriate state. The Boolean should be False if there is no instrument error to report, and True otherwise.
 - **Revision Query VI**—Modify the Scan From String function to parse out the firmware query for the instrument.

3. Save the project (.lvproj) and project library (.lvlib) after modifying files.

Modifying Instrument-specific VIs

Complete the following steps for each remaining VI in the driver.

1. Open the VI and edit the front panel by adding, modifying, or removing controls and indicators as needed to perform the operation defined by the VI.
2. Update the default values of the controls to match the default values of the instrument. Set the value on each control and then right-click the control and select **Data Operations»Make Current Value Default** from the shortcut menu.
3. Update the control and indicator labels and descriptions to reflect any changes.
4. Edit the VI icon as needed.
5. If you removed or added controls and indicators, edit the connector pane to match.
6. Edit the block diagram by updating the command strings and parsed responses to match instrument functionality.
7. Edit the VI documentation as needed. An example of documentation you could enter for the Initialize VI is: This instrument driver supports serial communication. If you encounter problems when using serial communication, try using baud rates less than 19,200 and hardware handshaking.
8. If you created the driver from a template, each VI includes modification instructions in yellow text boxes on the front panel and/or block diagram. Read the VI documentation, follow the modification instructions in the VI, and then delete the instructions.
9. Save and test the VI.
10. Save the project .lvproj and project library .lvlib after modifying files.

Creating New Instrument-specific VIs

1. Use the Instrument Driver VI Wizard to create additional VIs that the instrument supports. If you created a driver from a class template, such as the Digital Multimeter template, the instrument probably supports additional functionality not covered by the class template. You should create additional VIs for each supported feature, using the existing VIs and structure for guidance. For example, oscilloscopes often have many types of triggers but the Oscilloscope template only includes support for edge triggering. If the instrument supports additional types of triggers, add this functionality with additional instrument driver VIs.
2. Use polymorphic VIs to group related instrument driver VIs. For example, multiple trigger configuration VIs can be presented with one polymorphic VI.
3. Save the project .lvproj and project library .lvlib after modifying files.

Below are some specific modifications for drivers created from the class templates.

- **Digital Multimeter** —The low-level VIs on the Configure and Data palettes allow more control of the DMM. They are not intended for use with the Read (Single Points) and Read (Multiple Points) template VIs, because these VIs already include portions of this functionality. Refer to the Read Triggered Multiple VI for an example of using these low-level VIs.
- **Oscilloscope** —The private Default Instrument Setup.vi should set the oscilloscope in single acquisition mode. This command may be similar to `:ACQ:STOPA SEQ;`.
- **Function Generator** —If the instrument supports arbitrary waveforms, you must create separate VIs to implement arbitrary waveform support. Do not add arbitrary waveform support to the Configure Standard Waveform.vi.

If the instrument automatically disables one mode when enabling another mode, you can implicitly enable the status of the modulation and sweep VIs. To do this, remove the **Enable <x> Mode** control and delete the False case from the block diagram.

Creating Example VIs

1. Create two or more example VIs that demonstrate how to use the instrument driver. Examples should help users understand the recommended calling sequence necessary to perform instrument operations. Include an example for each typical use case for the instrument to benefit a wide range of users.
2. Include example VIs in the instrument driver project, but not in the project library.
3. Select **Tools»Prepare Example VIs for NI Example Finder** to add an example data file `.bin3` to the `examples` directory in the instrument driver project.



NOTE

LabVIEW reserves the following prefixes and you cannot use them when you name the `.bin3` file: `ms`, `test`, `lv`, `daq`, `cvi`, `tree`, `fr`, `de`, `ja`, `vi`, `im`, `dsc`, `rt`, `_`, `ko`, `zh`, `it`, and `es`.

Updating the VI Tree VI

1. Open the VI Tree VI and update the block diagram to reflect changes to the instrument driver. The VI Tree is not a subVI in an application. It is intended as a quick overview of the hierarchy of the driver VIs.
2. Remove any VIs that are not supported by the instrument, including any common VIs such as Self-Test or Reset.
3. Add any public VIs that you added to the driver.
4. Save and close the VI.

Updating Palette Files

1. After all VIs are complete, update the function palette files `.mnu` to reflect changes to the instrument driver. All public instrument driver VIs should be accessible from the palette file. Do not include example VIs in the palette file.

2. Remove links to any VIs you removed from the driver.
3. Add palette files for subpalettes as needed to enhance the palette usability for new VIs. The palette file should match the layout of the VI Tree and the project folder hierarchy.

Submitting the Instrument Driver to the Instrument Driver Network

Whether you are an instrument manufacturer interested in making an instrument driver more accessible or you are an individual wanting to share your driver development efforts with others, consider submitting the driver to the Instrument Driver Network at ni.com/idnet.

Creating Instrument Driver Projects

Before you build an instrument driver, check that no driver exists for the instrument you are using. You can use the NI Instrument Driver Finder to find, download, and install LabVIEW Plug and Play drivers for an instrument. You also can check the manufacturer's Web site, and the Instrument Driver Network and NI Drivers and Updates on the National Instruments Web site for drivers. You also can [search for and install](#) existing LabVIEW instrument drivers.

Use the Create New Instrument Driver Project wizard to create a LabVIEW Plug and Play instrument driver project that conforms to the Instrument Driver Guidelines. The generated instrument driver project includes instrument driver VIs and function palette files .mnu.

Complete the following steps to create an instrument driver project using the Create New Instrument Driver Project wizard.

1. Select **Tools»Instrumentation>Create Instrument Driver Project** to display the wizard.
2. Select **New driver from template** or **New driver, copy existing driver** from the **Project type** pull-down menu, depending on whether you want to create the instrument driver project from a template driver or an existing project-style driver.



NOTE

If you want to create a new instrument driver project from an existing instrument driver, you must install the existing driver in the `labview\instr.lib` directory before you launch the Create New Instrument Driver Project wizard. You can search for and install existing instrument driver projects using the NI Instrument Driver Finder.

1. If you selected **New driver, copy existing driver** from the **Project type** menu, select an available instrument driver from the **Source driver** pull-down menu and click the **Next** button. If you selected **New driver from template** from the **Project type** menu, select a template type from the **Source driver** pull-down menu and click the **Next** button. The following table lists the templates that the Create New Instrument Driver Project wizard uses. Select a template that best matches the capabilities of your instrument. All the templates except the General Purpose

template create the core driver architecture with VIs and palette files .mnu for a particular type of instrument or device. The General Purpose template creates a basic driver that includes only the VIs and palette files common to most instrument drivers, regardless of instrument type.

Template	Description
	Controls basic operations such as outputting DC power and configuring the over-voltage and over-current protection. The template also includes features such as monitoring the output voltage and current.
	Controls basic operations such as setting the measurement function, range, and resolution. The template also includes advanced features such as configuring the trigger count and sample count and taking multipoint measurements.
	Controls basic operations such as outputting standard waveforms. The template also includes the ability to configure frequency and amplitude modulation.
	Controls basic operations such as acquiring waveforms using edge triggering and transferring waveform data from the instrument.
	Use for message-based instruments for which there is no class-specific template. Common message-based instruments include: GPIB, USBTMC, VXI-11, and serial.
	Use for register-based instruments for which there is no class-specific template. Common register-based instruments include: VXI and PXI.
	Controls basic operations such as setting the frequency range and sweep properties. The template also includes advanced features such as configuring and querying the marker.
	Controls basic operations such as configuring the measurement type and trigger level. This template also includes the ability to configure the start and stop arming conditions.
	Controls basic operations such as configuring the measurement function, range, and offset. This template also includes the ability to configure the averaging and correction frequency.

2. Enter a driver identifier for the new instrument driver project in the **Driver Identifier** text box, such as Agilent 34401 or Tektronix 2000 Series. The wizard uses the driver identifier you enter as the base name and path for the instrument driver project .lvproj and project library .lvlib. Include the instrument manufacturer name and model in the name to help users quickly identify a driver when viewing files on disk or within the LabVIEW environment. Each instrument driver VI includes the project library name as part of the VI name. Because the name is used to create a folder structure for the driver, and drivers can be used on a variety of platforms, the name cannot include any special characters, such as #, \$, and &. The name cannot conflict with other installed driver names, and the Create New Instrument Driver Project wizard checks for name conflicts.
3. Enter a description for the instrument driver project in the **Driver Description** text box and click the **Next** button. This description appears in the **Context Help** window if you move the cursor over the project library.

4. Create a VI icon banner by clicking the upper **Edit** button to open the Icon Editor and edit the top 8 rows of pixels of the VI icon. LabVIEW clears any edits made below the top 8 rows after you exit the Icon Editor. The wizard uses this banner as the icon banner for each driver VI in the instrument driver project. Include an abbreviated name for the driver in the banner to help users quickly identify a VI belonging to the driver.
5. Create a top-level palette icon by clicking the lower **Edit** button and editing the VI icon, and click the **Next** button. The wizard uses the top-level palette icon as the icon visible on the Instrument Driver palette. Design the icon to represent the driver and instrument type to help users quickly identify the driver on the palette.
6. Review the path to the installed driver in the **New Project Location** box and click the **Finish** button to create the instrument driver and open a new [Project Explorer](#) window with the newly created driver. The wizard generates all the required files for an instrument driver project, generates a warnings report if you have errors or warnings, and opens the [Instrument Driver Modification Instructions](#). Possible warnings include modifications in the new driver that differed from the existing driver. For example, if the existing driver included a VI that was not saved within the driver directory, the wizard creates a copy of the VI but places the VI within the new driver hierarchy.
7. Follow the Instrument Driver Modification Instructions to complete the driver. You can use the Instrument Driver VI Wizard to [create instrument driver VIs](#) for the project.

After you complete the instrument driver project, consider submitting the driver to the Instrument Driver Network at ni.com/idnet to make the instrument driver accessible to others.

Creating Instrument Driver VIs

You can use the Instrument Driver VI Wizard to create instrument driver VIs for an instrument driver project.

Complete the following steps to create an instrument driver VI using the Instrument Driver VI Wizard.

1. Create or open an instrument driver project. You can [create an instrument driver project](#) using the Create New Instrument Driver Project wizard. You can search for and install existing instrument driver projects using the NI Instrument Driver Finder.
2. Right-click a LabVIEW instrument driver project library `.lvlib` in the [Project Explorer](#) window and select **New»Instrument Driver VI** from the shortcut menu to display the Instrument Driver VI Wizard.
3. On the **Start** page of the Instrument Driver VI Wizard, select one of the following options and click the **Next** button.
 - **Create from an instrument driver template VI** —Creates the instrument driver VI from a template VI.
 - **Copy from an existing VI** —Copies and configures an existing instrument driver VI and its subVIs.

- **Configure your own VI** —Configures an instrument command, specifies how to parse a response, interacts with connected instruments, and generates the corresponding instrument I/O and string formatting/parsing code into a new instrument driver VI.
 - **Add a blank VI** —Creates an instrument driver VI that contains only a VISA resource name control and indicator and an error cluster control and indicator.
4. On the second page of the wizard:
- If you selected **Create from an instrument driver template VI** on the **Start** page, select a VI from the available instrument driver VI templates on the **Clone from Template** page and click the **Next** button.
 - If you selected **Copy from an existing VI** on the **Start** page, click the **Browse** button on the **Browse for VI** page to navigate to and select the VI you want to add to the project and click the **Next** button.
 - If you selected **Configure your own VI** on the **Start** page, select **Configuration VI** or **Measurement VI** on the **Select VI type** page and click the **Next** button. Select **Configuration VI** to create a VI that sends a command to the instrument. Select **Measurement VI** to create a VI that sends a command to the instrument and then reads a response from the instrument.
 - If you selected **Add a blank VI** on the **Start** page, continue to step 6.
5. If you selected **Configure your own VI** on the **Start** page, configure the **Control setup** page and click the **Next** button. Otherwise, continue to the next step.
1. Type a command string to send to the instrument in the **Command text with parameters in percents** text box. Specify parameters in the command string by placing % characters around the parameter name. LabVIEW displays the parameter names you create in the **Configure parameters** list. For every parameter you create, LabVIEW adds a corresponding control to the front panel and connector pane of the final VI.
 2. Select the appropriate data type for each parameter from the **Data type** pull-down menu.
 3. Set a default value for each parameter in the numeric box next to the **Data type** pull-down menu.
 4. Verify the configured command string in the **Command preview** text box.
 5. If you selected **Configuration VI** on the **Select VI type** page, select the instrument from the pull-down menu in the **Test the Command** section. If you selected **Measurement VI** on the **Select VI type** page, click the **Next** button and select the instrument from the pull-down menu in the **Test the Command** section.
 6. Click the **Run Test** button to test the command you configured. LabVIEW sends the command in the **Command preview** text box to the instrument and shows the number of bytes written to the instrument in the **Bytes written** box. If the command succeeds, the light turns a brighter green. If VISA returns an error, the light turns red. Click the **Explain Error** button to display the errors. If you selected **Measurement VI** on the **Select VI type** page, LabVIEW reads the response from the instrument and displays it in the **Response** text box. You can select the data type of the response in the **Configure response data type** section.

6. On the **VI Properties** page, edit the VI name, path, icon, and description and click the **Next** button.
7. On the **Summary** page, review the information about the new VI and click the **Finish** button to add the instrument driver VI and any subVIs to the instrument driver project and open the VI. If you created the VI from a template VI or an existing VI, LabVIEW renames subVIs that have the same name but a different connector pane as an existing VI in the project library. LabVIEW does not copy subVIs that have the same name and the same connector pane as an existing VI in the project library. The new VI uses the existing VIs in the project library in place of any subVIs that LabVIEW did not add.

Displaying Pixels Not Visible in a 2D Picture Control

If a picture is too large for the 2D picture control that displays it, LabVIEW crops the picture so you can see only the pixels that fit within the display region of the control. You can display the pixels that do not fit within the display region in the following ways:

- Use the Positioning tool to resize the control and run the VI again to view the entire picture. You also can use the Draw Area Size property to resize the control programmatically.
- Right-click the control and select **Visible Items»Scrollbar** from the shortcut menu to display scroll bars for the control. You also can use the Scrollbar Visible property to display scroll bars for the control programmatically.
- Use the Origin property to change the horizontal and vertical coordinates of the origin in the control so a different portion of the picture is visible in the control.
- Use the Zoom Factor property to scale the image in the control.

Resetting 2D Picture Controls

By default, the 2D picture control clears its display before drawing an image. If you want the 2D picture control to draw the new image over the previous image, right-click the control and select **Erase First** from the shortcut menu to remove the checkmark from this menu item.

You also can use the Erase First property to specify whether the 2D picture control erases the previous image programmatically.

Using Sound Input and Sound Output Devices

Complete the following steps if you are using sound input and sound output devices instead of accessing a sound file on a computer.

1. Use the Sound Input Configure and Sound Output Configure VIs to configure a sound device for input or output operations. When you configure a sound device for sound input or sound output, LabVIEW creates a task ID. A task ID is an identification number associated with the configuration on the specified device.

2. To manipulate the sound coming from or going to the device, pass the task ID to the various Sound VIs. Manipulate the data you receive from or send to the device by using the Sound Input VIs or the Sound Output VIs.
3. Use the Sound Input Clear or Sound Output Clear VIs to close the device.

Creating a 3D Scene with the 3D Picture Control

Complete the following steps to create a 3D scene using the 3D Picture Control VIs and the SceneObject methods.

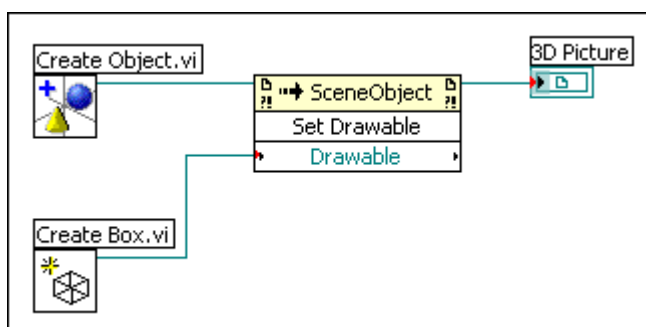
1. Add a 3D picture control to the front panel window.
2. Add the Create Object VI to the block diagram to the left of the 3D picture control terminal.
3. Right-click the **Scene:New Object** output of the Create Object VI and select **Create»Method for SceneObject Class»Set Drawable** from the shortcut menu to create the Set Drawable method.
4. Add the Set Drawable method to the block diagram to the right of the Create Object VI.
5. Wire the **Scene:New Object** output of the Create Object VI to the **reference** input of the Set Drawable method.
6. Select any VI from the Geometries palette and add it to the block diagram to the left of the Set Drawable method.



NOTE

If you use the Create Mesh VI, you must use the Set Mesh Parameters method to configure the mesh.

1. Wire the reference output of the VI you added in step 5 to the **Drawable** input of the Set Drawable method.
2. Wire the reference output of the Set Drawable method to the 3D picture control terminal. The following block diagram shows a simple 3D scene.



3. Wire error in and error out terminals on the block diagram.
4. Run the VI to generate the 3D scene.
5. (Optional) Configure a camera to view the 3D scene from different perspectives.

After you create a basic 3D scene, you can use other 3D Picture Control VIs to enhance the 3D scene. You can add additional 3D objects and create object relationships, apply textures to 3D objects, and configure light sources for the 3D scene.

You also can configure a scene window to display and render the 3D scene.

Refer to the 3D Model of Solar System VI in the labview\examples\Graphics and Sound\3D Picture Control directory for an example of a 3D scene.



RELATED INFORMATION

[Using Meshes to Draw 3D Objects](#) on page 516

[Setting up Error I/O in a VI](#) on page 1436

[Configuring a Camera for a 3D Scene](#) on page 1235

[3D Picture Control VIs](#) on page 239

[Creating Relationships between 3D Objects](#) on page 1241

[Changing the Surface Appearance of a 3D Object](#) on page 1238

[Configuring a Light Source for a 3D Scene](#) on page 1236

[Configuring a 3D Scene Window](#) on page 1244

Configuring a Camera for a 3D Scene

After you [create a 3D scene](#), you can configure how a user-controlled camera interacts with the 3D scene.



NOTE

The following procedure explains how to configure a camera for a 3D scene rendered in a 3D picture control. If you [configure a scene window](#) for viewing the 3D scene, you must use the SceneWindow properties to configure the camera for the scene window programmatically.

Refer to the 3D Model of Solar System VI in the labview\examples\Graphics and Sound\3D Picture Control directory for an example of configuring a camera for a 3D scene window.

Complete the following steps to configure a camera for a 3D scene rendered in a 3D picture control.

1. [Create a 3D scene](#) or open a VI that contains a 3D scene.
2. On the front panel, right-click the 3D picture control and move the cursor over **Camera Controller** in the shortcut menu to display the available options.
3. Select among the following camera controller options:
 - **None** —Disables the camera for the 3D scene.
 - **Spherical** —Configures the camera to rotate around a fixed point. The fixed point is the point in the 3D scene at the center of the 3D picture control.

- **Flying** —Configures the camera to fly freely through the 3D scene. If you select this option, be careful not to move the camera so far from the 3D objects that you no longer can see any of the objects.
- **Oriented** —Configures the camera to rotate around a fixed point relative to the z-axis. If you select this option, LabVIEW prevents the camera from tilting to the right or left.

When you move the camera in the 3D scene, LabVIEW redraws the scene to update the view you see in the 3D picture control.

4. (Optional) To disable automatic redrawing, right-click the 3D picture control and select **Camera Controller»Auto Redraw** from the shortcut menu to remove the checkmark next to the option. Disabling automatic redrawing is useful when you make multiple updates to a 3D scene and do not want the view of the 3D scene to change until you complete all the updates.



NOTE

If you disable automatic redrawing, you can use a loop, such as a For Loop or While Loop, to update the 3D scene. Enclose the VIs, properties, and methods that compose the 3D scene in the loop. LabVIEW redraws the 3D scene with each iteration of the loop, which updates the view to match the current camera perspective.

1. Run the VI to generate the 3D scene.
2. Use the Operating tool to click and drag in the 3D picture control and move the camera around the 3D scene. You can zoom in and out of the 3D scene by holding the <Shift> key while using the Operating tool to click and drag. You can reposition the camera in the plane of the screen by holding the <Ctrl> key while using the Operating tool to click and drag. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.

You also can use the Camera Controller:Type and Camera Controller:Auto Redraw properties to configure camera controller options for a 3D scene rendered in a 3D picture control programmatically.

Configuring a Light Source for a 3D Scene

You can configure a light source to achieve visual effects such as sunlight or spotlights in a visible 3D scene. You can create three types of lights in a 3D scene: directional lights, positional lights, and spotlights. While the following procedure explains how to configure a directional light for a 3D scene, you can use similar steps to configure positional lights and spotlights.

Complete the following steps to configure a directional light for a 3D scene.

1. [Create a 3D scene](#), or open a VI that contains a 3D scene.
2. Remove the wire that connects the 3D object to the 3D picture control terminal.

3. Right-click the reference output of the Set Drawable method and select **Create»Method for SceneObject Class»Light»Add Light** from the shortcut menu to create the Light:Add Light method.
4. Add the Light:Add Light method to the right of the Set Drawable method, and wire the reference output of the Set Drawable method to the reference input of the Light:Add Light method.
5. Add the New Light VI to the block diagram.
6. Right-click the **Light Number** input of the New Light VI, select **Create»Constant** from the shortcut menu, and enter a number between 0 and 7, inclusive. This indexes, or sets the integer that identifies, the light. You cannot index more than 8 lights.

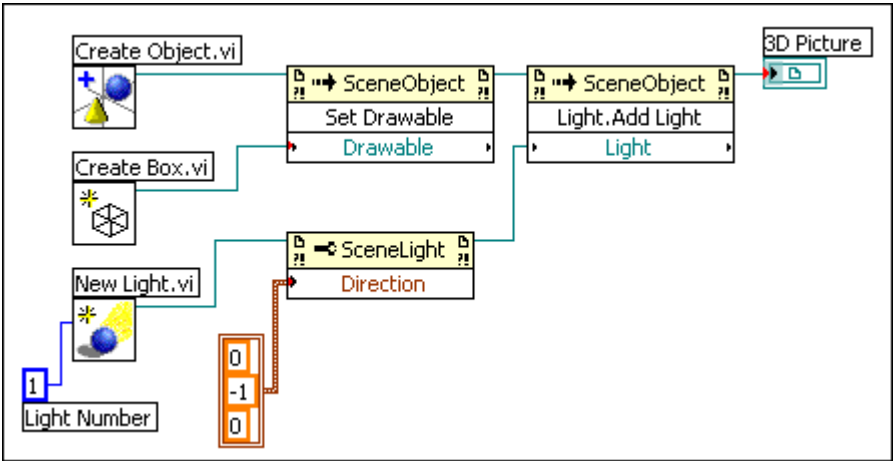


NOTE

LabVIEW cannot generate a light source if the index is undefined. By default, LabVIEW defines the light at index 0 as a directional light. If you configure a light at index 0, the light you configure overwrites the default light.

1. Right-click the **New Light Refnum** output of the New Light VI and select **Create»Property for SceneLight Class»Direction** from the shortcut menu to create the Direction property.
2. Add the Direction property to the right of the New Light VI, right-click the property, and select **Change to Write** from the shortcut menu.
3. Wire the **New Light Refnum** output of the New Light VI to the reference input of the Direction property.
4. Right-click the **Direction** input of the Direction property, select **Create»Constant** from the shortcut menu, and change the values of the constant to the x, y, and z coordinates that you want to use to define a direction for the light.
5. Wire the reference output of the Direction property to the **Light** input of the Light:Add Light method.
6. Wire the reference output of the Light:Add Light method to the 3D picture control terminal.

The following block diagram shows a 3D scene with a configured directional light. The direction coordinates in this 3D scene add the light source beneath the 3D object.



- 1. Wire error in and error out terminals on the block diagram.
- 2. Run the VI to generate the 3D scene.

Refer to the Using Meshes VI in the labview\examples\Graphics and Sound\3D Picture Control directory for an example of configuring a light source for a 3D scene.

Changing the Surface Appearance of a 3D Object

By default, new 3D objects have solid, single-colored surfaces. To change the appearance, you must apply a **texture** to the 3D object. A texture is a 2D image that you wrap around a 3D object.

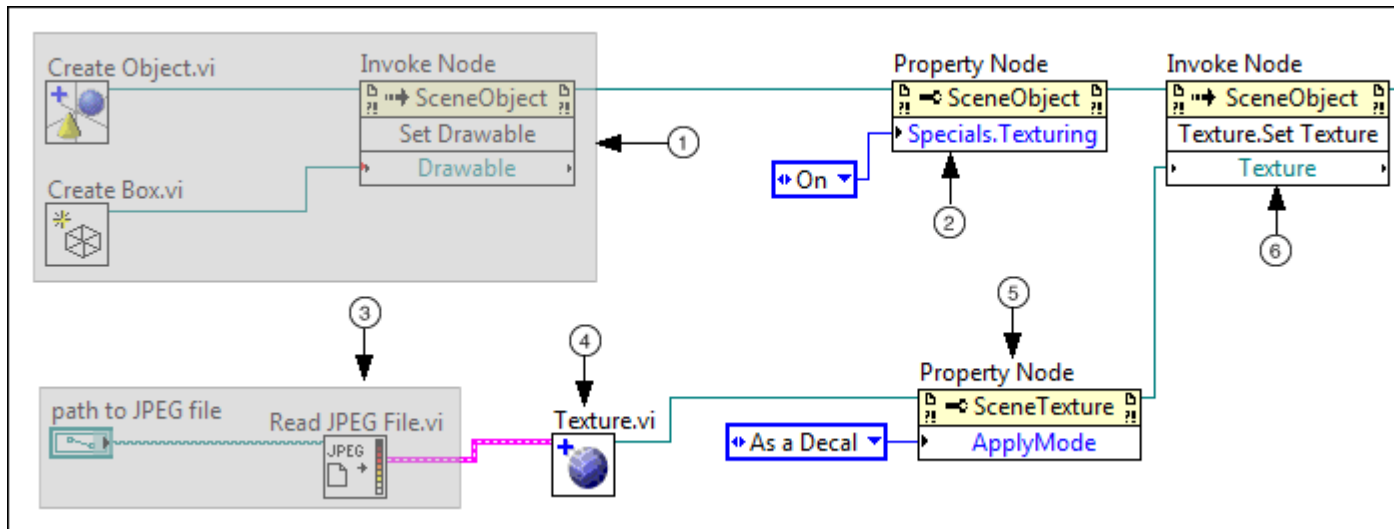
What to Use

Use the following block diagram objects to change the surface appearance of a 3D object:

Reference to 3D Object	Specials:Texturing Property	Texture VI	Apply Mode Property	Texture:Set Texture Method	3D Picture Control
Create single 3D object					

What to Do

Create the following block diagram to programmatically change the appearance of a 3D object. Customize the gray sections for your unique programming goals.



The following list describes important details about the previous diagram:

①	Obtain a reference to the 3D object whose appearance you want to change. Your programming goals determine how you obtain this reference. The most common way to obtain a 3D object reference is to <i>create a new 3D object</i> . This example obtains a 3D object reference by creating a new 3D box.
②	The Specials:Texturing property enables a 3D object to have a texture. If you do not set the Specials:Texturing property to On, you cannot apply texture to a 3D object. If the object to which you want to apply the texture is a <i>child of another object</i> , you can set the Specials:Texturing property to Inherit Value to inherit the texture settings of the parent object.
③	Obtain LabVIEW image data that defines the texture you want to appear. Use one of the following ways to obtain LabVIEW image data: - To convert a standard image format such as JPG to LabVIEW image data, use the appropriate Read VI from the Graphics Formats palette. - To access an image that is part of a pre-existing 3D object, use a 3D picture control property or method. - To define a custom image, create an image cluster. This example uses the Read JPEG File VI to convert a JPG file to LabVIEW image data.
④	The Texture VI converts the LabVIEW image data to a SceneTexture reference. You must convert image data to a SceneTexture reference so that you can interact with SceneTexture properties and methods to change the way that the texture appears on a 3D object.
⑤	The Apply Mode property specifies how the texture appears on a 3D object. If you do not set the apply mode for the texture, LabVIEW applies the texture with blending, where the image data blends with the background color of the object. If you want the texture to retain the coloration of the original 2D image, set Apply Mode to As a Decal or As a replacement. This example uses the Apply Mode property of the texture to specify that the texture image appears with its original colors as a decal.

6	The Texture:Set Texture method assigns a texture to the referenced 3D object.
7	Use the SceneObject reference to display or manipulate the new 3D object. You can use this reference to achieve the following goals: - To display the 3D object—Create a 3D picture control from the front panel and wire the reference to the associated block diagram terminal. - To control the size and position of the 3D object—Wire the reference to one of the Transformation VIs. - To add the object to multi-object 3D scene— Use the reference to <i>create a 3D scene that contains multiple 3D objects</i>. - To interact with other attributes of the 3D object— Wire the reference to a SceneObject property or method. These properties and methods control attributes and behaviors of the 3D object, such as its color, <i>light source</i>, and texture. This example displays the textured 3D object by using a 3D picture control.

Tips and Troubleshooting Information

- You can create more complex textures.** Use the SceneTexture properties of the SceneTexture reference to configure other texture settings, such as the filter mode, horizontal or vertical wrap mode, and transformations to apply to the texture.
- Does the texture unexpectedly appear on multiple 3D objects in the scene?** You may have applied the texture to the parent object in a *hierarchy of 3D objects*. Changes to a parent 3D object apply to all children of that object. To fix this problem, apply the texture to the child object reference rather than to the parent object reference.
- Does the texture not appear at all?** The Specials:Texturing property must resolve to the On value, whether because you directly set the property to On or because it inherits the On value from its parent. Verify that you set the Special.Texturing property to an appropriate value.
- Does the 3D picture control appear as a refnum on the front panel?** You must create the 3D picture control from the front panel, not from a terminal on the block diagram. After you create the 3D picture control from the front panel, wire the associated block diagram terminal to a SceneObject reference to display the referenced object.
- Can you not interact with the 3D picture control on the front panel?** You must *configure a camera* to view a 3D scene from different perspectives.
- Always wire error in and error out terminals.** The example in this topic omits error handling to emphasize programming logic, but error handling enables debugging and controls data flow.

Examples

Refer to the 3D Model of Solar System VI in the labview\examples\Graphics and Sound\3D Picture Control directory for an example of a 3D scene that contains objects with textures.

Related Topics

- [Create a 3D scene with one object](#)
- [Create a 3D scene with multiple objects](#)
- [Configure a camera to view the objects](#)
- [Configure a light source](#)
- [Add text](#)
- [Create a separate Scene Window](#)

Creating Relationships between 3D Objects

To [create a 3D scene](#) with more than one 3D object, you must set up a hierarchy of 3D object relationships. The 3D object that you use to wire the reference for all objects to the 3D picture control terminal or, if you [configure a scene window](#), the Scene property, is the parent of all other objects in the 3D scene. 3D objects that you add to the scene by wiring a reference to the object to a higher object in the hierarchy are child 3D objects.

If a 3D object is the parent of another 3D object, any transformations you apply to the object also affect the children of that 3D object. Child 3D objects can also be parents of children beneath them in the relationship hierarchy. You can use object relationships in large 3D scenes to program more efficiently and for 3D scenes that involve related movement.

Complete the following steps to create a 3D scene with multiple objects.

1. [Create a 3D scene](#), or open a VI that contains a 3D scene.
2. Remove the wire that connects the 3D object to the 3D picture control terminal.
3. Right-click the reference output of the Set Drawable method and select **Create»Method for SceneObject Class»Object»Add Object** from the shortcut menu to create the Object:Add Object method.
4. Add the Object:Add Object method to the right of the Set Drawable method and wire the reference output of the Set Drawable method to the reference input of the Object:Add Object method.
5. Wire the reference output of the Object:Add Object method to the 3D picture control terminal. The 3D object becomes the parent 3D object.
6. Add another Create Object VI to the block diagram to the left of the Object:Add Object method.
7. Right-click the **Scene:New Object** output of the Create Object VI and select **Create»Method for SceneObject Class»Set Drawable** from the shortcut menu to create another Set Drawable method.
8. Add the new Set Drawable method between the Create Object VI and the Object:Add Object method, and wire the **Scene:New Object** output of the Create Object VI to the reference input of the Set Drawable method.
9. Select any VI from the Geometries palette and add it to the block diagram to the left of the Set Drawable method.

10. Wire the **New xxx Refnum** output of the Geometries VI you selected to the **Drawable** input of the Set Drawable method.
11. (Optional) Translate the new 3D object to another location in the 3D scene. Add the Translate Object VI to the block diagram.
12. Wire the reference output of the new Set Drawable method to the **Scene.Object In** input of the Translate Object VI
13. Right-click the **Translation** input of the Translate Object VI, select **Create»Constant** from the shortcut menu, and change the values of the constant to the x, y, and z coordinates that you want to use to translate the child object.
14. Wire the **Scene.Object Out** output of the Translate Object VI to the **Object** input of the Object:Add Object method.

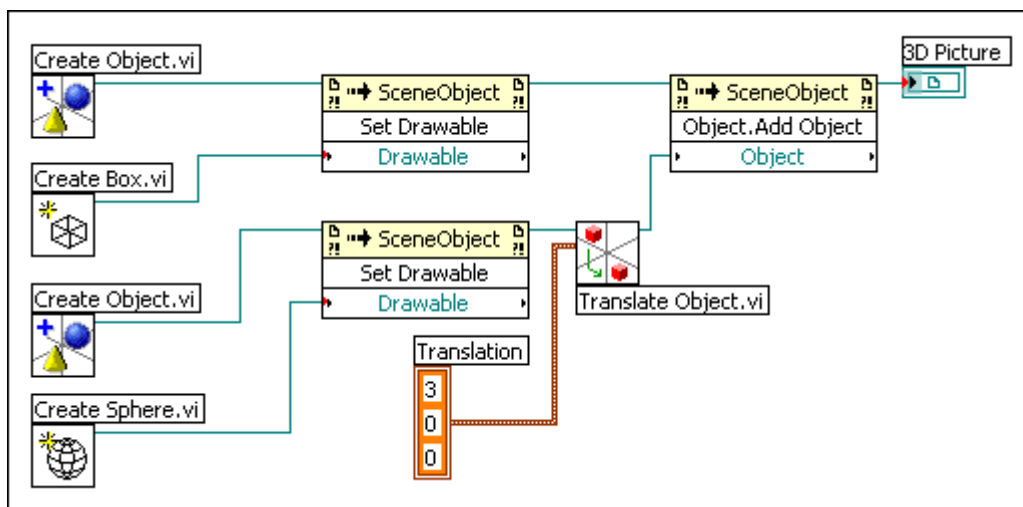


NOTE

If you do not translate additional objects that you add to a 3D scene, the 3D objects that you do not translate appear on top of each other, and may be difficult to see.

1. If you did not complete step 11, wire the reference output of the new Set Drawable method to the **Object** input of the Object:Add Object method.

The following block diagram shows a 3D scene with two 3D objects.



1. **Wire error in and error out terminals** on the block diagram.
2. Run the VI to generate the 3D scene.

Refer to the 3D Model of Solar System VI in the labview\examples\Graphics and Sound\3D Picture Control directory for an example of a 3D scene with multiple object relationships.

Adding Text to a 3D Scene

You can add text to a **3D scene** that appears when you generate the scene by creating a text object. Complete the following steps to add text to a 3D scene.

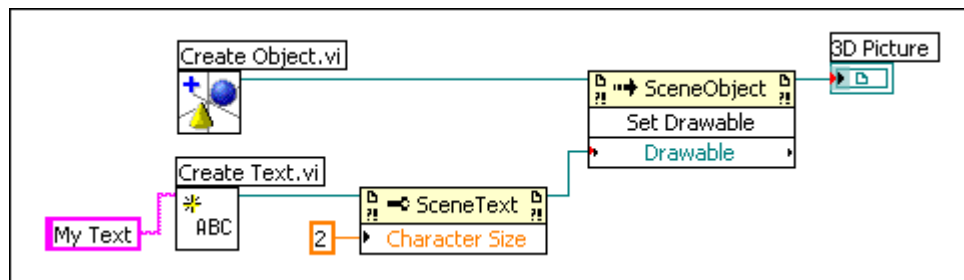
1. Add a 3D picture control to the front panel.
2. Add the Create Object VI to the block diagram to the left of the 3D picture control terminal.
3. Right-click the **Scene:New Object** output of the Create Object VI and select **Create»Method for SceneObject Class»Set Drawable** from the shortcut menu to add the Set Drawable method to the block diagram.



NOTE

You also can *use the Class Browser window* to browse for and create *properties and methods*.

1. Wire the **Scene:New Object** output of the Create Object VI to the reference input of the Set Drawable method.
2. Add the Create Text VI to the block diagram to create a new text object.
3. Right-click the **Text** input of the Create Text VI and select **Create»Constant** from the shortcut menu to create a new constant.
4. Enter the text you want to appear in the 3D scene in the constant you created in the previous step.
5. Right-click the **New Text Refnum** output of the Create Text VI and select **Create»Property for SceneText Class»Character Size** from the shortcut menu to add the Character Size property to the block diagram.
6. Wire the **New Text Refnum** output of the Create Text VI to the reference input of the Character Size property.
7. Right-click the Character Size property and select **Change To Write** from the shortcut menu.
8. Right-click the **Character Size** input of the Character Size property and select **Create»Constant** from the shortcut menu to create a new constant.
9. Enter a value for the character size in the constant you created in the previous step.
10. (Optional) *Expand the Character Size property node* to configure additional SceneText properties.
11. Wire the reference output of the Character Size property to the **Drawable** input of the Set Drawable method.
12. Wire the reference output of the Set Drawable method to the 3D picture control terminal. The following block diagram shows a 3D scene that contains the text My Text. The text is set to a character size of 2.



1. (Optional) For more configuration options, delete the wire that connects the Set Drawable method to the 3D picture control terminal and wire the reference output of the Set Drawable method to any SceneObject property or method.



Note You cannot apply *texture* to a text object.

1. *Wire error in and error out terminals* on the block diagram.
2. Run the VI to generate the 3D scene.

Configuring a 3D Scene Window

After you *create a 3D scene* you can configure a separate scene window in which to render the 3D scene.

Complete the following steps to configure a 3D scene window.

1. *Create a 3D scene*, or open a VI that contains a 3D scene.
2. Delete the 3D picture control terminal and remove any broken wires.
3. Add the Setup Window VI to the block diagram.
4. Right-click the **Window State** input of the Setup Window VI and select **Create»Constant** from the shortcut menu. Set the value of the constant to Open, Maximized, or Minimized.

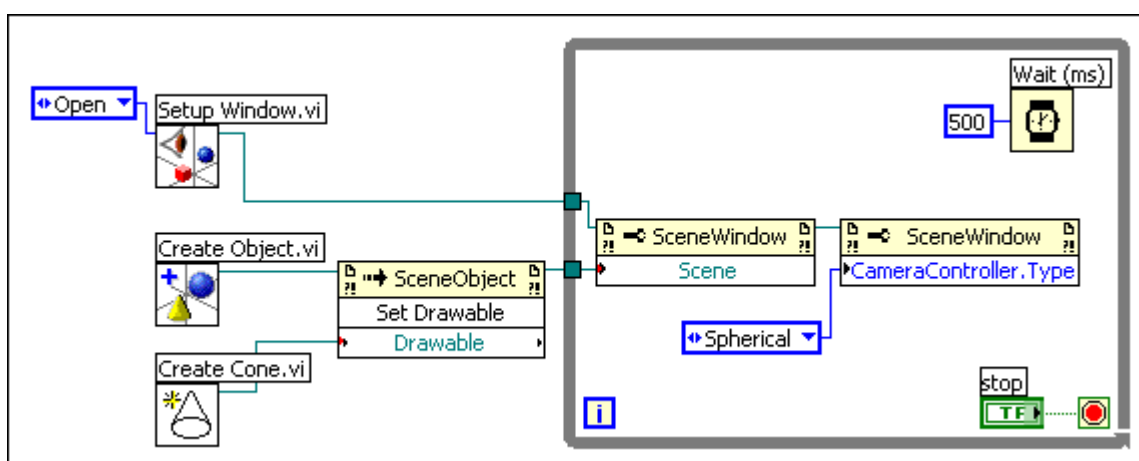


NOTE

If you set **Window State** to Hidden, you cannot see the scene window when you run the VI.

1. Add a While Loop to the block diagram to the right of the existing block diagram objects.
2. Right-click the *conditional terminal* of the While Loop and select **Create Control** from the shortcut menu to create a Boolean control. This control functions as the **Stop** button for the VI.

3. Right-click the **New Window Refnum** output of the Setup Window VI and select **Create»Property for SceneWindow Class»Scene** from the shortcut menu to create the Scene property.
4. Add the Scene property inside the While Loop, right-click the property, and select **Change to Write** from the shortcut menu.
5. Wire the reference output of the Set Drawable method from outside the While Loop to the **Scene** input of the Scene property.
6. Wire the **New Window Refnum** output of the Setup Window VI from outside the While Loop to the reference input of the Scene property.
7. (Optional) Configure a camera for the scene.
 1. Right-click the reference output of the Scene property and select **Create»Property for SceneWindow Class»Camera Controller»Type** from the shortcut menu to create the Camera Controller:Type property.
 2. Add the Camera Controller:Type property inside the While Loop to the right of the Scene property, right-click the property, and select **Change to Write** from the shortcut menu.
 3. Wire the reference output of the Scene property to the reference input of the Camera Controller:Type property.
 4. Right-click the **Camera Controller:Type** input of the Camera Controller:Type property, select **Create»Constant** from the shortcut menu, and set the value of the constant to Spherical or Flying.
8. (Optional) Add the Wait (ms) function to the While Loop to provide smoother camera movements in the scene window. If you do not add a wait to the While Loop, the 3D scene may appear to jump around when you move the camera.
 1. Right-click the **milliseconds to wait** input of the Wait (ms) function, select **Create»Constant** from the shortcut menu, and set the value of the constant to an integer of 1 or greater. The following block diagram shows a configured scene window with a spherical camera controller and a Wait (ms) function in the While Loop.



1. *Wire error in and error out terminals* on the block diagram.
2. Run the VI to generate the 3D scene in the scene window.

This procedure gives a basic overview of how to set up a scene window. Use the SceneWindow properties and methods to set additional configuration settings for a scene window, such as window bounds, automatic projection mode or projection matrix programmatically.

Refer to the 3D Model of Solar System VI in the labview\examples\Graphics and Sound\3D Picture Control directory for an example of a 3D scene rendered in a scene window.

Changing the Mechanical Action of a Boolean Object

Boolean controls have six types of mechanical action that allow you to customize Boolean objects to create front panel windows that more closely resemble the behavior of physical instruments.



NOTE

When you assign a keyboard shortcut to the mechanical action of a Boolean control, the keyboard toggles the value. For example, when you assign a keyboard shortcut to a Boolean control and set the keyboard shortcut to one of the latching mechanical actions, the button unlatches when read.

Complete the following steps to change the mechanical action of a Boolean control.

1. Add a Boolean control, such as a push button, to the front panel window.
2. Right-click the Boolean control and select **Properties** from the shortcut menu to display the **Boolean Properties** dialog box.
3. Click the **Operation** tab of the **Boolean Properties** dialog box.
4. Select a **button behavior** appropriate to the application from the **Button behavior** list. An explanation of the behavior appears in the **Behavior Explanation** section, and you can test the selected behavior in the **Preview Selected Behavior** section. In the icons that appear in the **Behavior Explanation** section, M represents the motion of the mouse button when you operate the control, V represents the output value of the control, and RD represents the point in time the VI reads the control.



NOTE

You cannot use any latch action for objects with a local variable. If you configure your Boolean value with a latching mechanical action, the Value property and the Value (Signaling) property return an error when the VI is in a non-idle state, such as when the VI is running or reserved to run. Due to race conditions that can occur when you have a Boolean value with latching mechanical action, you cannot programmatically read Boolean values that are set with a latching mechanical action.

Refer to the Mechanical Action VI in the labview\examples\Controls and Indicators\Boolean directory for an example of Boolean control mechanical actions.

Configuring a Mixed Checkbox

Use a mixed checkbox to display a TRUE, FALSE, or MIXED value. Complete the following steps to create and configure a mixed checkbox.

1. Add a mixed checkbox to the front panel window.
2. Right-click the control and select **Properties** from the shortcut menu to display the **Mixed Checkbox Properties** dialog box. On the **Appearance** page, you can configure the enabled state; strings that LabVIEW displays when the value of the control is on, off, or mixed; and so on.
3. Right-click the control and select **Allow Mixed** from the shortcut menu to disable the ability to set the MIXED value of a mixed checkbox interactively. When you remove the checkmark from the **Allow Mixed** shortcut menu item, LabVIEW can assign the value of the mixed checkbox as MIXED, but you cannot operate to MIXED by clicking on it.

You also can use the Allow Mixed property to disable the MIXED value of a mixed checkbox programmatically.

Refer to the System Mixed Checkbox VI in the `labview\examples\Controls and Indicators\Boolean` directory for an example that demonstrates the MIXED value of a mixed checkbox.

Creating a Radio Buttons Control

Complete the following steps to create a radio buttons control.

1. Add a radio buttons control to the front panel window.
2. (Optional) Add additional Boolean controls to the radio buttons control. You also can [clone](#) an existing Boolean control in the radio buttons control.
3. (Optional) Configure the radio buttons control by right-clicking the control and selecting from the following items in the shortcut menu:
 - **Allow No Selection** —Gives users the option of selecting none or one of the Boolean controls in the radio buttons control.
 - **Mechanical Action** —Sets the [mechanical action](#) for the radio buttons control to Switch When Pressed or Switch When Released. You cannot set the mechanical action for individual Boolean controls in the radio buttons control.
4. Use the Labeling tool to [edit the label](#) of each Boolean control in the radio buttons control. Each Boolean control must have a unique label. If you selected the **Allow No Selection** option and you create a label of No Selection for a Boolean control in the radio buttons control, the VI returns a broken **Run** button.
5. [Create a Case structure](#) to handle the item the user selected in the radio buttons control. A radio buttons control appears on the block diagram as an enumerated

type. If no item is selected, the numeric value of the radio buttons control is 0, and the string value is `No Selection`.

You also can use the Allow No Selection property to give users the option not to select any Boolean control in the radio buttons control programmatically.

Refer to the following VIs for examples of using a radio buttons control:

- Radio Buttons VI: `labview\examples\Controls and Indicators\Boolean`

Creating and Operating Boolean Controls and Indicators

Complete the following steps to add a Boolean control or indicator to the front panel.

1. Add a Boolean control or indicator, such as a push button, to the front panel.
2. (Optional) Type an item name. Click anywhere in the front panel window or press the <Ctrl-Enter> keys to end the editing session. (macOS) Press the <Option-Enter> keys. (Linux) Press the <Alt-Enter> keys.
3. Click the Boolean control with the Operating tool to toggle between the TRUE and FALSE states. In edit mode, you can operate both Boolean controls and Boolean indicators. If *automatic tool selection is enabled*, you might need to move the cursor over the Boolean control or indicator and press the <Ctrl> key to temporarily switch to the Operating tool. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.

Customizing Boolean Objects

You can *use the Control Editor window* to design your own Boolean style. You also can use the Control Editor window to *import pictures* for the TRUE, FALSE, and transition states of any Boolean control or indicator.

Building Digital Tables

Digital tables contain data that you can plot on a digital waveform graph. You can build a digital table manually, convert analog data to digital data, or acquire a digital signal to create a digital table. Complete the following steps to build or modify a table of digital data.

1. Add a Build Digital Data function to the block diagram.
2. (Optional) If you want to modify existing digital data, wire the **digital data** input.
3. Right-click the **data** input and select **Create»Constant** or **Create»Control** from the shortcut menu.

- Enter values in the 2D **data** array. Each row in the 2D array represents a single binary value for one sample of data with the most significant bit of data in the first column and the least significant bit of data in the last column. Each value represents a specific digital data state. Valid values include 0-7, as indicated by the following table.

Digital Data State	Description
0 (Drive Low)	Force logic low. Drive to the low voltage level (V_{OL}).
1 (Drive High)	Force logic high. Drive to the high voltage level (V_{OH}).
Z (Force Off)	Force logic high impedance. Turn the driver off.
L (Compare Low)	Compare logic low (edge). Compare for a voltage level lower than the low voltage threshold.
H (Compare High)	Compare logic high (edge). Compare for a voltage level higher than the high voltage threshold.
X (Compare Unknown)	Compare logic unknown. Do not compare.
T (Compare Off)	Compare logic high impedance (edge). Compare for a voltage level between the low voltage threshold (V_{OL}) and the high voltage threshold (V_{OH}).
V (Compare Valid)	Compare logic valid level (edge). Compare for a voltage level either lower than the low voltage threshold (V_{OL}) or higher than the high voltage threshold (V_{OH}).



NOTE

If you wire an array constant to the **data** input, you can use the digital ring constant to simplify entering digital data in the array. The digital ring constant allows you to select from the 8 digital data states. To use the digital ring constant, right-click a value in the array and select **Replace** from the shortcut menu to display a temporary version of the **Functions** palette. Navigate to the **Digital Waveform** palette and select the digital ring constant. LabVIEW converts any numeric values you previously entered to their corresponding digital data states.

- (Optional) If you want to compress the digital data, wire the **transitions** input to define non-incremental transitions. The **transitions** array must contain the same number of rows as the **data** array. Expand the node to display an unwired input. Click the input and place a checkmark next to **transitions** to display the **transitions** input.
- Right-click the **digital data** output and select **Create»Indicator** from the shortcut menu to create an empty digital table.
- Run the VI to display the data in the digital table.

Related Information

Digital Tables

Graphing Digital Data

Analog to Digital VI

Creating a Typical DAQ Application

Digital Waveform Graphs

Listboxes, Tree Controls, and Tables

Listboxes, Tree Controls, and Tables

Use the listbox, tree control, and table controls and indicators on the **List, Table & Tree** and **Classic List, Table & Tree** palettes to give users a list of items from which to select.



RELATED INFORMATION

[Creating Listboxes](#) on page 1251

[Creating a Tree Control](#) on page 1250

[Using Tables](#) on page 1261

Creating a Tree Control

Use the tree control to give users a hierarchical list of items from which to select. You organize the items you enter in the tree control into groups of items, or nodes.

Complete the following steps to create a tree control.

1. Add a tree control to the front panel.
2. Use the Labeling tool to enter an item name in the tree control and press the <Enter> key. LabVIEW creates a unique tag for the item. The default tag is the text in the first cell that contains text, starting from the left. If an existing tag already uses that string, LabVIEW appends a number to the string to create a unique tag for the new item. You use this tag to modify items or to gather information about items programmatically.
3. Repeat step 2 for each item you want to enter. LabVIEW creates each new item at the same hierarchical level as the item above it. If there is no item above the new item, LabVIEW creates the item at the topmost hierarchical level.
4. Set the hierarchy of the items in the tree control by indenting items.
5. (Optional) Configure the tree control. You can add symbols and edit, disable, and delete items in a tree control. You also can set the expand/contract symbol type, case sensitivity, and item dragging for a tree control.
6. (Optional) Set the number of items a user can select in the tree control.

7. Create a Case structure to handle the selected item. A tree control with a selection mode set to **0 or 1 Item** or to **1 Item** appears on the block diagram as a string. You can right-click a tree control on the front panel and select **Selection Mode»Data Type** from the shortcut menu to change the data type. The data value for a single-selection tree control is the unique tag of the currently selected option. If no item is selected, the value is an empty string. A tree control with a selection mode set to **0 or More Items** or to **1 or More Items** appears on the block diagram as an array of strings, where the value(s) in the array represent the currently selected item(s). If no item is selected, the value is an empty array.

You also can use the TreeControl methods to add items to a tree control programmatically.



RELATED INFORMATION

[Indenting Tree Control Items](#) on page 1255

[Adding Symbols to Listboxes and Tree Controls](#) on page 1262

[Disabling or Enabling Listbox and Tree Control Items](#) on page 1253

[Deleting Rows in Listboxes and Tree Controls](#) on page 1252

[Setting Expand/Contract Symbols in Tree Controls](#) on page 1260

[Setting Case Sensitivity in Listboxes and Tree Controls](#) on page 1260

[Configuring Run-Time Drag-and-Drop Behavior for Text-Based Controls](#) on page 1085

[Setting the Number of Items You Can Select in Listboxes and Tree Controls at Run Time](#) on page 1261

[Creating a Case Structure to Execute Conditional Code](#) on page 1292

Creating Listboxes

Complete the following steps to create a listbox.

1. Add a listbox control to the front panel window.
2. Use the Labeling tool to enter an item name in the listbox and press the <Enter> key.
3. Repeat step 2 for each item you want to enter in the listbox.
4. (Optional) [Set the number of items a user can select](#) in the listbox.
5. [Create a Case structure](#) to handle the selected item. A listbox with a selection mode set to **0 or 1 Item** or to **1 Item** appears on the block diagram as a 32-bit signed integer. You can right-click a tree control on the front panel and select **Selection Mode»Data Type** from the shortcut menu to change the data type. The data value for a single-selection listbox is a number that represents the currently selected option. The first item in the list has a value of zero. If no item is selected, the value is -1. A listbox with a selection mode set to **0 or More Items** or to **1 or More Items** appears on the block diagram as an array of 32-bit signed integers, where the value(s) in the array represent the currently selected item(s). If no item is selected, the value is an empty array.

You also can use the ListBox properties to add items to a listbox and to configure a listbox programmatically.

Creating Multicolumn Listboxes

Complete the following steps to create a multicolumn listbox.

1. Add the multicolumn listbox control to the front panel window.
2. To add an option item, use the Labeling tool to type in a cell in the leftmost column.
3. Type in the cells in the same row to add information about each option item.
4. (Optional) *Set the selection mode* of a multicolumn listbox.
5. (Optional) *Change the color of the symbols* and the *color of the headers and cells*.
6. *Create a Case structure* to handle the selected item. A listbox with a selection mode set to **0 or 1 Item** or to **1 Item** appears on the block diagram as a 32-bit signed integer. You can right-click a tree control on the front panel and select **Selection Mode»Data Type** from the shortcut menu to change the data type. The data value for a single-selection listbox is a number that represents the currently selected option. The first item in the list has a value of zero. If no item is selected, the value is **-1**. A listbox with a selection mode set to **0 or More Items** or to **1 or More Items** appears on the block diagram as an array of 32-bit signed integers, where the value(s) in the array represent the currently selected item(s). If no item is selected, the value is an empty array.

You also can *add header text* to a multicolumn listbox, *show and hide multicolumn listbox headers*, and *add rows and columns* to a multicolumn listbox.

Deleting Rows in Listboxes and Tree Controls

In a listbox, right-click the row you want to delete and select **Delete Row** from the shortcut menu to delete the entire row. In a tree control, right-click the row you want to delete and select **Delete Item** from the shortcut menu to delete the row. Subsequent rows in the listbox or tree control shift up.



NOTE

Deleting an item in a tree control deletes all of the items underneath the item.

If you do not update the value of a tree control after deleting a row, the control may return old data. Deleting a selected item does not change the value of a tree control. After you delete an item, the tree value may be a tag for an item that no longer exists. You can select one of the remaining tree items to update the value of the tree control.



Note You cannot delete individual cells in a row.

Right-click a tree control and select **Delete All Items** from the shortcut menu to delete all rows in the tree control.

Disabling or Enabling Listbox and Tree Control Items

Right-click a listbox or tree control item and select **Disable Item** from the shortcut menu to disable an item. Right-click a listbox or tree control item and select **Enable Item** from the shortcut menu to enable a disabled item.

You also can use the Disabled Items property to disable listbox items programmatically and the Active Item:Disabled? property to disable tree control items programmatically.

Displaying Components in Listboxes, Tables, Tree Controls, and Digital Data Controls

Complete the following steps to select which components of the listbox, table, tree control, or digital data control are visible.



NOTE

The options available in the **Visible Items** shortcut menu vary depending on the control.

1. Right-click the listbox, table, tree control, or digital data control.
2. Select **Visible Items** and select one of the following items from the shortcut menu:
 - **Label** —Displays the *label*.
 - **Caption** —Displays the *caption*.
 - **Vertical Scrollbar** —Displays the vertical scroll bar.
 - **Horizontal Scrollbar** —Displays the horizontal scroll bar.
 - **Row Headers** —Displays the row headers of a listbox, table, or tree control.
 - **Transitions** —Displays the sample numbers in a digital data control.
 - **Column Header** —Displays the column header of a single-column listbox.
 - **Column Headers** —Displays the column headers of a multicolumn listbox, table, or tree control.
 - **Signal Numbers** —Displays the signal numbers in a digital data control.
 - **Vertical Lines** —Displays the vertical lines that separate columns.
 - **Horizontal Lines** —Displays the horizontal lines that separate rows.
 - **Symbols** —Allows you to display symbols next to items in the listbox, table, or tree control. If you select this item in a listbox, the listbox adds an extra column to the left side of the list for symbols. By default, items do not include symbols. You can *add symbols* by using the shortcut menu or a *Property Node*.
 - **Hierarchy Line Visibility** —If **Visible**, always displays lines to outline the hierarchy of the items as vertical and horizontal lines to the left of the items

in the tree control. If **Invisible**, the hierarchy lines are always invisible. If **Default**, LabVIEW shows the hierarchy lines if trees in the operating system show hierarchy lines. The default is **Visible**.

- **Index Display** —Displays the row-column index of the top left visible cell in a table.

You also can use the Table, DigitalTable, ListBox, MulticolumnListbox, and TreeControl properties to display components programmatically.

Editing Data in Tables

Complete the following steps to cut, copy, or paste data in a table.

1. **Select the cells, rows, or columns** that contain the data you want to cut or copy, or select the cells, rows, or columns in which you want to paste the data.
2. Right-click the selected cells and select **Data Operations»Cut Data, Copy Data, or Paste Data** from the shortcut menu.
 - **Cut Data** —LabVIEW copies only the selected data to the clipboard and removes the data from the table. If you cut any row or rows of data, all rows below it move up. If you cut any column of data, all columns to the right move left. If you cut a portion of a row or column, LabVIEW removes the entire row or column containing the cells you selected from the table, but copies only the selected data to the clipboard.
 - **Copy Data** —LabVIEW copies only the selected data to the clipboard.
 - **Paste Data** —LabVIEW pastes the data from the clipboard into the cells you selected. If you paste the data without first selecting a target or if the number and placement of the cells you selected are not the same as the data on the clipboard, LabVIEW adds rows and columns to the table as needed to accommodate the data.

Editing Listbox and Tree Control Cells at Run Time

Complete the following steps to edit cells in a listbox or tree control at run time.

1. Right-click the listbox or tree control and select **Editable Cells** from the shortcut menu to allow users to edit cells at run time. You also can use the Allow Editing Cells property to programmatically allow users to edit cells in a single-column listbox, multicolumn listbox, or tree control.
2. At run time, click the cell text you want to edit, and click it a second time after a short pause. You also can use the Edit Position property to programmatically select which cell to edit in a multicolumn listbox, or tree control.
3. Enter the text you want.

Use the Edit Cell event to handle the event that a user changes text in the cell of a single-column listbox, multicolumn listbox, or tree control.

Entering Data in Tables

Complete the following steps to use the keyboard to enter data into a table quickly.

1. Use the Operating tool or the Labeling tool to click inside a cell and enter the data.
2. Press the <Shift> key while you press the arrow keys to move the cursor to adjacent cells.
3. If the VI is in edit mode, press the <Enter> key on the keyboard to move the cursor to the cell below. When *tabbing through the front panel objects* in run mode, press the <Ctrl> and down arrow keys to tab into the table. Press the <Tab> key to move the cursor to adjacent columns and the <Enter> key on the keyboard to move to adjacent rows. To tab out of the table, press the <Ctrl> and up arrow keys. (macOS) Press the <Command> key. (Linux) Press the <Alt> key.



NOTE

If you *enable multiple line input*, pressing the <Enter> key on the keyboard creates a new line in the same cell and does not move the cursor to the cell below.

1. Press the <Ctrl-Enter> keys on the keyboard or the <Enter> key on the numeric keypad to end data entry.

Indenting Tree Control Items

Complete the following steps to move an item higher or lower in the tree control hierarchy.

1. Right-click an item in the tree control.
2. Select one of the following options from the shortcut menu:
 - **Indent Item** —Indents the item one level lower in the hierarchy. If the item already appears indented below the item directly above it, you cannot indent it. You can indent the current item, using the Operating tool, by pressing the period . key.
 - **Outdent Item** —Moves the item one level higher in the hierarchy. If the item already appears at the highest level in the hierarchy, you cannot outdent the item. You can move the current item one level higher, using the Operating tool, by pressing the comma , key.

If you want to configure an item so you cannot indent other items under it, right-click the item and select **Child-Only Item** from the shortcut menu. If you drag another item under a child-only item, LabVIEW places the item at the same hierarchical level as the child-only item. This menu item is not available if you right-click an item that has other items indented under it. You also can use the Active Item:Child-Only? property to configure an item so you cannot indent other items under it programmatically.

You also can use the Active Item:Indent Level property to set the indent level of an item in a tree control programmatically.

Inserting Items in Listbox and Tree Controls

Complete the following steps to insert an item at the end of a listbox or tree control.

1. Click a cell below the last item in the control.
2. Enter the text you want.

Complete the following steps to insert an item below an existing item in a listbox or tree control.

1. Click the item below which you want to insert a new item.
2. Press the <Shift-Enter> keys to insert an item below the current item. In tree controls, this key combination places the new item at the same level in the hierarchy as the current item, and all items below the new item move down one row.
3. Enter the text you want.

You also can right-click the listbox or tree control and select **Insert Item Before** from the shortcut menu to insert an item.

Inserting or Deleting Rows or Columns in Listboxes and Multicolumn Listboxes

Complete the following steps to insert a row or column in a listbox or a multicolumn listbox.

1. Right-click the row below which you want to insert a new row or right-click the column to the right of which you want to insert a column.
2. Select **Insert Row Before** or **Insert Column Before** from the shortcut menu.

To delete a row or column, right-click the row or column you want to delete and select **Delete Row** or **Delete Column** from the shortcut menu.



NOTE

You cannot insert or delete columns in a single-column listbox.

Inserting or Deleting Rows or Columns of Data in Tables

Complete the following steps to insert a row or column of data in a table.

1. Right-click the row below where you want to insert a new row or right-click the column to the right of where you want to insert a column.

2. Select **Data Operations»Insert Row Before** or **Insert Column Before** from the shortcut menu.

To delete a row or column, right-click the row or column you want to delete and select **Data Operations»Delete Row** or **Delete Column** from the shortcut menu.

These operations affect only the data in the table and do not affect the appearance of the rows or columns. For example, suppose a table has a red row containing data above a blue row containing data. If you right-click the red row and select **Data Operations»Insert Row Before** from the shortcut menu, LabVIEW inserts empty cells in the red row, and the data from the red row shifts down to the blue row. LabVIEW does not insert a new red row above the existing red row.

Limiting Tree Control Item Selection to Siblings

By default, LabVIEW limits the interactive selection of multiple items in a tree control to items that share the same parent. To allow the selection of items with different parents, right-click the tree control and remove the checkmark from the **Selection Mode»Limit Multiselect to Siblings** item in the shortcut menu. You also can use the Sibling Multiselect property to programmatically limit the selection of multiple items to siblings or to allow the selection of items with different parents.

If you limit the interactive selection of multiple items to siblings, you still can select items with different parents programmatically by using a local variable or setting the Value property of the tree control. However, you then cannot add new items to the selection interactively.

If you enable the **Limit Multiselect to Siblings** shortcut menu item, but there are multiple non-sibling items already selected, subsequent selection behavior might not be intuitive. If you reset the selection to something valid, the selection behavior returns to normal.

Opening and Closing Tree Control Items

You can open and close individual parent items in a tree control at run time in any of the following ways:

- Click the expand/contract symbol to the left of the parent item.
- Double-click the parent item tag or its *hierarchy lines*.
- Use the + and – keys on the numeric keypad.
- Use the right and left arrow keys.

You also can right-click a tree control and select **Open All Items** from the shortcut menu to open all items in the tree control at once. Right-click a tree control and select **Close All Items** from the shortcut menu to close all items in the tree control at once.

Resizing Rows and Columns in Listboxes, Tables, and Tree Controls

Complete the following steps to resize rows and columns in listbox, table, and tree controls.

1. Right-click the control and select **Autosize Row Height** from the shortcut menu to remove the checkmark from this menu item.
2. Move the Positioning tool over a horizontal line to resize a row or a vertical line to resize a column. A cursor similar to the following cursor appears when the tool is positioned properly: 
3. Drag the horizontal or vertical line to resize the row or column, respectively. Press the <Shift> key while dragging the line to size all the rows or columns to the same size. You can size multiple rows or columns of a table to the same size. Use the Operating tool to select the rows or columns and press the <Shift> key while using the Positioning tool to resize a row or column in the selection.



NOTE

If you *enable multiple line input*, right-click the control and select **Autosize Row Height** from the shortcut menu to size each row to its contents and display all the data in each cell. This option is enabled by default.

Selecting Listbox and Tree Control Items at Run Time

Use the following ways to select items in a listbox or tree control while a VI is in run mode:

- Click an item. In a multiple-selection listbox or tree control, press the <Ctrl> key and click additional items to select more than one item. (macOS) Press the <Command> key. (Linux) Press the <Alt> key. Press the <Shift> key and click an item above or below the current item to select all items between the current item and the second item you selected.



NOTE

By default, LabVIEW *limits* the interactive selection of multiple items in a tree control to items that share the same parent. To allow the selection of items with different parents, right-click the tree control and remove the checkmark from the **Selection Mode»Limit Multiselect to Siblings** item in the shortcut menu.

- Click an item, then press and hold the <Ctrl> key and use the up and down arrow keys to navigate through listbox or tree control items. (macOS) Press the <Command> key. (Linux) Press the <Alt> key. Press the spacebar to add an item with focus to the list of selected items. While still holding the <Ctrl> key, press the spacebar again to remove the item with focus from the list of selected items.
- Type part of the name of the item you want to select. Use the left and right arrow keys to go to the previous or next items that match the characters you type.

In a listbox or tree control with a selection mode that allows you to select 0 items, press the <Ctrl> key and click an item to deselect it. (macOS) Press the <Command> key. (Linux) Press the <Alt> key.



Note If you assign <PageUp>, <PageDown>, <Home>, or <End> as a shortcut key in the **Key Navigation** page of the **Properties** dialog box for a control, you no longer can use that key to select items in a listbox.

Selecting Parent Items in Tree Controls

Complete the following steps to allow users to select items that have other items indented under them and items that are not child-only and do not have other items indented under them.

1. Right-click the tree control.
2. Select **Selection Mode»Allow Selection of Parent Items** from the shortcut menu. If you remove the checkmark next to this item, the user can select only items that you configure as child-only items.

You also can use the Allow Selection of Parent Items property to select parent items programmatically.

Selecting Table Cells, Rows, or Columns

Use the Operating tool to click a cell and drag the cursor to select multiple cells in a table. You also can press the <Shift> key and drag the cursor to select multiple cells. If you move the cursor outside the visible contents of the table, the table scrolls and extends the selection. A border appears around the cells you select.



NOTE

If you do not want to extend the selection as you move the cursor outside the visible contents of the table, right-click the table and select **Selection Scrolling** from the shortcut menu to remove the checkmark. If you do not want to outline the selection with a border, right-click the table and select **Advanced»Show Selection** from the shortcut menu to remove the checkmark.

Complete the following steps to select an entire row or column of data.

1. Use the Operating tool to click the left border of the row or the top border of the column. An arrow cursor appears when the tool is positioned properly.
2. Drag the cursor across the height of the row or the width of the column.

To select all the data in a table, use the Operating tool to click the upper left corner of the table. A double arrow cursor appears when the tool is positioned properly.

Setting Case Sensitivity in Listboxes and Tree Controls

Complete the following steps to set how LabVIEW handles upper and lowercase characters when you type characters in a listbox or tree control at run time.

1. Right-click the listbox or tree control.
2. Select **Keyboard Mode** from the shortcut menu and select from the following options:
 - If you select **System Default** (the default setting), the listbox or tree control behaves the same as other listboxes and tree controls for a given platform. (Windows and macOS) The listbox and tree control are case insensitive. (Linux) The listbox and tree control are case sensitive.
 - If you select **Case Sensitive**, an item is selected only if the uppercase and lowercase letters typed in by the user exactly match the item.
 - If you select **Case Insensitive**, case is ignored.

You also can use the Keyboard Mode property to set the case sensitivity of a listbox, multicolumn listbox, or tree control programmatically.

Setting Column Widths at Run Time

To allow users to adjust column widths at run time, right-click the multicolumn listbox, tree control, or table and select **Moveable Column Separators** from the shortcut menu.

Setting Expand/Contract Symbols in Tree Controls

Complete the following steps to set the type of symbol to display next to each item that has other items indented under it. You can use the expand/contract symbols to *open and close parent items* in the tree.

1. Right-click a tree control.
2. Select **Expand/Contract Symbol Type** and select one of the following items from the shortcut menu:
 - **None** —Displays no symbols.
 - **LabVIEW** —Displays platform-independent triangle symbols.
 - **Mac OS** —Displays macOS-style triangle symbols.
 - **Windows** —Displays Windows-style plus and minus symbols.
 - **Default** —Displays the default symbols for the operating system.

You also can use the Expand/Contract Symbol:Symbol Type property to set programmatically the type of symbol to display next to each parent item.

Right-click a tree control and select **Expand/Contract Symbol Type»Show at Indent Level 0** from the shortcut menu to show or hide the expand/contract symbols for items at the highest level of the hierarchy. If you remove the checkmark next to this option, you can still expand or collapse an item at indent level 0 by double-clicking the item. To disable the double-clicking functionality, wire a True constant to the **Discard?** parameter of the Item Close? event.

You also can use the Expand/Contract Symbol:Show Symbol at Root property to show or hide the symbols programmatically.

Setting the Number of Items You Can Select in Listboxes and Tree Controls at Run Time

Complete the following steps to specify how many items you can [select in a listbox or tree control](#) at run time.

1. In the front panel window, right-click the listbox or tree control.
2. Select **Selection Mode** from the shortcut menu and select from the following options:
 - **0 or 1 Item** —The behavior is the same as for **1 Item**, except that you can deselect the current item, leaving no items selected. You can move items in the listbox by dragging them with the cursor.
 - **1 Item** —(Default) One item in the listbox or tree control always is selected (highlighted). When you click a different item, the new item is selected (highlighted), and the previous item is deselected.
 - **0 or More Items** —You can select multiple items.
 - **1 or More Items** —You can select multiple items, but at least one item must be selected.

You also can use the Selection Mode property to set the number of items you can select in a listbox or tree control programmatically.

In tree controls, you also can [limit the selection of multiple items](#) to siblings.

Right-click a listbox or tree control and select **Selection Mode»Highlight Entire Row** from the shortcut menu to highlight the entire row when you select an item.

Using Tables

A table is a 2D array of strings. Use the String functions to manipulate tables.

Each cell in a table is a string, and each cell resides in a column and a row. You enter headings when you add the table to the front panel, and you can change them using the Labeling tool or Operating tool. The row and column headings of a table are not part of the table data. Table headings are separate pieces of data that you can read and set using Property Nodes.

The index display at the upper left corner of the table indicates which cell is visible. You can operate these indexes just as you do on an array.

**TIP**

You can export data and images from tables to locations such as Microsoft Excel. Right-click a table and select **Export** from the shortcut menu to view available export options.

**RELATED INFORMATION**

[Displaying an Array in a Table](#) on page 1100

[Exporting Data and Images](#) on page 1182

Adding Header Text to Listboxes, Tables, and Tree Controls

Complete the following steps to add header text to a listbox, table, or tree control.

1. Right-click the control and select **Visible Items»Row Headers** and **Visible Items»Column Header**, or **Visible Items»Column Headers** for controls with multiple columns, from the shortcut menu to display the row and column headers.

**NOTE**

Single-column listboxes do not have row headers.

1. Use the Labeling tool to click the header in which you want to add text.
2. Enter the text you want.



Note You cannot enter text in the top left header cell.

Adding Symbols to Listboxes and Tree Controls

Complete the following steps to add symbols to a listbox or tree control.

1. Add one of the following controls to the front panel window:
 - Listbox
 - Tree control
2. Use the Labeling tool and enter text into the listbox or tree control.
3. Right-click the control and select **Visible Items»Symbols** from the shortcut menu.

4. Right-click an item in the listbox or tree control, select **Item Symbol** from the shortcut menu, and select the symbol you want.

**NOTE**

In a multicolumn listbox, LabVIEW adds the symbol to the first column.

You also can use the Item Symbols property to add symbols to a listbox programmatically.

You also can use the TreeControl, ListBox, and MulticolumnListBox methods to customize item symbols in a tree control or listbox.

Coloring Listboxes, Multicolumn Listboxes, Tree Controls, and Tables

Complete the following steps to *change the color* of the headers or cells of listboxes, multicolumn listboxes, tree controls, and tables.

1. Right-click the control and select **Visible Items»Row Headers** to show the row headers and **Visible Items»Column Header**, or **Visible Items»Column Headers** on controls that have multiple columns, to show the column headers.
2. Using the Coloring tool, right-click a non-header cell to display the color picker and select a color for all cells.
3. Using the Coloring tool, right-click a row header to display the color picker and select a color for all row headers.
4. Using the Coloring tool, right-click a column header to display the color picker and select a color for all column headers.
5. Using the Coloring tool, right-click the top left header cell of the control to display the color picker and select a color for the top left header cell.

You also can use the Active Cell property to set the row and column of the active cell(s) programmatically. Use the Cell Background Color property to set the background color of the active cell(s) programmatically.

Configuring Cells for Multiple Line Input

To configure the cells in a listbox, table, or tree control to create a new line in a cell when you press the <Enter> key on the keyboard, right-click the control and select **Multi-line Input** from the shortcut menu.

If you enable multiple line input, right-click the control and select **Autosize Row Height** from the shortcut menu to size each row to its contents and display all the data in each cell. This option is enabled by default.

You also can use the Multiple Line Input property to configure the cells in a listbox, multicolumn listbox, table, or tree control to create a new line in a cell programmatically when you press the <Enter> key.

Structures

Adding Structures to the Block Diagram

Complete the following steps to add a structure to the block diagram.

1. Select a structure on the Structures palette. The cursor becomes a small icon of the structure.
2. Click the block diagram where you want to place the top corner of the structure and move the cursor down and to the right or left.
3. Click the block diagram again when the structure is the size you want.

After you add the structure, you can [resize](#) it.

Resizing Structures

Complete the following steps to resize a structure on the block diagram.

1. Move the Positioning tool over the structure border. Resizing handles appear at the corners of the structure and in the middle of each structure border.
2. Move the cursor over a resizing handle to change the cursor to the resizing cursor.
3. Use the resizing cursor to drag the resizing handles. When you resize a structure by dragging the resizing handles, the structure grows or shrinks in real time.



NOTE

By default, you cannot resize a structure smaller than the objects inside it. To disable this behavior for a structure, right-click the structure border and select **Auto Grow** from the shortcut menu to remove the checkmark. If you drag an object inside a structure that is smaller than the object, the structure resizes to enclose the object after you place the object on the block diagram.

1. Release the mouse button.



Note If the [alignment grid is enabled](#), you can resize a structure only in units the size of the grid.

When resizing a single frame in a Flat Sequence structure using the resizing handle on a shared edge of two frames, make sure you are resizing the correct frame by noting which frame displays resizing handles. To shrink a frame using the resizing handle on the left

edge of a frame, drag the handle to the right. To grow a frame using the same resizing handle, drag the handle to the left. When you shrink or grow a frame, the other frames in the Flat Sequence structure remain their current size and move to adjust to the resized frame.

Adding Objects inside Structures

Add an object inside a structure by dragging it inside the structure or by building the structure around the object. You cannot put an object inside a structure by moving the structure over the object.

Complete the following steps to practice adding objects inside a structure.

1. Add the Case Structure to the block diagram.
2. Add the Tick Count (ms) function inside the structure.
3. Move the Tick Count (ms) function close to the border of the structure. Notice that when you add or move an object in a structure near the structure border, the structure resizes to add space for that object after you add the object to the block diagram. To disable the automatic resizing behavior for a structure, right-click the structure border and select **Auto Grow** from the shortcut menu to remove the checkmark.
4. Move the Tick Count (ms) function outside the structure.
5. Add another structure around the Tick Count (ms) function.
6. Select the second structure and delete it. Notice that you also deleted the function inside the structure.



NOTE

You can **remove a structure** without deleting the objects in the structure.

1. Add another Tick Count (ms) function outside the remaining structure.
2. Move the structure over the Tick Count (ms) function. The shadow on the function indicates that the function is on top of the structure but is not inside it.
3. Select the Case structure and delete it. Notice that you did not delete the function because it was not inside the structure.

Replacing One Structure with Another Structure

Complete the following steps to replace a structure or loop with another structure without deleting the objects in the structure or loop.



NOTE

You cannot replace the **In Place Element** structure or the Event structure with another structure.

1. Right-click the structure or loop you want to replace.
2. Select **Replace with Name of Structure or Loop** from the shortcut menu.

Removing Structures without Deleting Objects in the Structure

Complete the following steps to remove a structure without deleting the objects in the structure.

1. Right-click the structure you want to remove.
2. Select **Remove Name of Structure** from the shortcut menu.

With For Loops and While Loops, LabVIEW copies the contents of the loop to the block diagram and automatically connects any elements wired through tunnels.

Removing a *structure with multiple cases* preserves only the visible case, frame, or subdiagram and any block diagram terminals. All other cases, frames or subdiagrams are deleted.

Loops

Automatically Concatenating Arrays Leaving Loops

To automatically concatenate an array or string leaving a loop, wire the array or string directly to the loop output tunnel, right-click the tunnel, and select **Tunnel Mode»Concatenating** from the shortcut menu. Selecting **Concatenating** appends all inputs in order, forming an output array of the same dimension as the array input wired or a single concatenated output string. Additional tunnel modes include **Last Value** and **Indexing**. **Last Value** displays the last value from the last loop iteration. **Indexing** builds an array of higher dimension. You can *change when LabVIEW writes values to the loop output tunnel* based on a condition.

Refer to the Loop Tunnel Modes VI in the labview\examples\Structures\Shift Registers and Tunnels directory for an example of automatically concatenating arrays.

Processing Individual Elements in an Array or a Collection with a Loop

You can configure a Loop to pass individual elements of an array or a collection into and out of the loop.

Configuring the Input Tunnel to Process Individual Elements of an Array or a Collection

If you wire an array or a collection to a For Loop or While Loop, you can read and process every element in that array or collection by enabling auto-indexing. When you wire an array or collection from an upstream node to an auto-indexing input tunnel on the loop border and enable auto-indexing on the input tunnel, elements of that array or collection enter the loop one at a time, starting with the first element.

Complete the following steps to pass elements into an array through a loop:

1. Right-click the tunnel at the loop border.
2. Choose the option that corresponds to how you want to process array elements:
 - Select **Disable Indexing** from the shortcut menu to disable auto-indexing, which passes the entire array or collection into the loop and returns an array with the same number of elements.
 - Select **Enable Indexing** from the shortcut menu to enable auto-indexing, which passes each individual element of the array or collection into the loop and returns a single element.



NOTE

To avoid default data output values when a loop does not execute, enable auto-indexing every time a For Loop has an output tunnel.

A bracketed glyph appears on the loop output tunnel to indicate that auto-indexing is enabled.

Configuring the Output Tunnel to Return an Alternative Value of an Array

Complete the following steps to configure the output tunnel of the loop to return the indexed value of an array, the last value of an array, or a concatenated array:

1. Right-click the output tunnel.
2. Select **Tunnel Mode»Last Value, Indexing, or Concatenating** from the shortcut menu respectively.

Refer to the following table for more information about each of the output options the tunnel can return.

Output Name	Description
Indexing	The output array receives a new element from every iteration of the loop. Therefore, auto-indexed output arrays are always equal in size to the number of iterations. For example, if the loop executes 10 times, the output array has 10 elements.
Last value	The output tunnel returns only the element from the last iteration of the loop.

Output Name	Description
Concatenating	LabVIEW appends all inputs in order, forming an output array of the same dimension as the array input wired. The Concatenating tunnel mode concatenates arrays the same as the Build Array function.

Between the output tunnel and the next node, the thickness of the wire indicates whether the loop is indexing, returning the last value, or concatenating the values. Indexing creates a thicker wire than concatenating because the wire contains an array of higher dimension instead of an array of the same dimension. When the output tunnel returns the last value, the wire is thinner than when indexing or concatenating.



NOTE
The loop indexes scalar elements from 1D arrays, 1D arrays from 2D arrays, and so on. The opposite occurs at output tunnels. Scalar elements accumulate sequentially into 1D arrays, 1D arrays accumulate into 2D arrays, and so on.

Refer to the labview\examples\Arrays\Arrays.lvproj for examples of building arrays by disabling auto-indexing.

Choosing the Type of Auto-Indexing for an Array or Collection Wired to a Loop

Loop Type	Array or Collection Configuration	Auto-Indexing Behavior	Example
For Loop	One array or collection wired to an input tunnel and an unwired count terminal	If you enable auto-indexing on an array or a collection wired to a For Loop input tunnel, LabVIEW uses the array or collection size as the number of times to execute so you do not need to wire the count terminal. LabVIEW enables auto-indexing by default for every array or collection you wire to a For Loop. NOTE  You can disable auto-indexing if you do not need to process arrays or collections one element at a time.	—
For Loop	Wired count terminal	If you enable auto-indexing when the count terminal is wired, the actual number of iterations that LabVIEW executes is the smaller of either the array or collection size or the value of the count terminal.	For example, if you wire an array or a collection with 10 elements to a For Loop and a value of 15 to its count terminal, the loop executes only 10 times.

Loop Type	Array or Collection Configuration	Auto-Indexing Behavior	Example
For Loop	Multiple arrays or collections	If you enable auto-indexing for more than one tunnel, the number of iterations that LabVIEW executes is the smallest of the array or collection sizes.	For example, if two auto-indexed arrays enter the loop, with 10 and 20 elements respectively, and you wire a value of 15 to the count terminal, the loop executes only 10 times, indexing all elements of the first array but only the first 10 elements of the second array. As another example, if you plot data from two sources on one graph and you want to plot the first 100 elements, wire 100 to the count terminal. However, if the smaller data source includes only 50 elements, the loop executes 50 times and indexes only the first 50 elements of each data source. Use the Array Size function to determine the size of arrays.

Loop Type	Array or Collection Configuration	Auto-Indexing Behavior	Example
While Loop	One wired array or collection	If you enable auto-indexing for an array or collection entering a While Loop, the While Loop indexes the array or collection the same way as a For Loop. However, the number of iterations a While Loop executes is not limited by the size of the array or collection because the While Loop iterates until a specific condition occurs. If the number of loop iterations is greater than the array or collection size, the While Loop handles all the elements in the array or collection and uses the default value of the array or collection element type for later iterations. You can prevent the default value from passing into the While Loop by using the Array Size or Collection Size functions, which indicate how many elements are in the array or collection. Configure the While Loop to stop executing when the number of iterations equals the array or collection size. Auto-indexing for While Loops is disabled by default. TIP Because you cannot determine the size of an output array or collection in advance, enabling auto-indexing for the output of a For Loop is more efficient than that of a While Loop. Iterating too many times can cause your system to run out of memory.	—

Related Information

[Arrays](#)

[Grouping Data with Collections](#)

[For Loops: Repeating Operations a Set Number of Times](#)

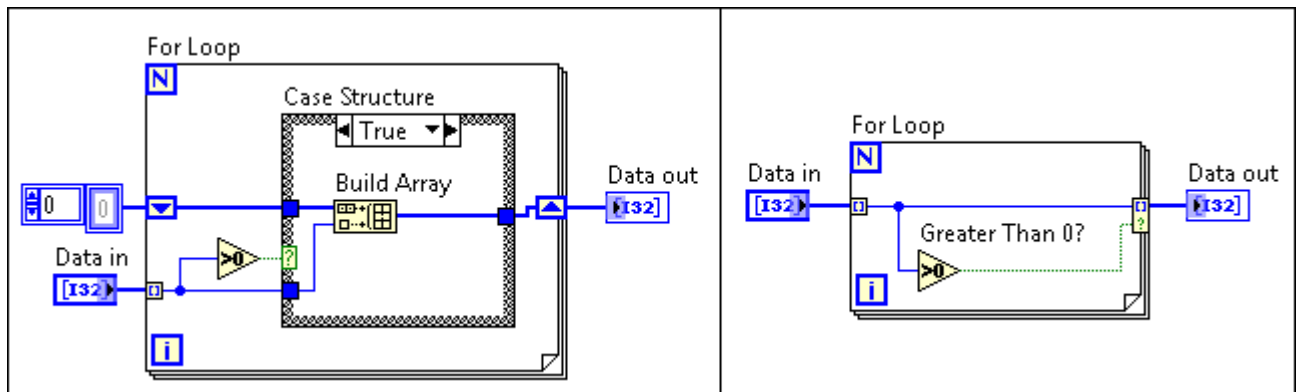
[While Loops: Repeating Operations Until a Condition Occurs](#)

[Automatically Concatenating Arrays Leaving Loops](#)

Conditionally Writing Values to Loop Output Tunnels

You can determine what values LabVIEW writes to the loop output tunnel based on a condition you specify by right-clicking the loop output tunnel and selecting **Tunnel Mode»Conditional** from the shortcut menu.

The **Conditional** menu item simplifies the block diagram. For example, the following block diagram shows how the Conditional output tunnel from the right diagram accomplishes the same goal as the Case structure from the left diagram.



You can select the **Conditional** menu item for any tunnel mode. The following table shows the effects of selecting the **Conditional** menu item for each tunnel mode.

Tunnel Mode	Result
Last Value	Conditionally stores the last value from the last loop iteration in the tunnel. Reading the last value is useful when determining what value was last written to a Feedback Node .
Indexing	Conditionally indexes elements into the array.
Concatenating	Conditionally appends all inputs in order, forming an output array of the same dimension as the array input wired.

Refer to the Loop Tunnel Modes VI in the `labview\examples\Structures\Shift Registers and Tunnels` directory for an example of conditionally writing values to loop output tunnels.

Checking Boolean Values on Each Iteration

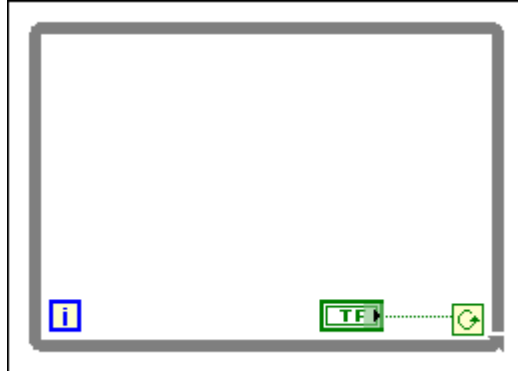
If you want the loop to check the value of a terminal on each iteration, place the terminal inside the loop.

Complete the following steps to cause the loop to check the value of a Boolean terminal every iteration.

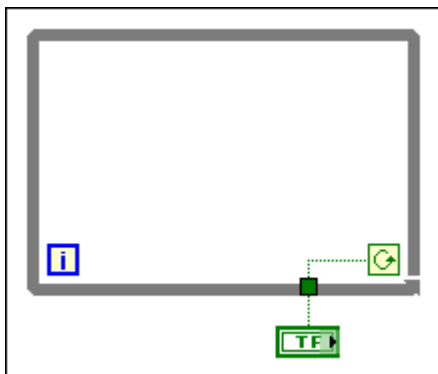
1. Place a Boolean control on the front panel.
2. Place the Boolean terminal inside the loop on the block diagram.
3. Wire the Boolean terminal to the conditional terminal of the loop.

**NOTE**

You can stop the While Loop by changing the value of the front panel control to FALSE, as shown in the following block diagram.



If you wire the conditional terminal of a While Loop set to **Continue if True** to a Boolean terminal set to TRUE and placed outside the loop, you cause an infinite loop, as shown in the following block diagram.



You also cause an infinite loop if you wire the conditional terminal of a While Loop set to **Stop if True** to a Boolean terminal set to FALSE and placed outside the loop.

To stop an infinite loop, you must abort the VI by clicking the **Abort** button on the toolbar. Changing the value of the control does not stop the infinite loop because the value is not propagated until the loop stops and you run the VI again.

Changing the Continuation Behavior of Loops

If the conditional terminal of a While Loop, a For Loop, or a (Real-Time, Windows) Timed Loop is **Stop if True** , the loop executes its subdiagram until the conditional terminal

receives a TRUE value. If the conditional terminal is **Continue if True** , the loop executes its subdiagram until the conditional terminal receives a FALSE value.



NOTE

A For Loop with a conditional terminal executes until the conditional terminal receives the stop value or until the set number of iterations completes, whichever happens first.

Complete the following steps to change the continuation behavior of a loop with a conditional terminal.

1. Right-click the conditional terminal. In a While Loop or a Timed Loop, you also can right-click the loop border.
2. Select **Stop if True** or **Continue if True** from the shortcut menu.

You also can use the Operating tool to click the conditional terminal to change the condition.

You must wire a Boolean value or an error cluster to the conditional terminal in a While Loop or a For Loop with a conditional terminal.

Controlling the Timing on Loops

By default, LabVIEW executes loops and block diagrams as quickly as possible. However, you might want to control the rate at which a process executes, such as the rate at which data values are plotted to a chart.

You can control timing on loops in one of the following ways:

- Use a Wait function in the loop to specify the amount of time in milliseconds before the loop re-executes.
- (Real-Time, Windows) Use a **Timed Loop** to execute code with time bounds and delays.

Handling Errors Using Conditional Terminals on Loops

You can control the iteration of a While Loop or a For Loop when an error passes to a conditional terminal. Complete the following steps to handle errors using conditional terminals.

1. Select the error cluster terminal.

2. Wire the error cluster to the conditional terminal of a While Loop or a For Loop.

**NOTE**

If you wire the error cluster to the conditional terminal, only the TRUE or FALSE value of the **status** parameter of the error cluster passes to the terminal. In addition, the shortcut menu items **Stop if True** and **Continue if True** change to **Stop on Error** and **Continue while Error**.

Specifying the Number of Times Operations in For Loops Execute

A For Loop repeatedly executes the subdiagram inside it the number of times you wire to the count terminal.

Complete the following steps to specify the number of times an operation executes using a For Loop.

1. Add a For Loop to the block diagram.
2. *Add objects inside the For Loop* to create a subdiagram that the For Loop repeats.
3. Determine how many times the For Loop iterates in one of the following ways:
 - Wire a numeric value to the count terminal .
 - Wire an array to an input tunnel on the loop border by *auto-indexing* the array.

**NOTE**

A VI does not run if it contains a For Loop that does not have a numeric value wired to the count terminal or an auto-indexing input tunnel in which the size of the array determines the iteration count.

You also can *add a conditional terminal* to a For Loop to configure the loop to stop when a Boolean condition or error occurs. A For Loop with a conditional terminal executes until the condition you define occurs or until all iterations complete, whichever happens first.

Stopping For Loops When a Condition Occurs

You can add a conditional terminal to configure a *For Loop* to stop when a condition occurs. A For Loop with a conditional terminal executes until the condition occurs or until all iterations complete, whichever happens first. For example, if you want a loop to execute a set number of times unless an error occurs, you can use a For Loop with a conditional terminal and wire an error cluster to the conditional terminal.

Complete the following steps to stop a For Loop when a condition occurs.

1. Add a For Loop to the block diagram.

2. Right-click the loop border and select **Conditional Terminal** from the shortcut menu. A conditional terminal  appears inside the loop and the count terminal appearance changes from the  symbol to the  symbol.
3. *Add objects inside the For Loop* to create a subdiagram that the For Loop repeats.
4. Wire Boolean data, such as a Boolean control or an output from a Comparison function, or an error cluster to the conditional terminal. You also can right-click the terminal and select **Create Constant** or **Create Control** from the shortcut menu to create the Boolean object.
5. (Optional) By default, the conditional terminal is set to **Stop if True**. You can *change the behavior* of the conditional terminal by right-clicking the terminal and selecting **Continue if True**. When the conditional terminal is **Continue if True** , the loop executes until the terminal receives a FALSE value. You also can use the Operating tool to click the conditional terminal to change the condition.



NOTE

If you wire the conditional terminal in a For Loop, you cannot predetermine the number of iterations the loop executes. You can *determine the number of iterations* the loop completes by *auto-indexing an output array* and checking the number of elements after the loop executes or by wiring an indicator to the loop iteration terminal and checking the count after the loop executes.

The number wired to the count terminal or the number of elements in the auto-indexed input array acts as the maximum number of loop iterations that complete if the condition wired to the conditional terminal never occurs. If you want a loop to iterate until a condition occurs with no maximum number of iterations, use a While Loop instead.

Parallel Iterations: Improving For Loop Execution Speed

You can improve the execution speed of a For Loop by enabling parallel iterations. LabVIEW normally partitions For Loop iterations into chunks and executes them sequentially. When you enable parallel iterations on a For Loop, LabVIEW executes chunks simultaneously, improving the execution speed of computationally intensive For Loops.

Complete the following steps to execute For Loop iterations in parallel:

1. Enable parallel For Loop iterations.
2. Verify that the For Loop can run with parallel iterations.
3. If necessary, programmatically configure parallel For Loop iterations.

Review the performance considerations to ensure that a For Loop will benefit from parallelization.

Examples

Refer to the Parallel For Loop Reduction VI in the `labview\examples\Structures\Parallel For Loop` directory for an example of using a For Loop with parallel iterations.



RELATED INFORMATION

[Enabling Parallel For Loop Iterations](#) on page 1276

[Verifying That a For Loop Can Run with Parallel Iterations](#) on page 1276

[Configuring Parallel For Loop Iterations to Tweak Performance](#) on page 1277

[Performance Considerations for Parallel For Loop Iterations](#) on page 1279

Enabling Parallel For Loop Iterations

You can improve For Loop execution speed by enabling parallel For Loop iterations. Complete the following steps to enable parallel iterations:

1. Right-click the For Loop and select **Configure Iteration Parallelism**. LabVIEW displays the For Loop Iteration Parallelism dialog box.
2. Place a checkmark in the **Enable loop iteration parallelism** checkbox.
3. Set the **Number of generated parallel loop instances** to the maximum number of logical processors you expect any computer running the VI to ever contain. For example, if you are developing an application for use on computers with a maximum of eight processors, set the number to eight.
4. Click the **OK** button. The parallel instances **P** terminal appears below the count **N** terminal of the configured For Loop. You do not need to wire a value to the parallel instances terminal to enable parallel iterations. However, you can use the terminal to [tweak the performance](#) of a parallel For Loop.

After you enable parallel iterations, [verify](#) that a For Loop can run with parallel iterations.

Verifying That a For Loop Can Run with Parallel Iterations

Not all For Loops can run with parallel iterations. After you enable parallel iterations on a For Loop, complete the following steps to verify that a For Loop can run with parallel iterations.

1. Ensure that the environment settings are configured to display warnings.
2. Select **View»Error List** to open the Error List window.
3. Review the **errors and warnings** section. The For Loop can run with parallel iterations if none of the following errors or warnings appear:

- Error: Parallel For Loop cannot contain an array dependence between loop iterations
 - Error: Parallel For Loop cannot contain a dependence between loop iterations
 - Error: Parallel For Loop cannot contain a stacked shift register
 - Error: Parallel For Loop cannot contain an uninitialized shift register
 - Error: Parallel For Loop cannot contain a conditional terminal
 - Error: Parallel For Loop cannot contain a Feedback Node
 - Error: Parallel For Loop cannot contain a Boolean control with latch option
 - Error: Feature not supported on target
 - Warning: Node with side effects
4. If enabling parallel iterations results in an error or warning, disable parallel iterations or modify the code. Refer to the following support document at ni.com to learn more about when to use parallel iterations.

Refer to the Parallel For Loop Iteration Order VI in the `labview\examples\Structures\Parallel For Loop` directory for an example of a For Loop with parallel iterations that causes a warning.



NOTE

If you have multiple VIs, such as a project or a VI containing a subVI, use the Find Parallelizable Loops Results window to find For Loops that can run with parallel iterations.



RELATED INFORMATION

[Enabling Parallel For Loop Iterations](#) on page 1276

[Displaying Warnings](#) on page 952

Configuring Parallel For Loop Iterations to Tweak Performance

When you [enable parallel For Loop iterations](#), you can tweak performance programmatically using the parallel instances **P** and chunk size **C** terminals. The default configurations of the terminals produce optimal performance improvements in most cases, so programmatic configuration is rarely necessary. However, you can use the parallel instances and chunk size terminals to customize a configuration different from the default.

Using the Parallel Instances Terminal to Tweak Performance

After you enable parallel For Loop iterations, the parallel instances terminal appears inside the For Loop. Wire a numeric value to the parallel instances terminal to programmatically configure the number of parallel instances to execute. At compile time, LabVIEW generates a number of parallel instances equal to the minimum of the value provided in the **Number**

of generated parallel loop instances field of the For Loop Iteration Parallelism dialog box and the value wired to the input of the parallel instances terminal. Processors execute parallel instances simultaneously to improve performance.

If you leave the input of the parallel instances terminal unwired, LabVIEW automatically detects the number of logical processors in the computer and uses it as the default parallel instances terminal value. In most cases, optimal performance occurs when the number of executed parallel instances is equal to the number of processors in the computer, so you should usually leave the input of the parallel instances terminal unwired.

If code in a For Loop performs any waiting operation, optimal performance occurs when the number of loop instances executed is greater than the number of logical processors, called oversubscribing. For example, if a parallel For Loop instance waits to acquire data from external hardware, oversubscribing allows a processor to execute a second parallel instance while it waits on the first. When a parallel For Loop instance executes at the same time as other computationally intensive code, undersubscribing, or executing a number of loop instances less than the number of logical processors, results in optimal performance. For example, if a For Loop and a subVI execute simultaneously, undersubscribing limits the processing resources devoted to the For Loop and reserves resources for other operations.

Using the Chunk Size Terminal to Specify a Custom Iteration Schedule

LabVIEW partitions loop iterations into chunks consisting of loop iterations. With parallel iterations enabled, processors execute chunks simultaneously to improve execution speed. By default, LabVIEW schedules chunks by size from larger to smaller. Executing larger chunks first decreases scheduling overhead, while executing smaller chunks last decreases processor idleness. You should programmatically configure chunk size only if the For Loop would benefit from an iteration schedule different from the default, such as a schedule that executes smaller chunks before larger chunks.

After you enable parallel For Loop iterations, complete the following steps to programmatically configure the iteration schedule using the chunk size terminal:

1. Right-click the For Loop and select **Configure Iteration Parallelism**. LabVIEW displays the For Loop Iteration Parallelism Dialog Box.
2. In the **Iteration partitioning schedule** section, select **Specify partitioning with chunk size (C) terminal**. The chunk size terminal appears below the parallel instances terminal.
3. Wire a numeric value or an array of numeric values to the chunk size terminal.

When you wire a numeric value to the chunk size terminal, the value specifies the number of iterations to include in each chunk. Wiring an array of numeric values provides more precise control over chunk size. Each value in the array specifies the number of iterations to include in a chunk, with the first chunk beginning at index 0.



NOTE

If you wire an array with too many chunk sizes, LabVIEW ignores the extra values. If you wire too few chunk sizes, the last element in the array determines the size of the remaining chunks.

Performance Considerations for Parallel For Loop Iterations

In some cases, [enabling parallel iterations](#) on a For Loop may not result in significant performance improvements. Use the following guidelines to optimize the performance benefits of a For Loop running with parallel iterations:

- Enabling parallel loop iterations adds execution overhead. Do not enable parallel loop iterations on For Loops that are not computationally intensive. You can only improve performance if that overhead is countered by a decrease in execution time.
- If you have a group of nested For Loops, enable parallel iterations on only the outer loop.
- [Blocking nodes](#), such as the Write to Text File function and non-reentrant subVI calls, can limit the parallelism between loop iterations. Avoid using blocking nodes in a For Loop running with parallel iterations.

Refer to the white paper at ni.com to learn more about improving performance with parallel loop iterations.

Specifying Conditions for While Loops

The [While Loop](#) contains a subdiagram that executes until the conditional terminal receives a particular Boolean value.

Complete the following steps to specify conditions for a While Loop.

1. Add a While Loop to the block diagram.
2. [Add objects inside the While Loop](#) to create a subdiagram that the While Loop repeats.
3. Wire Boolean data, such as a Boolean control or an output from a Comparison function, to the conditional terminal . You also can right-click the terminal and select **Create Constant** or **Create Control** from the shortcut menu to create the Boolean object.
4. (Optional) By default, the conditional terminal is set to **Stop if True**. You can [change the behavior](#) of the conditional terminal by right-clicking the terminal and selecting **Continue if True**.



NOTE

If you do not specify a condition, the While Loop becomes an infinite loop. Changing the value of the control does not stop the infinite loop because the value is only read once, before the loop starts. To stop an infinite loop, you must abort the VI by clicking the **Abort Execution** button on the toolbar.

Because the VI checks the conditional terminal at the end of each iteration, the While Loop always executes at least one time. The VI does not run if you do not wire the conditional terminal.

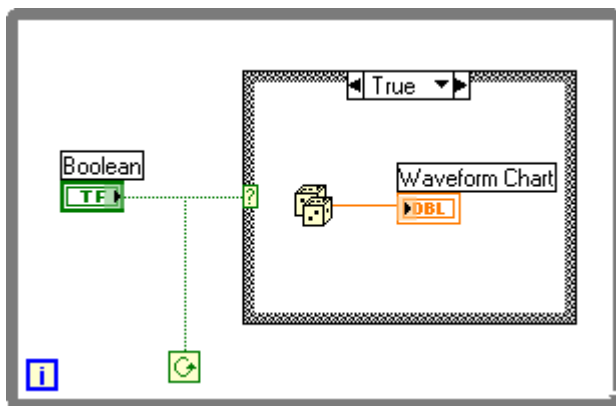
You can *prevent code execution* in the first iteration by placing a Case structure inside the loop.

Preventing Code Execution in the First While Loop Iteration

Because a VI checks the conditional terminal at the end of each iteration, the *While Loop* always executes at least one time.

Complete the following steps to create a While Loop that tests the condition and does not execute any code if the condition is FALSE on the first iteration.

1. Add a While Loop to the block diagram and drag out a rectangular region.
2. Add a Case structure inside the While Loop and drag out a rectangular region inside the While Loop.
3. Create a Boolean control to control the VI.
 1. Right-click the selector terminal of the Case structure and select **Create Control** from the shortcut menu to create a Boolean control.
 2. Wire the Boolean control to the conditional terminal of the While Loop. To keep the wiring neat, move the conditional terminal below the Boolean control and create a wire branch from the existing wire segment.
4. *Change the behavior* of the conditional terminal to **Continue if True**.
5. Add the Random Number (0-1) function inside the **True** case.
6. Add a waveform chart to the front panel.
7. On the block diagram, wire the Random Number (0-1) function to the waveform chart. Make sure that both the function and the chart terminal are inside the case you selected.



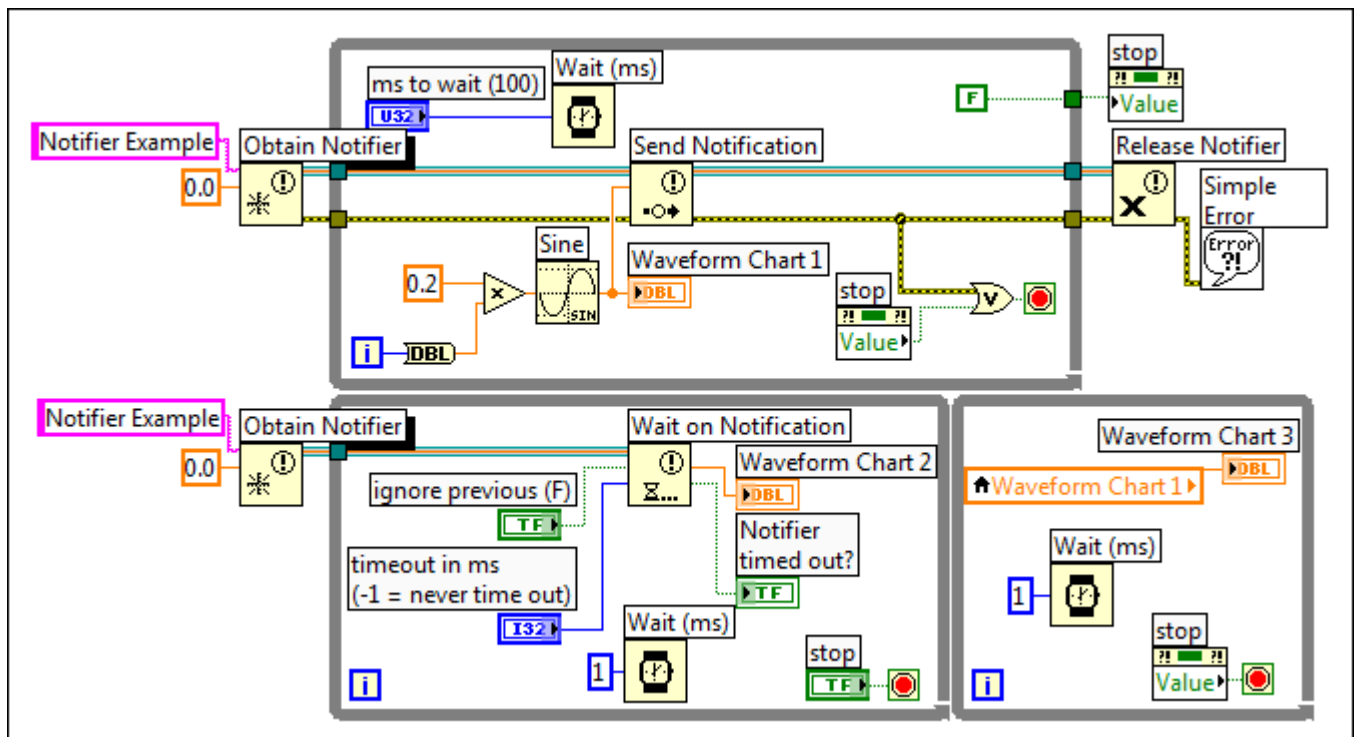
Complete the following steps to verify that the While Loop operates correctly and to run the VI.

1. Examine the front panel. Notice that the default value of the Boolean control is OFF.
2. Click the **Run** button a few times to run the VI. Notice that the chart does not display any values. The While Loop executes once, but the Case structure does not execute while the Boolean control is set to OFF.
3. With the Operating tool, click the **Boolean** button to ON.
4. Click the **Run** button. The While Loop executes continuously, generating random numbers and displaying them on the chart.
5. Click the **Boolean** button to stop the VI.

Example: Handling Notifier Operation Functions in While Loops

The following example shows a common mistake when using the Notifier Operation functions and a stop Boolean control in a While Loop.

When the value of the **stop** Boolean control in the second loop changes to TRUE, the VI does not always stop, and the user must click the **Abort Execution** button to stop the VI completely. The reason is because the loop does not finish executing until each function in the loop completes its last execution, and the Wait on Notification function in the second loop is not connected to a stop condition.



Complete the following steps to correct this problem.

1. Wire the **error out** cluster on the Wait on Notification function to the stop button.
2. Set the **force destroy?** input on the Release Notifier function to TRUE to destroy the notifier once the loop completes execution.

Passing One Value to the Next Loop Iteration

Use shift registers on For Loops, While Loops, or (Real-Time, Windows) Timed Loops to transfer values from one loop iteration to the next.

Complete the following steps to pass values to the next loop iteration by creating shift registers.

1. Add a structure to the block diagram. For example, add a While Loop to the block diagram.
2. Right-click the left or right border of the loop and select **Add Shift Register** from the shortcut menu.

You can use multiple shift registers to remember iteration values and you can use an initialized shift register to reset the initial value of a shift register.



NOTE

(FPGA Module) The FPGA Module uses groups of flip-flops to implement shift registers.



RELATED INFORMATION

[Shift Registers: Passing Values between Loop Iterations](#) on page 416

[Loops and Other Structures](#) on page 411

[Passing Multiple Values to the Next Loop Iteration](#) on page 1282

[Resetting the Initial Value of Shift Registers](#) on page 1283

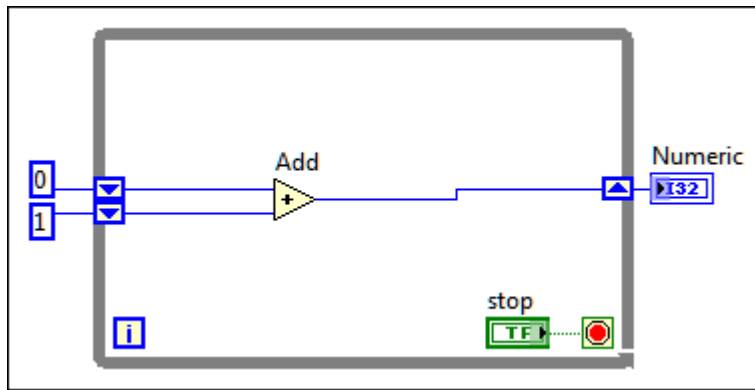
Passing Multiple Values to the Next Loop Iteration

You can create stacked shift register terminals on the left side of a loop to remember multiple previous iterations and carry those values to the next iterations. This technique is useful for averaging data points.

Complete the following steps to configure a [shift register](#) to carry over more than one previous value to the next iteration.

1. [Create a shift register](#).
2. Right-click the left or right shift register terminal.
3. Select **Add Element** from the shortcut menu to create an additional terminal on the left side of the loop.

Stacked shift registers can occur only on the left side of the loop because the right terminal transfers the data generated from only the current iteration to the next iteration, as shown in the following block diagram.



Iteration	Top Left Shift Register End Value	Bottom Left Shift Register End Value	Right Shift Register End Value
0	0	1	1
1	1	0	1
2	1	1	2
3	2	1	3
4	3	2	5

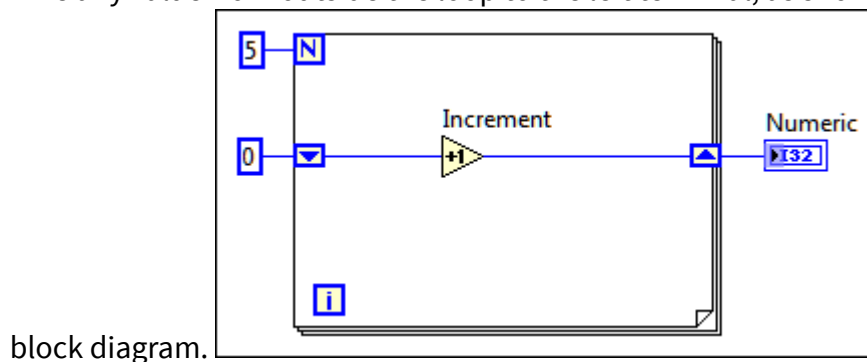
If you add another element to the left terminal in the previous block diagram, values from the last two iterations carry over to the next iteration, with the most recent iteration value stored in the top shift register. The bottom terminal stores the data passed to it from the previous iteration.

Resetting the Initial Value of Shift Registers

Use an initialized shift register to reset the initial value of a shift register each time a VI runs.

Complete the following steps to reset the initial value of a *shift register*.

1. *Create a shift register.*
2. Wire any value from outside the loop to the left terminal, as shown in the following



block diagram.

In the previous block diagram, the For Loop executes five times, incrementing the value the shift register carries by one each time. After five iterations of the For Loop, the shift register passes the final value, 5, to the indicator and the VI quits. Each time you run the VI, the shift register begins with a value of 0.

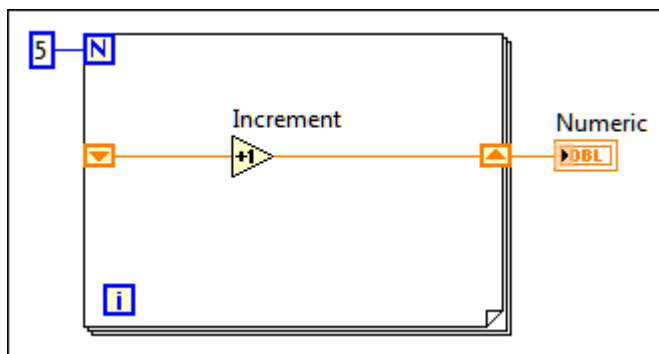
Use a loop with an uninitialized shift register so that each time the VI runs, the initial input of the shift register is the last value from the previous execution. Leave the input to the left shift register terminal unwired for an uninitialized shift register to preserve state information between subsequent executions of a VI.



NOTE

If you do not initialize the register, the loop uses the value written to the register when the loop last executed or the **default value** for the data type if the loop has never executed.

The following block diagram shows an uninitialized shift register.



In the previous block diagram, the For Loop executes five times, incrementing the value the shift register carries by one each time. The first time you run the VI, the shift register begins with a value of 0, which is the default value for a 32-bit integer.

After five iterations of the For Loop, the shift register passes the final value, 5, to the indicator, and the VI quits. The next time you run the VI, the shift register begins with a value of 5, which was the last value from the previous execution. After five iterations of the For Loop, the shift register passes the final value, 10, to the indicator.

If you run the VI again, the shift register begins with a value of 10, and so on. Uninitialized shift registers retain the value of the previous iteration until you close the VI or **LabVIEW recompiles the block diagram**.

Replacing Shift Registers with Tunnels

Tunnels are the terminals that feed data into and out of structures. Replace shift registers with tunnels when you no longer need to transfer values from one loop iteration to the next.

Complete the following steps to replace shift registers with tunnels:

1. Right-click the shift register.

2. Select **Replace with Tunnels** from the shortcut menu.

Replacing Tunnels with Shift Registers

Replace tunnels with shift registers when you need to transfer values from one loop iteration to the next.

Complete the following steps to replace tunnels with shift registers:

1. Right-click the tunnel.
2. Select **Replace with Shift Register** from the shortcut menu.

If no tunnel exists on the loop border opposite of the tunnel you right-clicked, LabVIEW automatically creates a pair of shift register terminals. If a tunnel exists on the loop border opposite of the tunnel you right-clicked, LabVIEW replaces the tunnel you right-clicked with a shift register terminal, and the cursor becomes a shift register icon, shown as follows.



When the cursor becomes a shift register icon, complete one of the following actions:

- Click a tunnel on the opposite side of the loop to replace the tunnel with a shift register.
- Click the block diagram to place the shift register on the loop border directly across from the other shift register terminal.



NOTE

If the shift register terminal appears behind a tunnel, the shift register is not wired.

If you convert a tunnel with auto-indexing enabled to a shift register on a While Loop, the wire to any node outside the loop breaks because shift registers cannot auto-index. Complete the following steps when the wire breaks:

1. Delete the broken wire and the output following the broken wire.
2. Right-click the output wire of the right shift register and select **Create»Indicator** from the shortcut menu.

Related Information

[Shift Registers: Passing Values between Loop Iterations](#)

[Processing Individual Elements in an Array or a Collection with a Loop](#)

While Loop

For Loop

Passing the Last Value from the Previous Execution to the Feedback Node

When you add a Feedback Node to the block diagram, the initializer terminal is attached to the node and the node globally initializes when the VI compiles or loads by passing the last value from the previous execution. You also can use the Configuration page of the Properties dialog box to configure a Feedback Node.

Complete the following steps to configure the Feedback Node to globally initialize when you have an input value and the VI compiles or loads.

1. Add a Feedback Node to the block diagram.
2. Wire an input value to the initializer terminal. When you set **Delay** to two or more executions or iterations, you can wire an array to the initializer terminal to initialize each delay element to an arbitrary value.



NOTE

If a Feedback Node globally initializes and you set an initial value, by default the Feedback Node initializes to that value on the first call of the VI at the beginning of each execution. If you do not wire an input value to the initializer terminal, the initial input of the Feedback Node for the first execution is the default value for the data type.

3. Right-click the initializer terminal and select **Globally Initialize»Initialize On Compile Or Load**.



RELATED INFORMATION

[Feedback Nodes: Storing Data between Loop Iterations](#) on page 417

[Block Diagram Objects](#) on page 126

Resetting the Input Value on a Feedback Node Each Time a Loop Executes

By default, the [Feedback Node](#) globally *initializes when the VI compiles or loads*. However, you also can configure the node to initialize on loop execution.

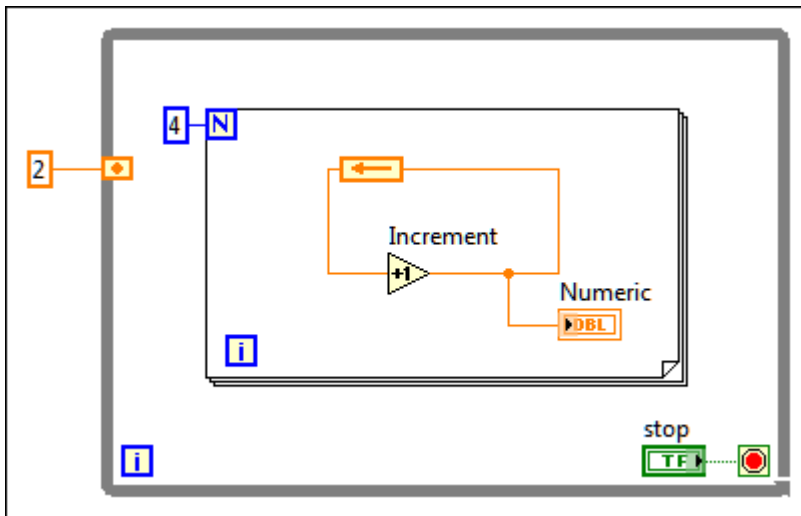
Complete the following steps to move the initializer terminal to the left edge of a loop, which resets the input value each time a loop executes.

1. Add a Feedback Node to a loop or to a group of nested loops and structures on the block diagram.
2. Right-click the initializer terminal and select **Move Initializer One Loop Out** from the shortcut menu to move the initializer terminal to the next loop out.
3. Wire an input value from outside the loop to the initializer terminal.

After you move the initializer terminal to the left edge of a loop, you can move the initializer terminal back to its original location inside the loop by right-clicking the initializer terminal and selecting **Globally Initialize»Initialize On Compile Or Load** from the shortcut menu. Select **Move Initializer One Loop In** to move the initializer to the next loop in.

If you move the initializer terminal to the edge of the outermost loop, select **Move Initializer One Loop In** to move the initializer terminal to the next loop in.

If you configure the Feedback Node to initialize when a loop executes, the Feedback Node initializes before the first iteration of every loop execution. The initializer terminal changes to the  icon, which changes color depending on the data type you wire to the terminal, as shown on the loop in the following figure.



In the previous block diagram, although the Feedback Node is in the For Loop, the initializer terminal is on the edge of the outermost While Loop. Thus, the Feedback Node initializes to the input value 2 every time the While Loop executes, whereas the outcome of the Increment function builds on previous values for each For Loop execution. Separating the Feedback Node and initializer is useful in nested Case structures when you want the Feedback Node to execute only in certain cases.

Replacing Shift Registers with a Feedback Node

Complete the following steps to replace *shift registers* with a Feedback Node.

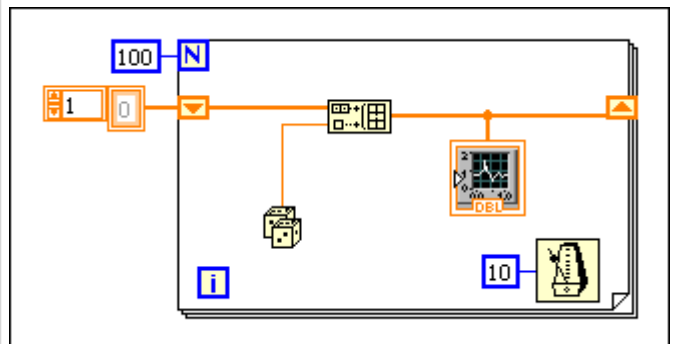
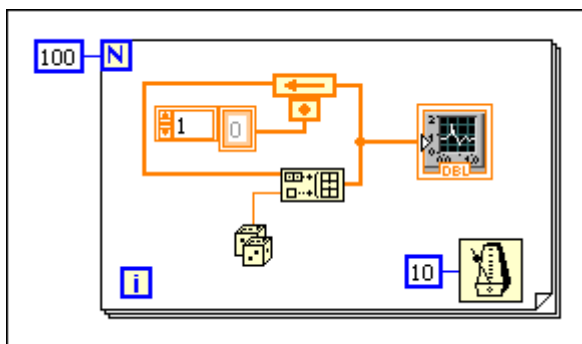
1. Right-click the shift register.
2. Select **Replace with Feedback Node** from the shortcut menu.

Replacing a Feedback Node with Shift Registers

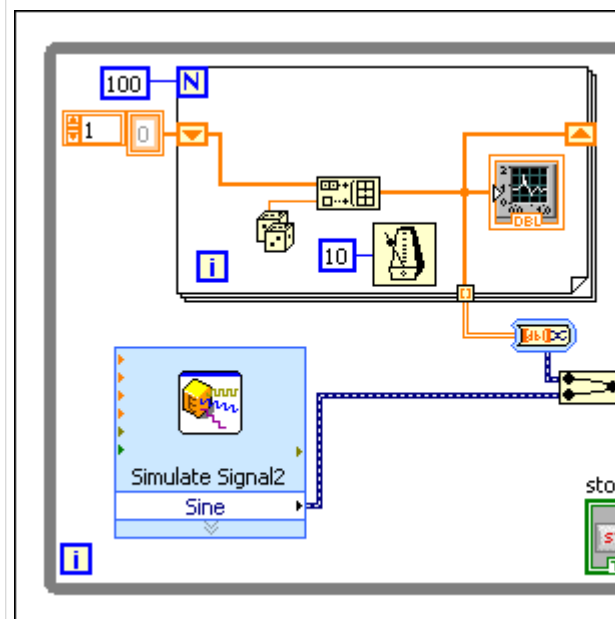
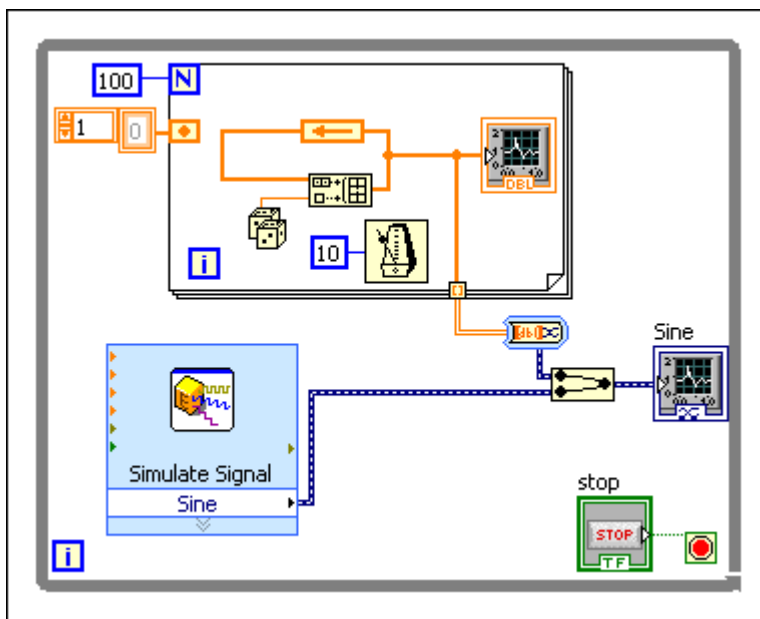
Complete the following steps to replace a Feedback Node with shift registers.

1. Right-click the Feedback Node.
2. Select **Replace with Shift Register** from the shortcut menu.

If you do not move the **initializer terminal** to the edge of a loop before replacing the Feedback Node with shift registers, LabVIEW places the shift registers on the edges of the loop that previously contained the node, as shown in the following block diagrams.



If you **move the initializer terminal** to the edge of a loop before you replace the Feedback Node with shift registers, LabVIEW places the shift registers on the edges of that loop, as shown in the following block diagrams.



Changing the Direction of the Feedback Node Arrow

You can change the arrow direction on a Feedback Node to indicate the flow of data along the wire. Changing the arrow direction changes only the cosmetic appearance of the Feedback Node and the wires connected to it on the block diagram and does not change the direction of the flow of data.

Complete the following steps to change the arrow direction on a Feedback Node.

1. Add a Feedback Node to the block diagram.

2. Right-click the Feedback Node and select **Change Direction** from the shortcut menu.

**NOTE**

LabVIEW automatically changes the arrow direction based on how you wire the Feedback Node. For example, if you change the direction of the flow of data along the wire, LabVIEW changes the direction of the Feedback Node arrow.

Subdiagram and Case Structures

Adding and Duplicating Subdiagrams

Right-click a structure border and select **Add Case** (or **Frame** or **SubdiagramAfter** or **Before**, if available, from the shortcut menu to add a [subdiagram](#). You also can press the <Shift-Enter> keys in the selector label to add a subdiagram to a structure.

Complete the following steps to make a copy of the visible subdiagram in a Case, Stacked Sequence, Conditional Disable, Diagram Disable, or Type Specialization structure and insert it after the original subdiagram.

1. [Move to the subdiagram](#) you want to duplicate.
2. Right-click the structure border and select **Duplicate Case** (or **Frame** or **Subdiagram** from the shortcut menu. The duplicate subdiagram contains all objects from the original subdiagram except block diagram terminals that correspond to front panel objects.

**NOTE**

You also can press the <Ctrl-Shift-Enter> keys in the case selector label to duplicate the visible subdiagram of a Case structure.

1. If you are [configuring a Conditional Disable structure](#), right-click the structure border and select **Duplicate Subdiagram** from the shortcut menu to display the Configure Condition dialog box. Select a **Symbol** and enter a **Value** in the **Configure Condition** dialog box.



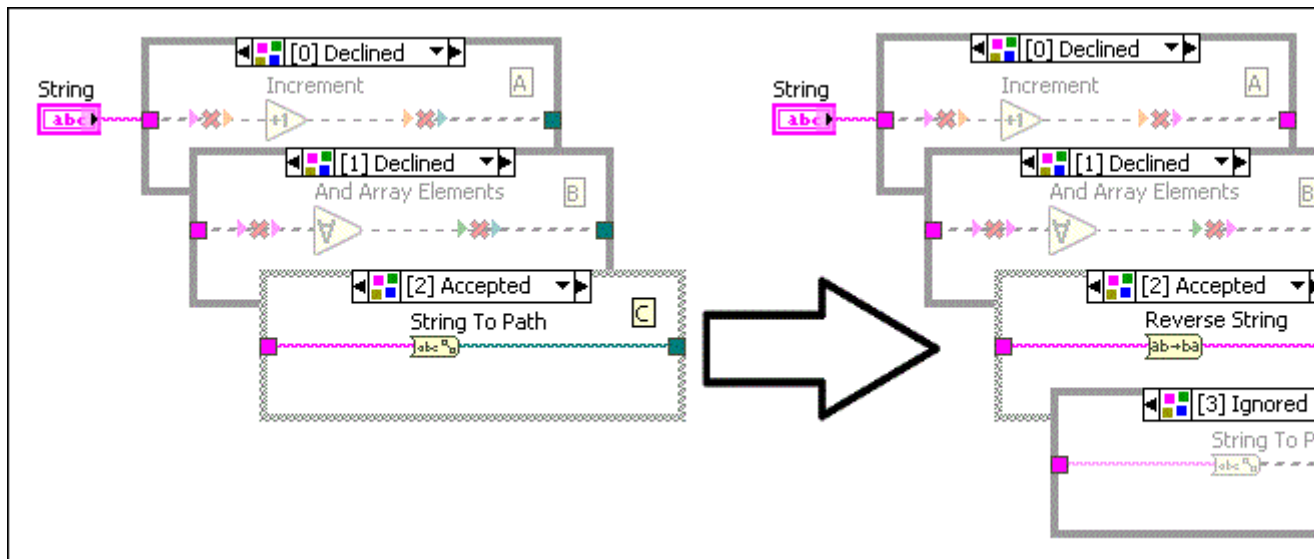
Note If you do not use the Conditional Disable structure in a LabVIEW project, RUN_TIME_ENGINE, TARGET_BITNESS, and TARGET_TYPE are the only symbols available. If you use the Conditional Disable structure in a LabVIEW project, you also can create and define symbols using the Conditional Disable Symbols page.

Consider the following issues when you add or duplicate a subdiagram:

- When you add, [delete](#), or [rearrange frames](#) in a Stacked Sequence structure, LabVIEW automatically adjusts the numbers in the frame labels.

- When you add or delete frames in a Flat Sequence structure, the structure resizes automatically.
- When you rearrange the subdiagrams in a Conditional Disable structure, multiple subdiagrams can be TRUE, but LabVIEW compiles and executes only the first subdiagram that is TRUE.
- Adding or deleting subdiagrams in a Type Specialization structure affects the subdiagram labels and how the VI runs. This is because the Type Specialization structure attempts to compile each subdiagram in order and declines subdiagrams with syntax errors. It accepts the first subdiagram with no syntax error and ignores the remaining subdiagrams.

For example, the following Type Specialization structure has subdiagrams A, B, and C. It declines the first two subdiagrams and accepts the last subdiagram C (indexed 2). If you add subdiagram D before subdiagram C, the index of subdiagram C becomes 3. The structure accepts subdiagram D and ignores subdiagram C because the structure compiles subdiagram D with no error before attempting to compile subdiagram C.



- You also can *add or duplicate cases for event structures*.

Moving between Subdiagrams

Use one of the following ways to scroll through the available *subdiagrams* :

- Click the decrement and increment arrows in the structure selector label, if available.
- (Windows) Move the cursor over the selector label and press the <Ctrl> key while moving the mouse wheel. (Linux) Move the cursor over the selector label and press the <Meta> key.

Use one of the following ways to move to a specific subdiagram:

- Click the down arrow next to the case or frame name and select the subdiagram from the shortcut menu.
- Right-click the structure border and select **Show Case** or **Show Frame**. Select the case or frame label from the shortcut menu.

Deleting Subdiagrams

Complete the following steps to delete the visible *subdiagram*.

1. Move to the subdiagram you want to delete.
2. Right-click the structure border and select one of the following options:
 - **Delete This Case**
 - **Delete This Frame**
 - **Delete This Subdiagram**
 - **Delete This Event Case**

If only one subdiagram exists, you cannot delete the visible subdiagram.

Rearranging Subdiagrams

Complete the following steps to rearrange *subdiagrams* for most structures:

1. Right-click the structure border and select **Rearrange Cases** or **Rearrange Subdiagrams** from the shortcut menu to display the Rearrange Cases dialog box.
2. Select a case or subdiagram in the **Case List** and drag it to a new location. A horizontal line appears to indicate where to insert the case or subdiagram in the list. You also can click the **Sort** button to rearrange the cases or subdiagrams alphabetically.
3. Click the **OK** button.

For Case structures, you also can rearrange the cases by *swapping* or *shifting* the cases.

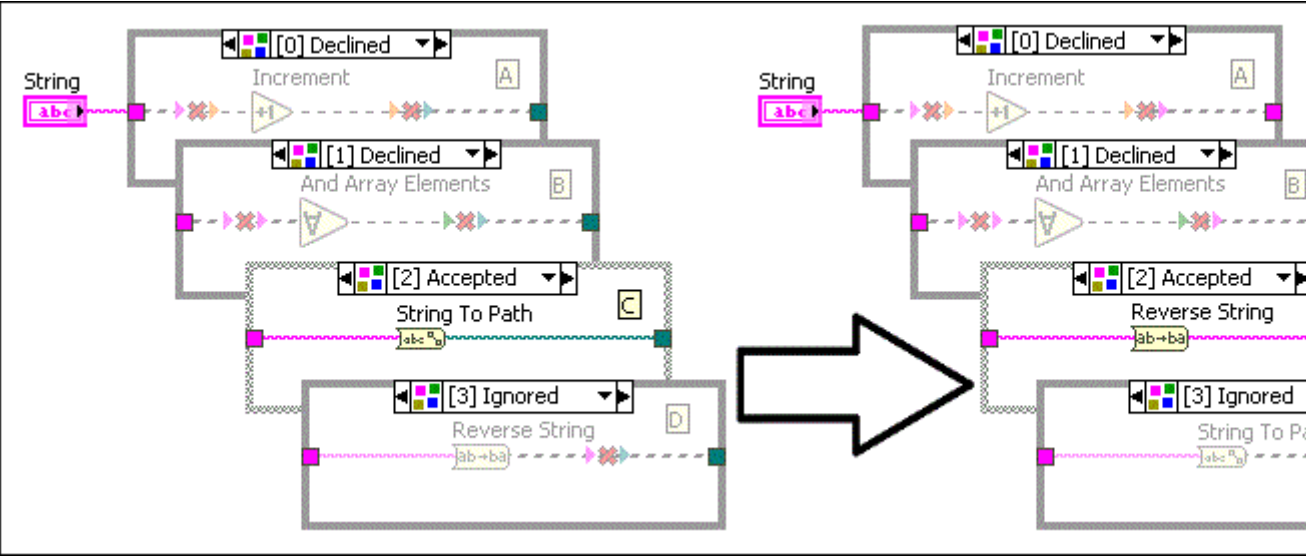
To rearrange the frames in a Stacked Sequence structure, right-click the structure border, select **Make This Frame**, and select a new frame number from the shortcut menu.

Consider the following issues when you rearrange the subdiagrams of a structure:

- You can rearrange the cases in a Case structure with three or more cases. You can rearrange the cases in an Event structure with two or more cases. You can rearrange the subdiagrams in a *disable structure* with two or more subdiagrams.
- When you rearrange the subdiagrams in a Conditional Disable structure, multiple subdiagrams can be TRUE, but LabVIEW compiles and executes only the first subdiagram that is TRUE.

- Rearranging the subdiagrams in a Type Specialization structure affects the subdiagram labels and how the VI runs. This is because the Type Specialization structure attempts to compile each subdiagram in order and declines subdiagrams with syntax errors. It accepts the first subdiagram with no syntax error and ignores the remaining subdiagrams.

In the following example, the Type Specialization structure has subdiagrams A, B, C, and D. It accepts subdiagram C and ignores subdiagram D. If you switch subdiagrams C and D, the indexes of subdiagrams C and D get switched. The structure accepts subdiagram D and ignores subdiagram C because the structure compiles subdiagram D with no error before attempting to compile subdiagram C.



Creating a Case Structure to Execute Conditional Code

Use a **Case** structure to specify two or more case subdiagrams that execute based on the input value you pass to the structure. Refer to the Case Structure - Selector Data Types VI in the `labview\examples\Structures` directory for an example of using Case structures.

Complete the following steps to specify values of case subdiagrams when you create a Case structure.

1. Add a Case structure to the block diagram.
2. Wire an input value to the case selector  to determine which subdiagram executes. You can position the case selector anywhere on the left border of the Case structure. Wire different **data types** to the case selector to create specific types of subdiagrams, as shown in the following table.

Data Type	Subdiagrams of Case Structure

The structure has a True case and a False case.

Data Type	Subdiagrams of Case Structure
The structure has an Error case and a No Error case. You also can use an error cluster to <i>handle errors</i> .	

The structure can have any number of cases.



NOTE

By default, string values you wire to the case selector are case sensitive. To allow case-insensitive matches, wire a string value to the case selector, right-click the border of the Case structure, and select **Case Insensitive Match** from the shortcut menu.

Tip To convert a tunnel to the case selector, right-click the tunnel and select **Replace with Case Selector** from the shortcut menu. If the previous case selector is wired, LabVIEW converts the previously wired case selector to a tunnel. You can also use the Replace With Case Selector method to convert a tunnel to the case selector programmatically.

3. *Add objects inside the Case structure* to create subdiagrams that the Case structure can execute. If necessary, *add or duplicate subdiagrams*.
4. For each case, use the Labeling tool to *enter a single value or lists and ranges of values* in the selector label at the top of the Case structure. The values you enter in each selector label execute a specific subdiagram based on the input value you pass to the structure.

Note For values of case subdiagrams with a string data type, use *backslash codes* to indicate backslashes and non-displayable characters. For example, enter `\\` to indicate a single backslash and `\r` to indicate a carriage return.

Entering Values for each Case Structure Subdiagram

You can enter a single value or lists and ranges of values in the selector label of a **Case** structure, as shown in the following table.

Type of Values	Description of Type of Values
Numeric ranges	- Numeric ranges—Specify a range as 10 . . 20, meaning all numbers from 10 to 20 inclusively. - Open-ended numeric ranges—Specify a range . . 100, meaning all numbers less than or equal to 100, and 100.. represents all numbers greater than or equal to 100. - Lists and ranges—Use commas to separate values, for example . . 5, 6, 7 . . 10, 12, 13, 14. When you enter values that contain overlapping ranges in the same case selector label, the Case structure redisplay the label in a more compact form. The previous example redisplay as . . 10, 12 . . 14. NOTE When the selector terminal is wired to an error cluster, LabVIEW converts numeric values in the selector label to error codes by appending Error to the numeric values or ranges.
String ranges	A range of a . . c includes all strings beginning with a or b, but not c. String ranges are case sensitive. For example, a range of A . . c behaves differently than a . . c because LabVIEW uses ASCII values to determine a string range.
Enumerated values	The values display in quotation marks, for example "red", "green", and "blue". However, you do not need to type the quotation marks when you enter the values unless the string or enumerated value contains a comma or range symbol (", " or " . . "). In a string value, use special backslash codes for non-alphanumeric characters, such as \r for a carriage return, \n for a line feed, and \t for a tab.

If you change the data type of the wire connected to the case selector of a Case structure, the Case structure automatically converts the selector label values to the new data type when possible. If you convert a numeric value, for example 19, to a string, the string value is "19". If you convert a string to a numeric value, LabVIEW converts only those string values that represent a number. The other values remain strings. If you convert a number to a Boolean value, LabVIEW converts 0 to False and 1 to True, and all other numeric values become strings.

If you enter a selector label value that is not the same type as the object wired to the case selector, the value appears red. This indicates that the VI will not run until you delete or edit the value.



NOTE

Because of the possible round-off error inherent in floating-point arithmetic, you cannot use floating-point numbers as selector label values. If you wire a floating-point value to the case selector, LabVIEW rounds the value to the nearest integer. If you type a floating-point value in the selector label, the value appears red to indicate that you must delete or edit the value before the structure can execute. You also cannot use fixed-point numbers as selector label values. If you wire a fixed-point number to the case selector, the VI appears with a broken **Run** button.

Avoiding Wiring Unwired Output Tunnels on Case Structures

You can create multiple input and output tunnels for a **Case** structure. Inputs are available to all case subdiagrams, but all subdiagrams do not have to use each input. Refer to the Case Structure - Selector Data Types VI in the `labview\examples\Structures` directory for an example of using Case structures.

However, when you create an output tunnel on the border of one subdiagram of the Case structure, tunnels appear at the same position on the border for all the other cases. If even one output tunnel is not wired, the center of every output tunnel on the structure appears as a white square.

Avoid wiring all unwired output tunnels by using the default data type value for the tunnel in one of the following ways:

- Right-click the output tunnel and select **Use Default If Unwired** from the shortcut menu to use the default value for the tunnel data type for all unwired tunnels.
- Configure the tunnels on a structure to [wire the input and output tunnels automatically](#) in unwired cases, which is the recommended best practice.



NOTE

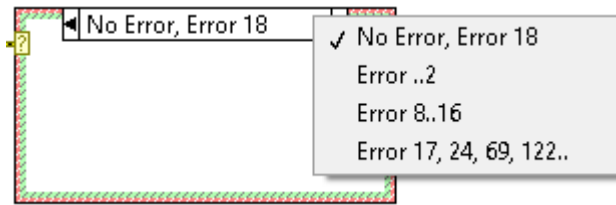
You can define a different data source for the same output tunnel in each case, but the data types must be compatible for each case.

Handling Errors Using Case Structures

Complete the following steps to handle errors using **Case** structures.

1. Select an error cluster terminal.
2. Wire an error cluster to the selector terminal of a Case structure. The case selector label displays two cases—Error and No Error—and the border of the Case structure changes color—red for Error and green for No Error. The Case structure executes the Error subdiagram for all errors.

- (Optional) To configure the case structure to execute a specific subdiagram for specific errors or list of errors, add more cases and [enter error values or ranges in the selector labels](#). You can also add error values to the No Error case. The border of the No Error case is both green and red when the selector label includes any error codes.

**NOTE**

If you add more error cases without customizing the selector label of the Error case, the Error case changes to Default, which handles all the remaining errors that are not covered in existing error cases.

- (Optional) Configure the display format of the error codes in the selector label by right-clicking the structure border and selecting **Radix**. If the display format is not decimal, a glyph representing the format appears on the left side of the selector label.
- Add code to the error case(s) to define behaviors for different cases.

Adding Cases between Case Structure Subdiagrams

Complete the following steps to add a case between existing case subdiagrams in a [Case](#) structure:

- [Add a new case](#) at the end of the order.
- With the new case visible, right-click the structure border and select **Rearrange Cases** from the shortcut menu.
- Select the position for the new case. The subsequent subdiagrams shift down in the order.
- Click the **OK** button to save the changes.

**TIP**

You also can press the <Shift-Enter> keys in the selector label to add a case to a Case structure.

You also can [shift existing cases down in the order](#). Refer to the Case Structure - Selector Data Types VI in the labview\examples\Structures directory for an example of using Case structures.

Specifying a Default Subdiagram for Case Structures

To avoid listing every possible input value, specify a default case subdiagram for the **Case** structure to handle out-of-range values. For example, if the case selector is an integer and you specify subdiagrams for 1, 2, and 3, you must specify a default subdiagram to execute if the input value is 4 or any other unspecified integer value. Refer to the Case Structure - Selector Data Types VI in the `labview\examples\Structures` directory for an example of using Case structures.

Complete the following steps to specify a default subdiagram.

1. **Move to an existing subdiagram.**
2. Right-click the selector label and select **Make This The Default Case** from the shortcut menu to make an existing subdiagram the default subdiagram.

You also can complete the following steps to specify a default subdiagram.

1. **Move to the subdiagram** you want to make default and click on the selector label with the Labeling tool.
2. Enter `Default` in the selector label.



NOTE

Do not add quotation marks around `Default`. Doing so indicates that the value is a string, rather than the default subdiagram.

1. Press the <Enter> key.



Note You cannot specify a default subdiagram if you wire a Boolean control to the case selector. Make the Boolean control TRUE or FALSE to determine which subdiagram to execute.

Swapping Case Structure Subdiagrams

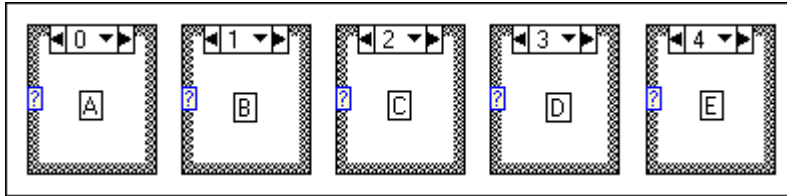
You can swap the visible case subdiagram of a **Case** structure with another case without affecting the other subdiagrams or the order of the cases as they appear in the shortcut menu. Only the subdiagrams of the two cases change. You can swap cases in Case structures of three or more subdiagrams. Refer to the Case Structure - Selector Data Types VI in the `labview\examples\Structures` directory for an example of using Case structures.

Complete the following steps to swap the visible subdiagram with another subdiagram.

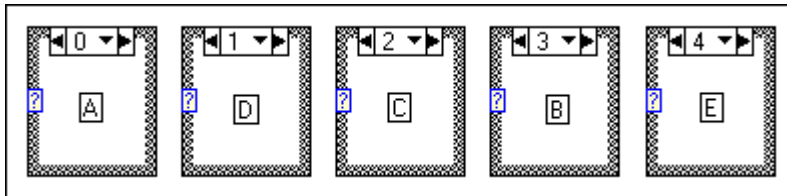
1. **Move to the case** whose subdiagram you want to swap with another subdiagram.
2. Right-click the structure border.

3. Select **Swap Diagram With Case** from the shortcut menu and select the case whose subdiagram you want to swap with the visible subdiagram.

In the following example of subdiagrams in a **Case** structure, case 0 contains subdiagram A, case 1 contains subdiagram B, and so on up to case 4, which contains subdiagram E.



You can swap case 1 with the subdiagram with case 3, causing LabVIEW to swap the subdiagrams, as shown in the following example.



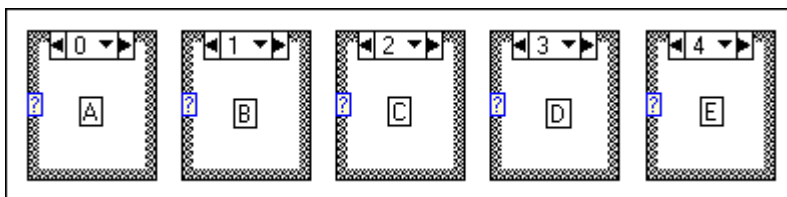
Shifting Case Structure Subdiagrams

You can shift the visible case subdiagram of a **Case** structure to a different position in the case order if the data type wired to the case selector is numeric and is not enumerated. You can shift subdiagrams in Case structures of three or more subdiagrams. Refer to the Case Structure - Selector Data Types VI in the `labview\examples\Structures` directory for an example of using Case structures.

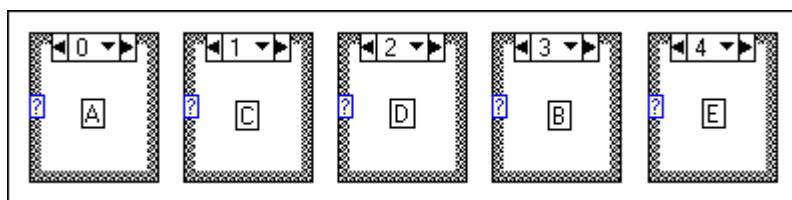
Complete the following steps to shift the visible subdiagram to a different position.

1. **Move to the case** whose subdiagram you want to shift to another position.
2. Right-click the structure border.
3. Select **Shift Diagram To Case** from the shortcut menu and select the case whose subdiagram you want to shift with the visible subdiagram.

In the following example of subdiagrams in a **Case** structure, case 0 contains subdiagram A, case 1 contains subdiagram B, and so on up to case 4, which contains subdiagram E.



You can shift the subdiagram of case 1 to the position of case 3, causing the subdiagrams for cases 2 and 3 to move up in sequence, as shown in the following example.



Passing Data through Frames of a Flat Sequence Structure

If you use a **Flat Sequence** structure, the data from outside the sequence structure enters the frame as each frame executes. The data leaves the frame after the frame executes.

Complete the following steps to pass data from one frame to any subsequent frame in a Flat Sequence structure.

1. Select the terminal you want to wire to the Flat Sequence structure.
2. Wire from the tunnel of one frame to any other frame in the structure. You can wire through a frame or out of one frame and into another frame.

Passing Data through Frames of a Stacked Sequence Structure

Use a sequence local terminal to pass data from one frame to any subsequent frame in a **Stacked Sequence** structure.



NOTE

Stacked Sequence structures do not start to execute until all data wired to the structure arrives. The data wired from each frame leaves only when all the frames complete execution.

Complete the following steps to pass data through the frames of a Stacked Sequence structure by adding a sequence local terminal.

1. **Move to the frame** from which you want to pass data to a subsequent frame.
2. Right-click the right structure border and select **Add Sequence Local** from the shortcut menu.
3. Wire the data you want to pass to the sequence local terminal. An outward-pointing arrow  appears in the terminal.
4. In subsequent frames, wire the sequence local terminal to any terminals where you want to pass the data from the previous frame. The terminals in subsequent frames contain an inward-pointing arrow, indicating that the terminal is a data source for that frame.



Note You cannot use the sequence local terminal in frames that precede the first frame where you wired the sequence local terminal.

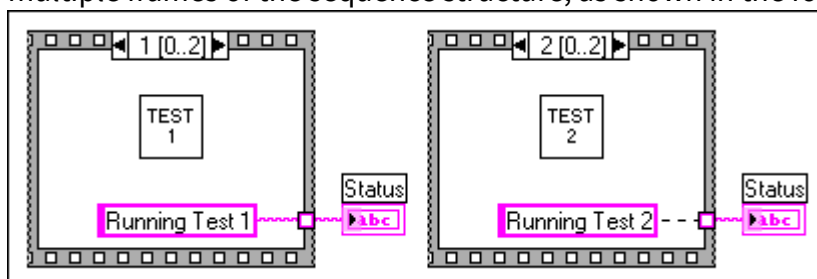
To remove a sequence local terminal, right-click the sequence local terminal and select **Remove** from the shortcut menu.

Determining When to Use Sequence Structures

Sequence structures guarantee the order of execution and prohibit parallel operations. For example, asynchronous tasks that use I/O devices, such as PXI, GPIB, serial ports, and DAQ devices, can run concurrently with other operations if sequence structures do not prevent them from doing so.

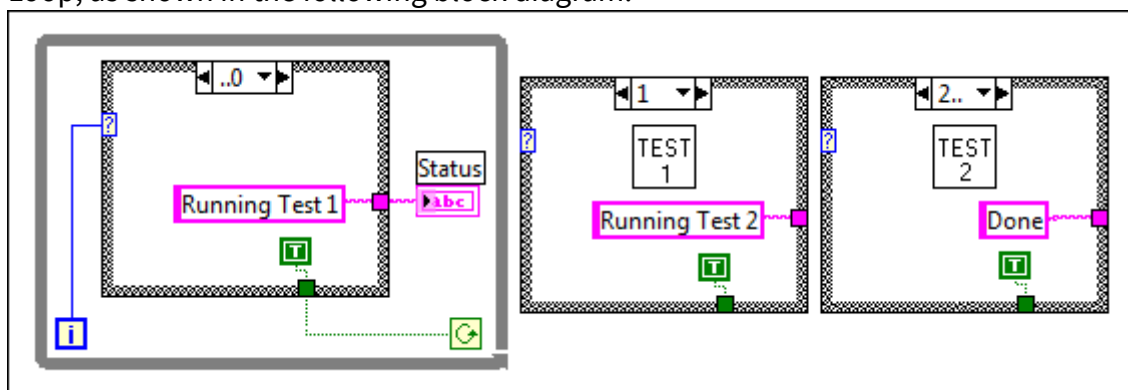
In these cases, you may need to take advantage of the inherent parallelism in LabVIEW. Avoid overusing sequence structures in one of the following ways:

- When you need to control the execution order, consider establishing data dependency between nodes. For example, you can use **flow-through parameters** such as **error I/O** to control the execution order.
- Use a Case structure and a While Loop. You cannot update an indicator from multiple frames of the sequence structure, as shown in the following illustration.



In the previous block

diagram, a VI used in a test application has a **Status** indicator that displays the name of the current test in progress. If each test is a subVI called from a different frame, you cannot update the indicator from each frame, as shown by the broken wire in the Stacked Sequence structure. Because all frames of a Stacked Sequence structure execute before any data passes out of the structure, only one frame can assign a value to the **Status** indicator. Instead, use a Case structure and a While Loop, as shown in the following block diagram.



Each case in the Case structure is equivalent to a sequence structure frame. Each iteration of the While Loop executes the next case. The **Status** indicator displays the

status of the VI for each case. Because data passes out of the structure after each case executes, the **Status** indicator is updated in the case before the one that calls the corresponding subVI. Unlike a sequence structure, a Case structure can pass data to end the While Loop during any case. For example, if an error occurs while running the first test, the Case structure can pass FALSE to the conditional terminal to end the loop. However, a sequence structure must execute all its frames even if an error occurs.

Configuring Conditions for Conditional Disable Structures

A Conditional Disable structure executes one subdiagram depending on the configuration of the condition, which can include the platform and other user-defined symbols.

Complete the following steps to configure conditions for a Conditional Disable structure by selecting symbols and values.

1. Add a Conditional Disable structure to the block diagram.
2. (Optional) If you are using the structure in a LabVIEW project, you can create custom conditional disable symbols.
3. Right-click the structure border and select **Edit Condition For This Subdiagram** to display the Configure Condition dialog box.
4. Select a **Symbol** and enter a **Value** for the symbol. The following table lists the default symbols and valid values for each symbol. **Value** is a case-sensitive string, so you must enter one of the following valid values exactly as it appears below.

Symbol	Valid values	Description
CPU	PowerPC	Specifies the processor on which the subdiagram executes. The VI must be in a LabVIEW project to access this symbol.
	x86	
	null	
	ARM	

Symbol	Valid values	Description
FPGA_EXECUTION_MODE	FPGA_TARGET DEV_COMPUTER_SIM_IO DEV_COMPUTER_REAL_IO THIRD_PARTY_SIMULATION	Specifies to execute a different subdiagram within an FPGA VI depending on whether the execution mode is set to: • FPGA target: FPGA_TARGET • Simulation (Simulated I/O): DEV_COMPUTER_SIM_IO • Simulation (Real I/O): DEV_COMPUTER_REAL_IO • Third-Party Simulation: THIRD_PARTY_SIMULATION The VI must be under an FPGA target in a LabVIEW project to access this symbol.
FPGA_TARGET_FAMILY	VIRTEX2 VIRTEX5 VIRTEX6 SPARTAN3 SPARTAN6 ZYNQ KINTEX7	Specifies to execute a different subdiagram within an FPGA VI depending on the FPGA family, such as Virtex-II or Virtex-5. The VI must be under an FPGA target in a LabVIEW project to access this symbol.
FPGA_TARGET_CLASS	Refer to the Conditional Disable Symbols page of the FPGA Target Properties dialog box to find the value defined by the target.	Specifies the target class of the FPGA target. For example, the FPGA_TARGET_CLASS of the NI PXIe-7965R is PXIE - 7965R and the FPGA_TARGET_FAMILY is VIRTEX5.

Symbol	Valid values	Description
OS	Linux Mac null PharLap VxWorks Win	Specifies the OS on which the subdiagram executes. The VI must be in a LabVIEW project to access this symbol.
RUN_TIME_ENGINE	True False	Specifies whether the subdiagram executes when you create a LabVIEW stand-alone application or shared library that uses the LabVIEW Run-Time Engine. RUN_TIME_ENGINE is True for Build Specifications of type Application, Real-Time Application, Shared Library, and .NET Framework Assembly. RUN_TIME_ENGINE is False for Build Specifications of type Packed Library. RUN_TIME_ENGINE is not applicable to Build Specifications of type Installer, Package, Source Distribution, and Zip File.
TARGET_BITNESS	32 64	Specifies the bitness of the instance of LabVIEW or the LabVIEW Run-time Engine that executes the subdiagram.

Symbol	Valid values	Description
TARGET_TYPE	Windows	Specifies on which platforms or which targets the subdiagram executes.
	FPGA	
	IPBuilder	
	Embedded	
	RT	
	Mac	
	Unix	
	PocketPC	
	DSP	



NOTE

If you do not use the Conditional Disable structure in a LabVIEW project, RUN_TIME_ENGINE, TARGET_BITNESS, and TARGET_TYPE are the only symbols available.

- 5. (Optional) Click the plus sign to add another condition. For example, if you want the subdiagram to apply to multiple platforms, such as Windows and Mac, you can set the **Value** of the TARGET_TYPE symbol to *Windows* in one condition, click the plus sign, select the **OR** operator, and set the **Value** of the TARGET_TYPE symbol to *Mac* in the second condition.
- 6. Click the **OK** button.
- 7. Add objects inside the Conditional Disable structure to create subdiagrams that the Conditional Disable structure can execute. If necessary, add or duplicate subdiagrams.



NOTE

You can join conditions through Boolean operators to create an expression that allows LabVIEW to evaluate multiple conditions. If more than one condition evaluates to TRUE, the first condition that evaluates to TRUE determines the active subdiagram for the structure. If you define a Boolean expression for a subdiagram, the symbol/value comparisons that you join through the AND operator are evaluated before symbol/value comparisons you join with the OR operator.

If you want to disable a section of code on the block diagram so that it does not compile at run time, use the Diagram Disable structure.

Refer to the Conditional Disable Structure VI in the labview\examples\Structures\Disable Structures directory for an example of using Conditional Disable structures.

Customizing Conditions for Conditional Disable Structures

If you are using a **Conditional Disable** structure in a **LabVIEW project**, you can define custom conditional disable symbols in LabVIEW. Conditional disable symbols consist of a symbol name and a symbol value. Comparing the symbol name and the symbol value creates a condition that must be TRUE for LabVIEW to compile and execute the subdiagram.

Complete the following steps to customize conditions by creating conditional disable symbols.

1. Select **Project»Properties** to display the Project Properties dialog box.
2. Select **Conditional Disable Symbols** from the **Category** list to display the Conditional Disable Symbols page.
3. Enter a name for the symbol you want to create in the **New Symbol** text box. You might choose names for the symbols to indicate the function they perform. The name of the symbol is case sensitive.
4. Enter the value you want the symbol to represent in the **New Value** text box.



NOTE

Values are case-sensitive strings, which means that LabVIEW handles numerical values you enter as strings rather than numbers. For example, if you define the value of a custom symbol as 0 . 0 and then configure the condition to 0 in the Configure Condition dialog box, that condition evaluates to false.

5. Click the **Add** button to add the new symbol to the conditional disable symbols list.
6. Click the **OK** button to close the **Conditional Disable Symbols** page. You now can [use the symbols](#) you just defined in the Conditional Disable structure.

You also can create conditional disable symbols on a target level in the appropriate **Target Properties** dialog box. For example, if you are using an FPGA target, you can create the conditional disable symbol in the **FPGA Target Properties** dialog box.

If the same symbol is defined on both a project level and a target level in the same project, the Conditional Disable structure recognizes only the symbol defined for the target.

Reducing Data Copies with In Place Element Structures

The **In Place Element** structure performs common data manipulation operations in a manner that does not require the LabVIEW compiler to make multiple data value copies or maintain those values in memory.

Complete the following steps to perform data manipulations by configuring an In Place Element structure.

1. Add an In Place Element structure to the block diagram.

2. Right-click the border of the In Place Element structure and select the border node you want to use. Each set of border nodes includes a border node on the left side of the structure and a border node on the right side of the structure.
3. *Add objects inside the In Place Element structure* to manipulate the array, cluster, waveform, or variant element.
4. Wire all required terminals on the border nodes. Otherwise, the VI is broken and will not run.

You can add multiple border nodes of various types to a single In Place Element structure.

Caveats and Recommendations for Using In Place Element Structures

The following list describes some of the caveats and recommendations to consider when you use the *In Place Element* structure:

- When you unbundle a cluster with the Unbundle / Bundle Elements border node of the In Place Element structure, you cannot unbundle any cluster element more than one time. For example, if the cluster you are unbundling contains another cluster and you unbundle an element of the subcluster, you cannot also unbundle the entire subcluster because you already unbundled one of the elements of the subcluster.
- Do not use more than one border node for a single array, cluster, variant, or waveform. Operate on each individual item with a single border node. If you use more than one border node for a single array, cluster, variant, or waveform, a copy of the entire array, cluster, variant, or waveform is placed in memory, which negates the positive performance effects of the In Place Element structure.
- Because variants can contain data of multiple types, you must wire the data type of the data you want to replace from the variant to the **data type** input of the Variant To / From Element border node. If the variant does not contain data of the type wired to the **data type** input, the VI returns an error.
- You can place an In Place Element structure inside another In Place Element structure. Nested In Place Element structures allow you to access elements of multiple complex data types. For example, if you want to access an element of an array within a cluster, use the outer In Place Element structure to unbundle the cluster, and use the inner In Place Element structure to access an element of the array.
- Consider using the **Mark As Modifier** shortcut menu option on nodes of the In Place Element structure to indicate that LabVIEW will modify the value of the data you wire to the node. For example, *dynamic dispatch terminals* can execute any implementation of a dynamic dispatch subVI in a class hierarchy. The parent implementation of the dynamic dispatch VI might have no operation and does not modify the data; however, a child implementation might modify the data. If you wire the parent implementation to the node and select the **Mark As Modifier** option, you can indicate that LabVIEW will modify the value wired to a dynamic dispatch terminal at some point in the class hierarchy. Using the **Mark As Modifier** option then optimizes performance by minimizing the number of copies of the data LabVIEW creates.

Adding and Deleting Timed Structure Frames (Real-Time, Windows)

Right-click a timed structure border and select **Add Frame After** or **Before** from the shortcut menu to add a frame. You also can right-click the border between two frames and select **Insert Frame** from the shortcut menu to insert a frame between two frames.

To delete a timed structure frame, right-click the bottom or top structure border of the frame and select **Delete This Frame** from the shortcut menu. If only one frame exists, you cannot delete the frame.



Merging Timed Structure Frames (Real-Time, Windows)

To merge two timed structure frames, right-click the structure border between both frames and select **Merge Frames** from the shortcut menu.

If you merge frames that have values wired to the Right Data node of the first frame or the left data node of the second frame, LabVIEW shows the values with broken wires.



Displaying and Hiding the Terminals of a Timed Structure Node (Real-Time, Windows)

You can wire values to the Input and Right Data nodes of a timed structure to configure the options of the structure. The structures also return different output values from the Left Data and Output nodes.

By default, a timed structure only displays the **Error** input or output on the Input, Output, and Data nodes. Right-click an Input, Output, or Data node and select **Show All Inputs** or **Show All Outputs** to display all the inputs or outputs of the node.

You also can display only selected inputs or outputs on a node. Right-click an Input, Output, or Data node and select **Add Input** or **Add Output** from the shortcut menu to display the next available input or output on the node. Right-click the new input or output and select **Select Input** or **Select Output** from the shortcut menu. The shortcut menu displays the currently selected input or output with a checkmark. Select the input that you want to display.

Right-click an input or output on a node and select **Remove Input** or **Remove Output** from the shortcut menu to hide the input or output on the node.

Refer to the Timed Loop and Timed Sequence topics for reference information about the inputs and outputs for the nodes of a timed structure.



Events

Adding and Duplicating Event Cases

Complete the following steps to add or duplicate an event case.

1. Right-click the Event structure border.
2. Select **Add Event Case** or **Duplicate Event Case** from the shortcut menu. The Edit Events dialog box appears.
3. [Edit the event case.](#)

When you duplicate an event case, LabVIEW does not duplicate the events you configured for that case. You must [configure the new case](#) to handle an event. If you configure multiple cases to handle the same event, the VI appears broken.

Configuring Events Handled by the Event Structure

Before you configure events for the Event structure to handle, review the [caveats and recommendations for using events](#) in LabVIEW.

Complete the following steps to configure an Event structure case to handle an event.

1. (Optional) If you want to configure the Event structure to handle a user event, a Boolean control within a radio buttons control, or a user interface event that is generated based on a reference to an application, VI, or control, you first must [dynamically register that event](#).
2. Right-click the border of the Event structure and select **Edit Events Handled by This Case** from the shortcut menu to display the Edit Events dialog box to edit the current case. You also can select **Add Event Case** from the shortcut menu to [create a new case](#).
3. Specify an event source in the **Event Sources** pane.
4. Select the event you want to configure for the event source, such as Key Down, Timeout, or Value Change, from the **Events** list. When you select a dynamic event source from the **Event Sources** list, the **Events** list displays that event. This is the same event you selected when you [registered the event](#). If you registered for events dynamically and wired **event reg refnum out** to the dynamic event terminal, the sources appear in the **Dynamic** section.
5. If you want to add additional events for the current case to handle, click the blue plus button  and repeat steps 3 and 4 to specify each additional event. The **Event**

Specifiers section at the top of the dialog box lists all the events for the case to handle. When you click an item in this list, the **Event Sources** section updates to highlight the event source you selected. You can repeat steps 3 and 4 to redefine each event or click the red X button  to remove the selected event.

6. Click the **OK** button to save the configuration and close the dialog box. The event cases you configured appear as selection options in the event selector label at the top of the event structure and the Event Data node displays the data common to all events handled in that case.
7. (Optional) You can use a Timeout event to configure an Event Structure to wait a specified amount of time for an event to occur. Wire a value to the Timeout terminal at the top left of the Event structure to specify the number of milliseconds the Event structure should wait for an event to occur before generating a Timeout event. The default value for the Timeout terminal is -1, which specifies to wait indefinitely for an event to occur.
8. Repeat steps 1 through 6 for each event case you want to configure.

Refer to the following VIs for examples of using events:

- labview\examples\Dialog and User Interface\Events\Events.lvproj
- labview\examples\Structures\Event Structure\Event Structure.lvproj

Creating Custom Events

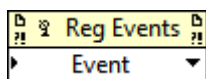
You can create and name custom events, called user events, to carry data that you define. User events allow different parts of an application to *communicate asynchronously*. Unlike *user interface events*, which require direct user interaction with front panel objects, user events allow you to create an application that responds to programmatic changes on objects. You can handle both user interface events and programmatically generated user events in the same Event structure.

What to Use

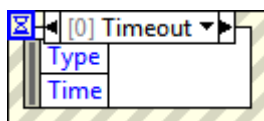
Use the following block diagram objects to create a user event.



—Create User Event



—Register For Events



—Event Structure



Generate User Event



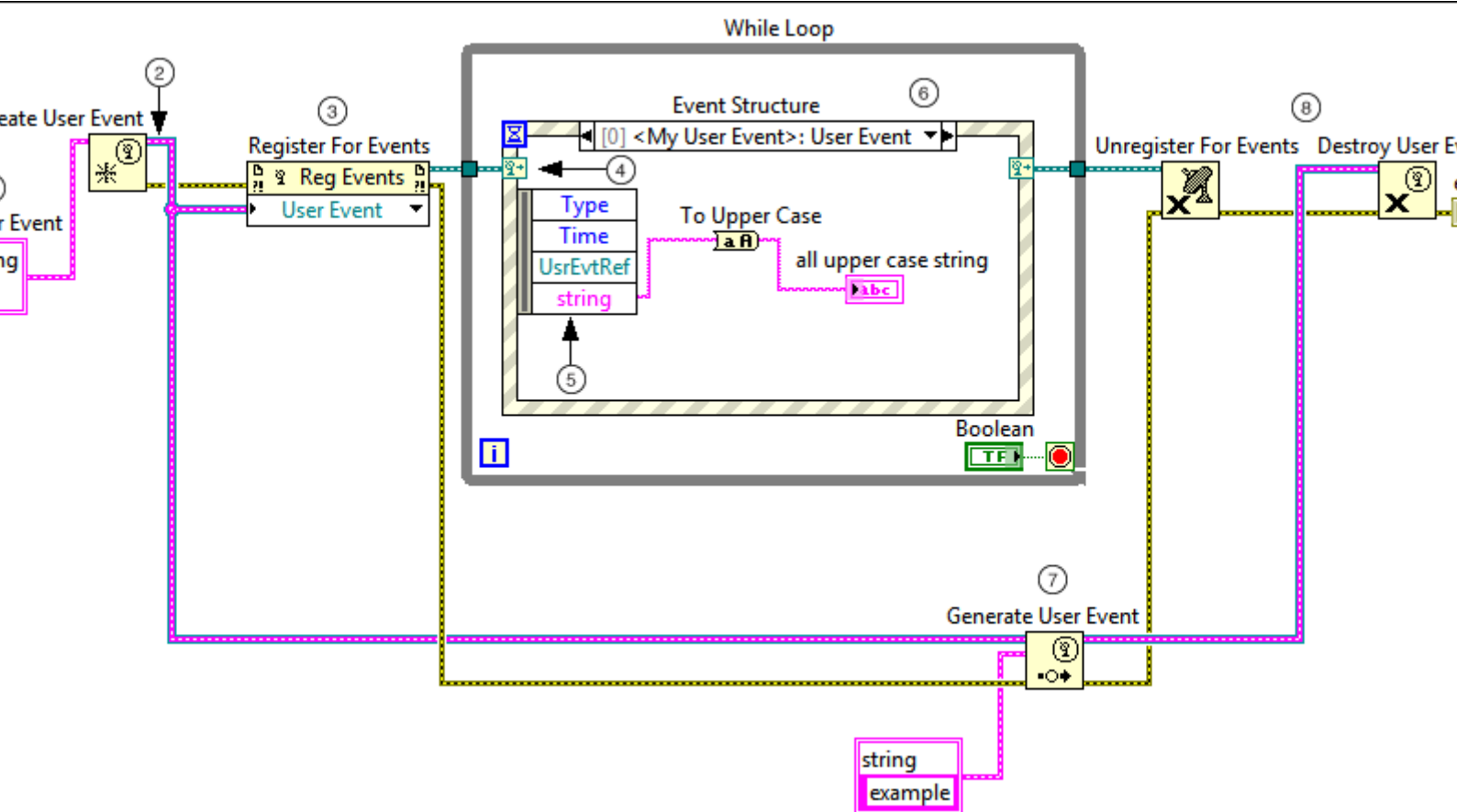
Unregister For Events



Destroy User Event

What to Do

Create the following block diagram to build a user event that programmatically carries data you define.



The following list describes important details about the previous diagram.

1 —To define a user event, wire a block diagram object to the Create User Event function. The input for the Create User Event function is an individual element or a cluster whose name becomes the name of the user event and whose data type defines the data type for the event. Furthermore, LabVIEW uses *owned labels* to identify the data the user event carries, so label the input data—or each element of the input data—with a descriptive owned label. You must provide a label on the input data for the user event to appear in the

Event Data Node of an Event structure. In the previous block diagram, the name of the user event is `My User Event`, and the name of the event data item is `string`.

2 — The **user event out** output of the Create User Event function is a strictly typed refnum that carries the name and data type of the user event. Use this refnum as a reference to the event name and data type when registering for events and generating a user event.

3 — Use the Register For Events function to *dynamically register* the user event. You can wire a combination of user events and user interface events to the Register For Events function.

4 — Wire the **event registration refnum** output of the Register For Events function to the dynamic event terminal on the left side of the Event structure. To display the dynamic event terminal, right-click the Event structure and select **Show Dynamic Event Terminals**. The user event data items appear in the Event Data Node on the left border of the Event structure.

5 — To add more data items to the Event Data Node, vertically resize the node. Use the Event Data Node to access any event data element associated with a particular event.

6 — Use the Edit Events dialog box to configure a case in the Event structure to handle the event. The name of the user event appears under the **Dynamic** subheading in the **Event Sources** section of the dialog box.

7 — Use the Generate User Event function to deliver the user event and associated data to other parts of an application through an Event structure configured to handle the event. The Generate User Event function accepts a user event refnum and a value for the event data. The data value must match the data type of the user event.



NOTE

If the user event is not registered, the Generate User Event function has no effect. If the user event is registered but no Event structure is waiting on it, LabVIEW queues the user event and data until an Event structure executes to handle the event.

8 — Use the Unregister For Events function and the Destroy User Event function to release the associated events and the user event reference. LabVIEW unregisters all events and destroys existing user events automatically when the top-level VI finishes running. However, to conserve memory resources, National Instruments recommends that you unregister and destroy user events explicitly, especially in an application that runs for a long period of time.

Caveats and Recommendations

- You cannot register for a user event statically.
- You can register for the same user event multiple times by using separate Register For Event functions. In this situation, each queue associated with an event registration refnum receives a copy of the user event and associated event data each time the Generate User Event function executes.
- User events are *notify events* and can share the same event case of an Event structure as user interface events or other user events.

- To simulate user interaction with a front panel, you can create a user event that has event data items with the same names and data types as an existing user interface event. For example, you can create a user event called `MyValChg` by using a cluster of two Boolean fields named `OldVal` and `NewVal`, which are the same event data items the Value Change user interface event associates with a Boolean control. You can share the same Event structure case for the simulated `MyValChg` user event and a real Boolean Value Change event. The Event structure executes the event case if a Generate User Event function generates the user event or if a user changes the value of the control.

Examples

Refer to the `labview\examples\Dialog and User Interface\Events\Events.lvproj` for examples of dynamically registering events.

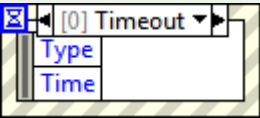
Refer to the User Event Generation VI in the `labview\examples\Dialog and User Interface\Events` directory for an example of using user events.

Dynamically Registering for Events

Dynamic event registration allows you to use VI Server references, such as Application, VI, or Control, to specify at run time the objects for which you want LabVIEW to generate events.

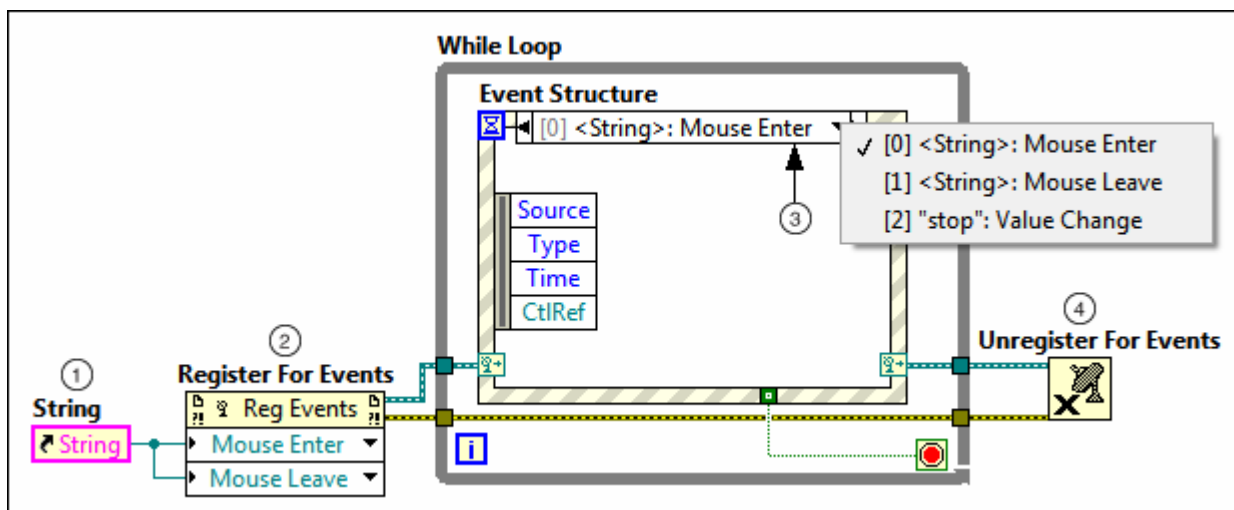
What to Use

Use the following block diagram objects to dynamically register events.

Register For Events	While Loop	Event Structure	Unregister For Events
			

What to Do

Create the following block diagram to dynamically register the Mouse Enter and Mouse Leave events on a string control.



The following list describes important details about the previous diagram.

①

Obtain [VI Server](#) references to the objects for which you want LabVIEW to generate events. To obtain a control reference constant, like the string control constant from the previous illustration, use a Property Node to query the VI for its controls or right-click the control and select **Create»Reference** from the shortcut menu.

②

Use the Register For Events function to dynamically register for events. You can register for multiple events for a single reference by resizing this function and selecting another event. Click the down arrow on the event source input and select the event you want LabVIEW to generate for the reference, such as the Mouse Enter and Mouse Leave events shown in the previous illustration. After registration, LabVIEW queues events as they occur until an Event structure executes to handle the events.



TIP

You can view the registered events in the **Context Help** window when you hover over the event registration refnum.

③

Right-click the event selector label to display the **Edit Events** dialog box. Use the dialog box to [configure an Event structure](#) to handle a dynamically registered event. The **Event Sources** section of the dialog box contains a **Dynamic** subheading that lists each dynamically registered event source. After you select the events you want the Event structure to handle, design the case to handle the event data.

④

Use the Unregister For Events function to stop event generation. When the Unregister For Events function executes, LabVIEW unregisters all events the event registration refnum specifies, destroys the event queue associated with the event registration refnum, and discards any events that remain in the queue.

Caveats and Recommendations for Dynamically Registering Events

- Wire the **event registration refnum** input of the Register For Events function only when you want to [modify an existing registration](#).

- Wire each Event structure that handles dynamic events to a unique Register For Events function. Branching the event registration refnum wire of a Register For Events function allows multiple Event structures to pull events from one queue, resulting in a race condition that may cause unpredictable behavior.
- Before you configure events for the Event structure to handle, review the [caveats and recommendations for using events](#) in LabVIEW.

Examples

Refer to the `labview\examples\Dialog and User Interface\Events\Events.lvproj` for examples of dynamically registering events.

Dynamically Modifying Event Registration

If you [register for events dynamically](#), you can modify the registration information to change when LabVIEW registers events and the objects for which LabVIEW generates events. For example, you can dynamically register for a Mouse Move event on a front panel control, and you can modify the registration so that LabVIEW generates this event only if the user clicks a certain button first.



NOTE

You cannot modify statically registered events at run time.

Complete the following steps to modify registration information:

1. [Create a placeholder for the event you want to occur only under certain circumstances.](#)
2. [Design an event case to specify when you want LabVIEW to register for the event.](#)
3. [Unregister for the event before you unregister for all events.](#)

Examples

Refer to the `labview\examples\Dialog and User Interface\Events\Events.lvproj` for examples of dynamically registering events.

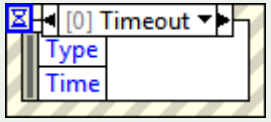
Step 1: Create a Placeholder for the Event You Want to Occur Only Under Certain Circumstances

Use a placeholder to register for an event you want to monitor once a specific condition occurs in the application. The placeholder creates an entry in the Register For Events function for a specific event but does not allow LabVIEW to generate that event for the placeholder object. When you wire the refnum output of a Register For Events function to the refnum input of another Register For Events function, the function that accepts the refnum contains the existing registration information. Therefore, you can modify the

registration information in the Register For Events function by replacing the placeholder object with a valid object reference.

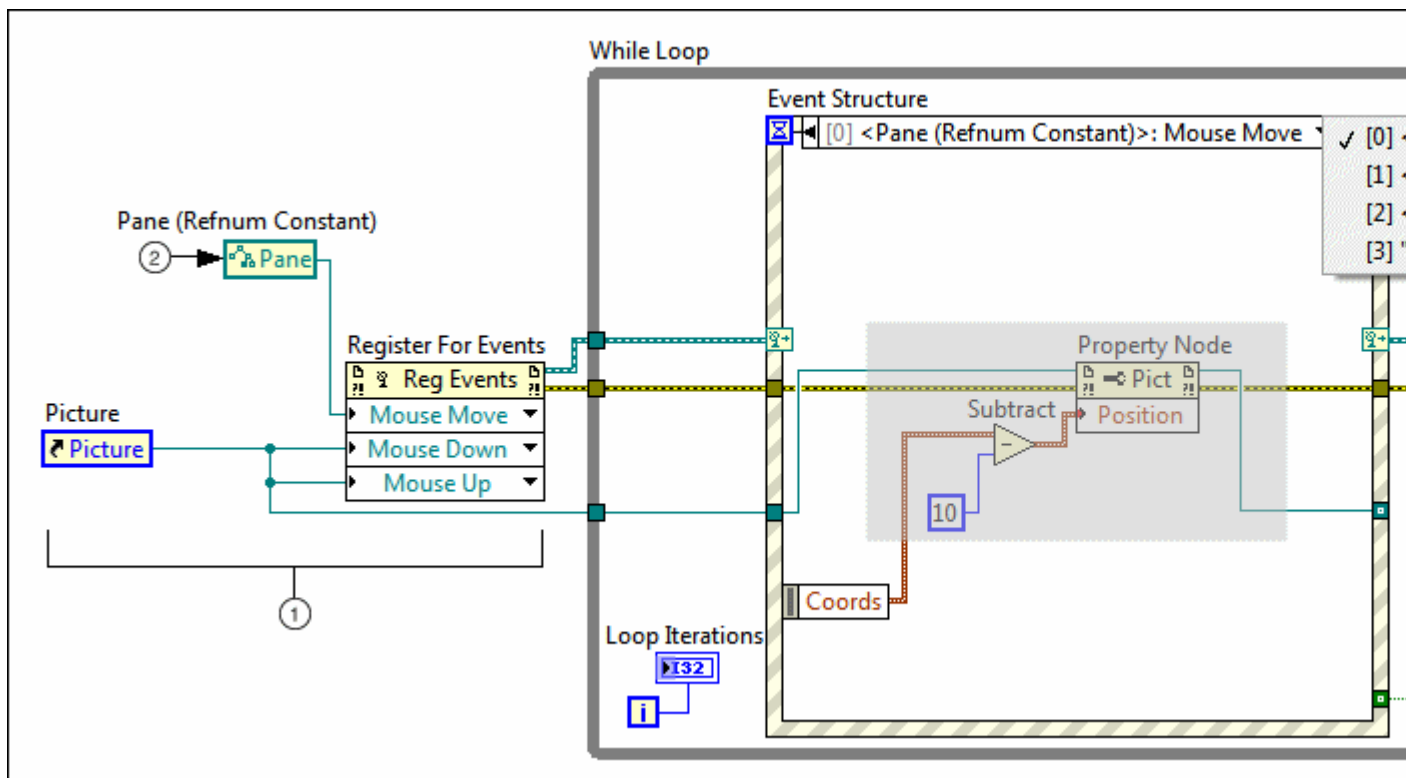
What to Use

Use the following block diagram objects to modify event registration dynamically.

VI Server Reference	Register For Events	While Loop	Event Structure
			

What to Do

Create the following block diagram to dynamically register for events and to create a placeholder for the event you want to register at a different location in the application. This example modifies registration dynamically so that the Mouse Move event on the VI pane triggers only when the user clicks the picture control on the front panel.



The following list describes important details about the previous step.

- ① **Dynamically register** the Mouse Down and Mouse Up events for the picture control reference.

2

To create a placeholder for an event, you can wire a refnum constant to the Register For Events function. Because the value of a refnum constant is Not a Refnum, LabVIEW registers the Mouse Move event on an invalid object. Therefore, no Mouse Move event occurs until data flow allows LabVIEW to register the event on a valid object. In the previous block diagram, a strictly-typed refnum constant for the pane of the current VI is wired to the Mouse Move event. To create this refnum constant, you can click a VI Server reference and select **Pane** from the shortcut menu. Right-click the Pane reference and select **Create»Constant** from the shortcut menu.

After you register a placeholder for the event you want to occur only under certain circumstances, *design an event case to specify when you want LabVIEW to register for the event.*

Step 2: Design an Event Case to Specify When You Want LabVIEW to Register for the Event

In this example, the Mouse Down event case contains the Register For Events function that registers the Mouse Move event. When the user clicks the picture control on the front panel, the Event structure executes to handle the Mouse Down event and the code in the Mouse Down event case executes to register the Mouse Move event on the pane reference.

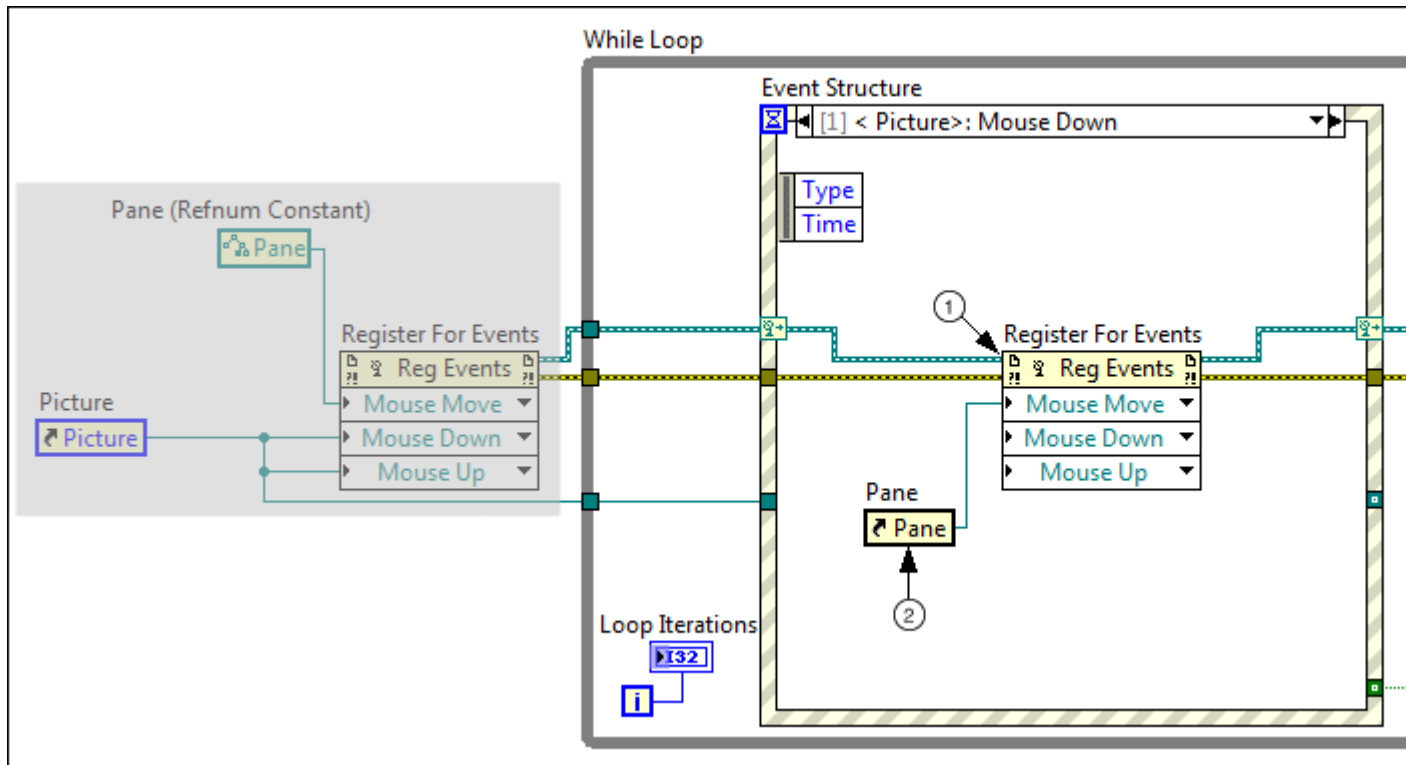
What to Use

Use the following block diagram objects to modify event registration dynamically.

VI Server Reference	Register For Events
	

What to Do

Create the following block diagram so LabVIEW registers the Mouse Move event on a valid object at run time.



The following list describes important details about the previous step.

- ① To modify the existing registration, wire the **event registration refnum** output of a Register For Events function to the top left input of another Register For Events function. When you wire the **event registration refnum** input, the function automatically resizes to show the same events on the same types of references as you specified in the Register For Events function that originally created the event registration refnum. You cannot manually resize or reconfigure the function while the **event registration refnum** input is wired.
- ② Wire the Pane reference to the Mouse Move event to replace the registered placeholder. When the user clicks the picture control, the Event structure executes to handle the Mouse Down event and LabVIEW registers the Mouse Move event.

After you design an event case to specify when you want LabVIEW to register for the event, *unregister for the event before you unregister for all events.*

Step 3: Unregister for the Event before You Unregister for All Events

After LabVIEW registers for the event you want to occur only under certain circumstances, LabVIEW continues to generate that event until you specify to unregister for the event. You can design an event case to specify under what circumstances you want LabVIEW to unregister for the event. In this example, LabVIEW registers the Mouse Move event when the user clicks the picture control on the front panel and unregisters the Mouse Move event when the user releases the mouse button.

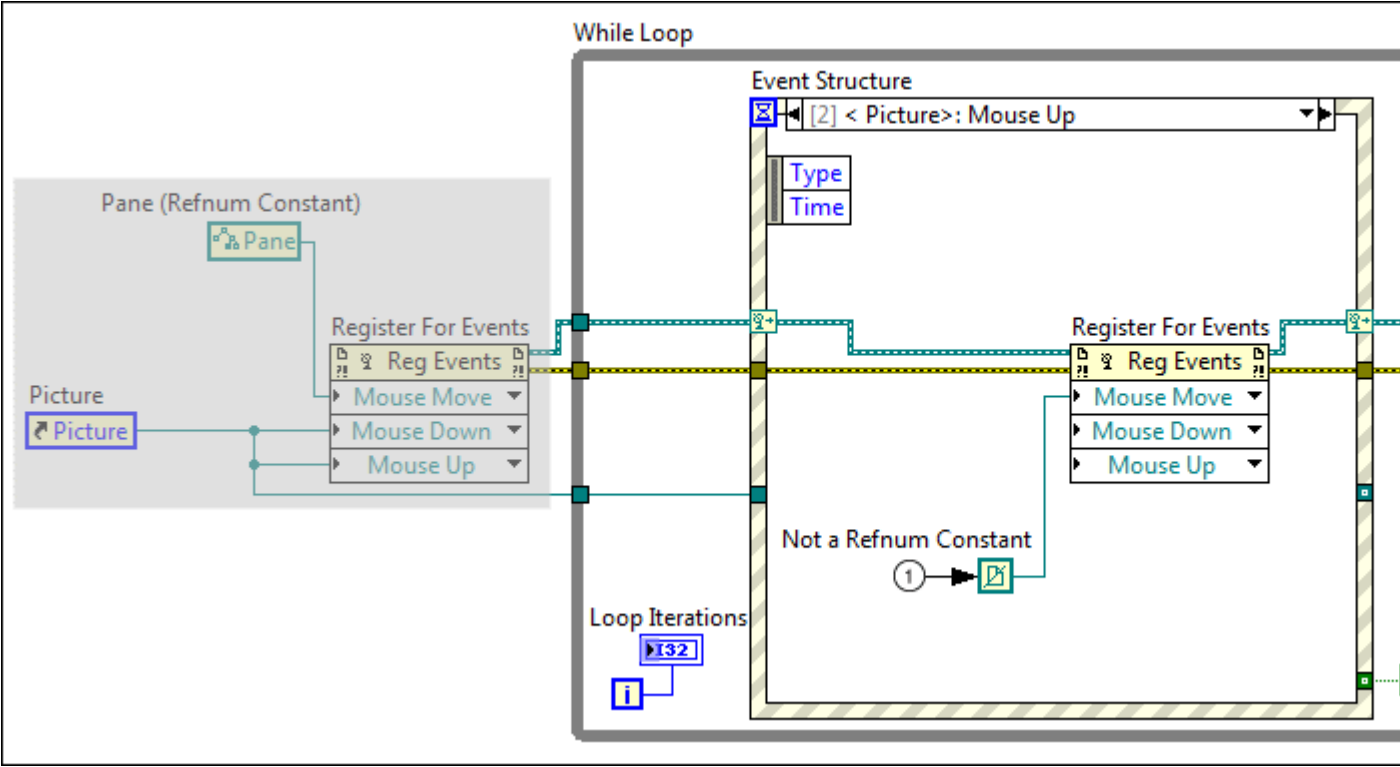
What to Use

Use the following block diagram objects to modify event registration dynamically.

Not a Refnum Constant	Register For Events	Unregister For Events
		

What to Do

Create the following block diagram to unregister the Mouse Move event so LabVIEW does not generate this event unless the Mouse Down event occurs.



The following list describes important details about the previous step.

①	Use the Not a Refnum constant to unregister the Mouse Move event when the user releases the mouse button on the picture control. Unregistering for this event stops LabVIEW from generating the event again unless the Event structure executes to handle the Mouse Down event case, which contains the code to register the Mouse Move event.
②	Unregister for all events associated with the event registration refnum.

Caveats and Recommendations

- To confirm that the Event structure executes to handle each event, you can wire an indicator to the iteration terminal on the While Loop to check the iteration count at run time.
- When designing event cases for the Event structure, use a latched stop Boolean control to determine when to **stop** the VI.

Finding a Control Associated with an Event Case

Complete the following steps to find the control with which an event case is associated.

1. Right-click the Event structure border.
2. Select **Find Control** from the shortcut menu. LabVIEW highlights the front panel control.

Locking Front Panels

By default, when an event enters a queue, LabVIEW locks the front panel that contains the object that generated that event. LabVIEW keeps the front panel locked until all Event structures finish handling the event. While the front panel is locked, LabVIEW does not process front panel activity but places those interactions in a buffer and handles them when the front panel is unlocked. For example, if an event case launches an application that requires text entry from the user, the user might begin typing before the application appears on the front panel. If the **Lock front panel until the event case for this event completes** option is enabled, once the application launches and appears on the front panel, it processes the key presses in the order in which they occurred. If the **Lock front panel until the event case for this event completes** option is disabled, the key presses might be processed elsewhere on the front panel, since LabVIEW does not queue their execution to depend on the completion of the event case.



NOTE

If an event case that locks the front panel takes a significant amount of time to execute, consider using the Set Busy function or [displaying a dialog box](#) to notify the user that the front panel is locked.

Front panel locking does not affect certain actions, such as moving the window, interacting with the scroll bars, and clicking the **Abort Execution** button.



Caution If no Event structure executes promptly to handle an event and front panel locking is enabled, the user interface of the VI may become unresponsive. If this occurs, click the **Abort Execution** button to stop the VI. You can disable front panel locking by right-clicking the Event structure and removing the checkmark from the **Lock front panel until the event case for this event completes** checkbox in the Edit Events dialog box.

You can disable front panel locking for notify events so that when an event enters a queue, all controls, including the one that generated the event, still can generate events. When multiple events occur, LabVIEW does not wait until the front panel is unlocked to handle them. Complete the following steps to disable front panel locking.

1. Right-click the Event structure border and select Edit Events Handled by This Case from the shortcut menu.
2. Remove the checkmark from the **Lock front panel until the event case for this event completes** checkbox.



Note You cannot disable this option for filter events, because LabVIEW cannot finish processing a filter event until its case completes. Allowing LabVIEW to process other events while it is handling a filter event could cause events to occur out of order.

Viewing Enqueued Events at Run Time

Complete the following steps to view the events waiting in an event queue:

1. Run the VI.
2. Right-click the Event structure border.
3. Select Event Inspector Window from the shortcut menu. The **Unhandled Events in Event Queue(s)** table displays events waiting in the queue. After the Event structure handles an event, the event appears in the **Event Log**.
4. (Optional) Use the **Active VIs with Event Structures** pull-down menu to view the names of running VIs that include Event structures registered for events or Event structures that handled events since you opened the window.
5. (Optional) Click **Save Log to File** to save the **Event Log** to a text file.
6. (Optional) Click **Clear All** to clear the **Event Logs** for all running VIs with Event structures and remove the **Event Logs** for Event structures no longer running. Click **Clear** to clear only the **Event Log** for the current running VI.

Caveats and Recommendations when Using Events in LabVIEW

Because LabVIEW is a graphical programming interface, handling events is different from handling events in other programming languages. The following list describes some of the caveats and recommendations when incorporating events into LabVIEW applications:

- Ensure that Event structures can handle events whenever events occur.
- Remember to read the terminal of a latched Boolean control in its Value Change event case.
- Use a Case structure to handle undo operations for a latched Boolean control.

- Use caution when you configure one case to handle multiple notify events.
- You cannot configure one case to handle multiple filter events with different event data.
- If a While Loop that contains an Event structure terminates based on the value of a latched stop Boolean control, remember to handle the latched stop Boolean control in the Event structure.
- Consider using the Wait for Front Panel Activity function if you do not need to monitor specific front panel objects programmatically.
- User interface events apply only to direct user interaction.
- Avoid using dialog boxes in an event case with the Mouse Down? filter event.
- Avoid placing two Event structures in one loop.
- Avoid using multiple Event structures in the same VI.
- When using dynamic registration, make sure you have a Register For Events function for each Event structure.
- If you register for the same event on both a VI and a Control class, LabVIEW generates the VI event first.
- When using subpanel controls, the top-level VI containing the subpanel control handles the event.
- If you need to generate or handle other events while handling the current event, consider using the Register Event Callback function.
- Use caution when selecting between a notify or filter event. An event case configured to handle a notify event cannot affect if or how LabVIEW processes a user interaction. If you want to modify if or how LabVIEW processes a user interaction, use a filter event.
- Do not use the Panel Close notify event for important shutdown code unless you have taken steps to ensure that the VI does not abort when the panel closes. For example, be sure the application opens a reference to the VI before a user can close the front panel. Alternatively, you can use the Panel Close? filter event, which occurs before the panel actually closes.
- If no Event structure executes promptly to handle an event and front panel locking is enabled, the user interface of the VI may become unresponsive. You must ensure

that the Event structure is in a loop that runs often enough to respond to user interface events in a timely manner or disable front panel locking.



RELATED INFORMATION

[Ensure That Event Structures Handle Events whenever Events Occur](#) on page 1323

[Using Events with Latched Boolean Controls](#) on page 1328

[Using Events with Latched Boolean Controls](#) on page 1328

[One Case Handling Multiple Notify Events](#) on page 1327

[One Case Handling Multiple Filter Events](#) on page 1326

[Handling a Latched Stop Boolean Control in an Event Structure](#) on page 1325

[Using Events versus Using the Wait for Front Panel Activity Function](#) on page 1328

[Use Events for Direct User Interaction Only](#) on page 1327

[Prompting Users with Dialog Boxes in a Mouse Down? Event Case](#) on page 1327

[Avoid Placing Two Event Structures in One Loop](#) on page 1322

[Use Only One Event Structure whenever Possible](#) on page 1327

[Dynamically Registering for Events](#) on page 1312

[Using Events with Subpanel Controls](#) on page 1328

[Determining Which Type of User Interface Events to Use](#) on page 453

[Dynamically Loading and Calling VIs](#) on page 245

[Locking Front Panels](#) on page 1319

[Types of Events](#) on page 451

Avoid Placing Two Event Structures in One Loop

National Instruments recommends that you place only one Event structure in a loop. When an event occurs in this configuration, the Event structure handles the event, the loop iterates, and the Event structure waits for the next event to occur. If you place two Event structures in a single loop, the loop cannot iterate until both Event structures handle an event. If you have enabled front panel locking for the Event structures, the user interface of the VI can become unresponsive depending on how the user interacts with the front panel.

For example, if you place two Event structures in a single While Loop and configure the first Event structure to handle a Mouse Down event and configure the second Event structure to handle a Key Down event, the first Event structure receives a Mouse Down event when the user clicks the mouse button. The first Event structure executes the correct event case and finishes execution. Meanwhile, the second Event structure waits for a key press to occur. When the user presses a key, the second Event structure receives a Key Down event. After the second Event structure handles the event, the While Loop iterates. If the user continues to alternate interactions, generating a Mouse Down event, a Key Down event, a Mouse Down event, a Key Down event, and so on, the VI runs smoothly because the Event structures handle the events as they occur, and the While Loop continues to iterate.

However, if the user clicks the mouse button twice, causing two Mouse Down events to occur sequentially with no intervening Key Down event, the user interface hangs. The first time the user clicks the mouse button, the first Event structure receives a Mouse Down event, handles the event, and finishes execution. However, the second Event structure continues to wait for a Key Down event, and prevents the While Loop from iterating again. When the user clicks the mouse button a second time, LabVIEW generates a second Mouse Down event and locks the front panel until the first Event structure handles the event. At this point, the VI is in a deadlock state. The first Event structure cannot execute until the While Loop iterates, and the While Loop cannot iterate until the second Event structure receives and handles a Key Down event. Because the front panel is locked, no Key Down events can occur. The front panel remains locked and unresponsive until the user aborts the VI.

To avoid hanging the user interface with front panel locking, [configure all events](#) you want a VI to handle in a single Event structure or always make sure there is only one Event structure in a While Loop. Additionally, make sure there is always an [Event structure available](#) to handle events as they occur.

Ensure That Event Structures Handle Events whenever Events Occur

LabVIEW begins queuing events when the VI runs or is reserved to run. The Event structure handles a queued event when [data flow](#) allows the structure to execute. Therefore, LabVIEW can generate events before an Event structure is waiting to handle them.

Because LabVIEW queues events for the entire time that a VI is running, LabVIEW can queue events after dataflow causes an Event structure to finish executing. For example, the loop containing the Event structure may terminate while the VI continues to run. To ensure that every queued event is handled, [design the VI to finish running](#) soon after the Event structure stops executing.

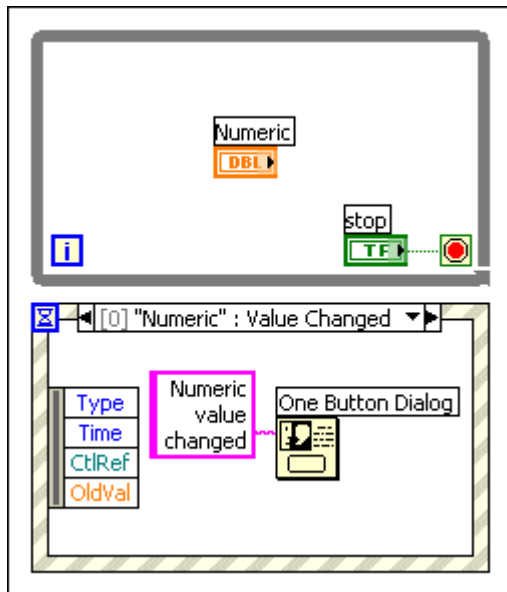


CAUTION

If no Event structure executes promptly to handle an event and [front panel locking](#) is enabled, the user interface of the VI may become unresponsive. If this occurs, click the **Abort** button to stop the VI. You can disable front panel locking by right-clicking the Event structure and removing the checkmark from the **Lock front panel until the event case for this event completes** checkbox in the Edit Events dialog box. You cannot turn off front panel locking for filter events.

Use a While Loop so the Event Structure Handles Every Event

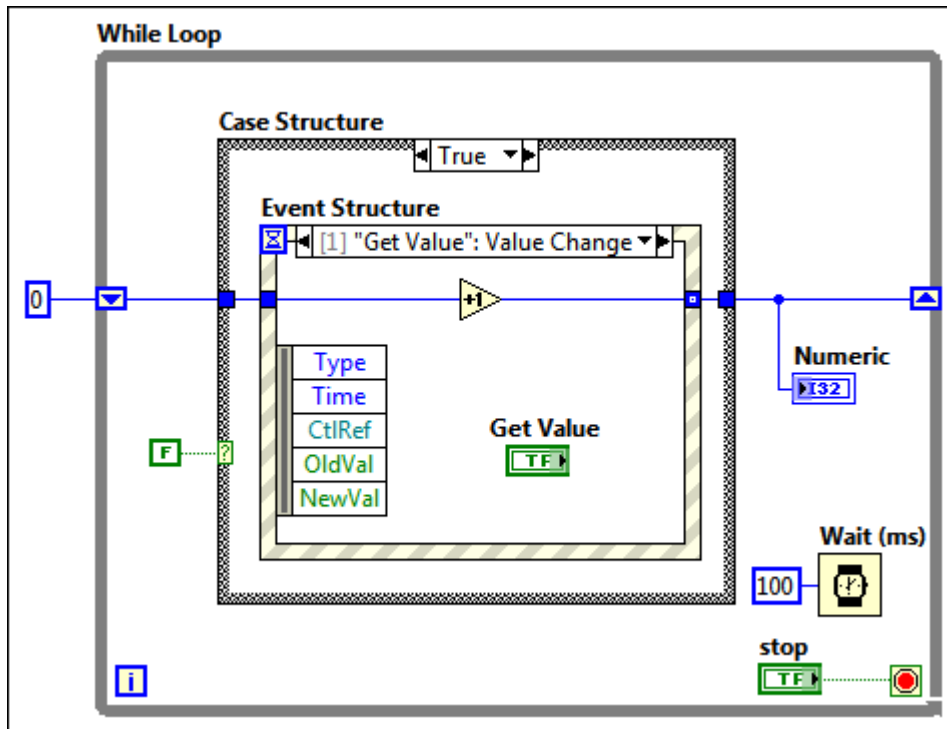
Place the Event structure in a While Loop that terminates when events can no longer occur. For example, in the following block diagram, the Event structure is outside the While Loop and front panel locking is enabled for the numeric Value Change case.



If you change the value of the numeric control, an event occurs. The Event structure executes once and handles the Value Change event. If you change the value of the numeric control again, another event occurs, and the *user interface locks* because the front panel locking is enabled. The Event structure already executed once, and because it is not inside a While Loop, it is unable to execute again to handle the second event. If you click the **stop** Boolean control to stop the While Loop and the VI, the VI cannot stop because the block diagram still has not handled the second event, and LabVIEW does not process the event when you click the **stop** Boolean control until the block diagram handles that event. You can avoid this behavior by placing the Event structure inside the While Loop.

Avoid Using a Case Structure to Determine when an Event Structure Handles Events

Avoid placing an Event structure within a Case structure. For example, in the following block diagram, the Event structure is inside a Case structure and front panel locking is enabled for the Value Change case.

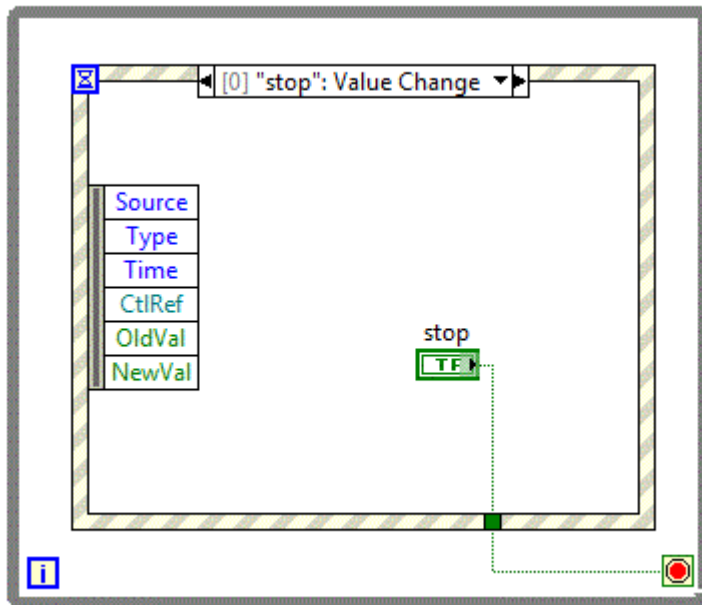


If you click the **Get Value** control, the front panel locks indefinitely. The False Boolean constant wired to the Case structure prevents the Event structure from handling the event inside the True case. Because panel locking is enabled for the event, the front panel waits for the Event structure, but dataflow prevents the Event structure from executing to handle the event. If you disable front panel locking for this application and click the **Get Value** control, the front panel becomes responsive and completes when you press the **stop** button. However, the numeric value does not increment because the Event structure still cannot execute to handle the event.

Handling a Latched Stop Boolean Control in an Event Structure

You must configure an Event structure case to handle a latched stop Boolean control. When using events with latched Boolean controls, you must place the control inside the event case for the mechanical action of a latched Boolean control to work correctly.

The following example shows the recommended way to handle a latched stop Boolean control in an Event structure. In this example, if a user clicks the **stop** button, the application stops running, and the latched stop Boolean control resets to FALSE.



The Event structure is inside a While Loop and a Value Change event case is configured for the latched stop Boolean control. When you run this application, the executions completes in the following order:

1. When the user clicks the **stop** button, the value of the latched stop Boolean control changes to TRUE, and the Event structure executes to handle the "stop": Value Change event case.
2. The "stop": Value Change event case reads the front panel control, passes a TRUE value to the control terminal, and resets the **stop** button to FALSE.
3. The While Loop conditional terminal reads the TRUE value and stops the loop.

Related Information

[Using Events with Latched Boolean Controls](#)

[Ensure That Event Structures Handle Events whenever Events Occur](#)

[Configuring Events Handled by the Event Structure](#)

[Changing the Mechanical Action of a Boolean Object](#)

One Case Handling Multiple Filter Events

A single case in an Event structure cannot handle multiple filter events unless the events have identical event data, such as Key Down? and Key Repeat?. With filter events, you can validate or change the event data before the front panel processes it, or you can discard the event data entirely, preventing the change from affecting the VI. Because you can modify the data, the Event structure cannot merge data like it can for notify events.

One Case Handling Multiple Notify Events

A single case in an Event structure can handle multiple notify events. However, if the events are not the same type, only the event data fields common to all events that the case handles appear on the left side of the case. If you handle multiple Value Change events for controls with different data types in the same case, the **OldVal** and **NewVal** event data fields return variant data.

If a single case in an Event structure handles multiple notify events for objects of different VI Server classes, the reference type is the common parent class of all objects. For example, if you configure a single case in the Event structure to handle events for a numeric control and a color ramp control, the type of the control reference of the event source is Numeric because the numeric and color ramp controls are in the Numeric class. You also cannot configure a single event case to handle an event on a dynamically registered DAQmx task and an event from some other event source, such as a LabVIEW control.

Prompting Users with Dialog Boxes in a Mouse Down? Event Case

Event cases that prompt users with a dialog box can cause unexpected behavior in the VI. For example, if you use a Mouse Down? event with a numeric control and configure the event case to prompt you with a dialog box, the Event structure responds when you click the numeric control, including the increment or decrement arrows, and displays a dialog box. Because the dialog box appears while you still have the mouse button depressed, the dialog box retains focus when you release the mouse button, and the numeric control never receives notification of the change in the state of the mouse button because the dialog box receives the notification instead. After you close the dialog box, the numeric control resumes responding to the previous Mouse Down? event and can continue to increment or decrement the control indefinitely, even though you are no longer pressing the mouse button.

To avoid this problem, National Instruments recommends that you do not use dialog boxes inside event cases that handle the Mouse Down? event. Instead, invoke the dialog box inside the Mouse Up event case or outside the Event structure.

Use Events for Direct User Interaction Only

User interface events occur only when a user directly interacts with the active front panel. In general, LabVIEW does not generate events if you use the VI Server, shared variables, global variables, local variables, DataSocket, and so on to change VI or front panel objects programmatically. The only exception is the Value (Signaling) property.

Use Only One Event Structure whenever Possible

You can use multiple Event structures in the same VI, especially if you [register events dynamically](#). However, events that you [register statically](#) rarely require multiple Event structures, so use one Event structure to handle events whenever possible.

When you use multiple Event structures in the same VI, observe the following guidelines:

- Place each Event structure in its own While Loop.
- Ensure that the Event structures execute simultaneously.
- Ensure that the Event structures stop handling events simultaneously.

Using Events versus Using the Wait for Front Panel Activity Function

Use the Wait For Front Panel Activity function when you want a VI to wait until a user interacts with the front panel. This function is designed to save system resources but does not give you any feedback about which event occurred.

Use the Event structure when you need to execute different code in response to particular user actions on the front panel, such as a mouse click, keystroke, and so on.

Using Events with Latched Boolean Controls

When you trigger an event on a Boolean control configured with a [latching mechanical action](#), the Boolean control does not reset to its default value until the block diagram reads the terminal on the Boolean control. You must read the terminal inside the event case for the mechanical action to work correctly. As a reminder, a note appears in the Edit Events dialog box when you configure a Value Change event on a latched Boolean control.

When you perform an undo operation on a Boolean control configured with a latching mechanical action, the Event structure handles the undo operation as a value change event, which might return unexpected results. You must use a Case structure within the event case for the undo operation to work correctly. Use the TRUE case of the Case structure to specify how you want to handle the Boolean control and the FALSE case to specify how you want to handle the undo operation.

Refer to the [Handling a Latched Stop Boolean Control in an Event Structure](#) caveat for information about how to handle a latched stop Boolean control.

Using Events with Subpanel Controls

If you configure a VI to handle an event and then configure the VI you [load into a subpanel control](#) to handle the same event, the top-level VI containing the subpanel control handles the event.

Strings

Strings

Strings group sequences of ASCII characters. Strings provide a platform-independent format for information and data. LabVIEW stores a string in a special format in which the first four bytes of the array of characters form a 32-bit signed integer that stores how many characters appear in the string. Thus, a string with n characters requires $n + 4$ bytes to store in memory.



RELATED INFORMATION

[Converting Numbers into Strings](#) on page 1329

[Formatting Strings](#) on page 1330

[Inserting Non-Displayable Characters in Strings](#) on page 1330

[Searching for and Replacing Text in a String](#) on page 1331

[Special Characters for Match Pattern](#) on page 1332

Converting Numbers into Strings

Complete the following steps to convert a number into a string using the Format Into String function.

1. Add a numeric control to the front panel window.
2. Add a string indicator to the front panel window.
3. Add the Format Into String function to the block diagram.
4. Right-click the Format Into String function and select **Edit Format String** from the shortcut menu to display the format string in the Edit Format String dialog box. Use this dialog box to create the **format string**, which specifies the format, precision, data type, and width of the converted value.
 1. From the **Selected operation** pull-down menu, select **Format fractional number**.
 2. Select **Right justify** from the **Justification** pull-down menu and **Pad using zeros** from the **Padding** pull-down menu.
 3. Place a checkmark in the **Use minimum field width** checkbox in the **Options** section and type 8. Press the <Enter> key. The **Corresponding format string** field should now contain %08f. If you enter a number in the numeric control with less than 8 digits, this entry pads the remaining spaces with zeros before the first digit.
 4. Place a checkmark in the **Use specified precision** checkbox in the **Options** section and type 4. Press the <Enter> key. The **Corresponding format string** field should now contain %08.4f. This entry converts the number in the numeric control into a string with four digits after the decimal point.

5. Click the **OK** button. The function creates a string constant and wires it to the **format string** input.
5. Wire the numeric control you created in step 1 to the **input 1** input of the Format Into String function.
6. Wire the string indicator you created in step 2 to the **resulting string** output of the Format Into String function.
7. Return to the front panel and run the VI. Notice that the string indicator displays a number with four decimal places.

You also can use the String/Number Conversion functions to convert between numeric and string representations.

Formatting Strings

To use data in another VI, function, or application, you often must convert the data to a string and then format the string in a way that the VI, function, or application can read. For example, Microsoft Excel expects strings that include delimiters, such as tab, commas, or blank spaces. Excel uses the delimiter to segregate numbers or words into cells.

In many cases, you must enter one or more format specifiers in the **format string** parameter of a String function to format a string. The format specifier is the code that indicates how to format a string.

The following functions format strings:

- Array to Spreadsheet String
- Spreadsheet String to Array
- Scan From String
- Format Into String
- Format Value
- Scan Value
- Scan From File
- Format Into File



RELATED INFORMATION

[Format Specifier Syntax](#) on page 469

Inserting Non-Displayable Characters in Strings

Use special escape codes to insert non-displayable characters, the backslash, and percent characters in the **format string** parameter. A code does not exist for the platform-dependent end-of-line (EOL) character. Use the end of line constant.

In the Scan From String function, a space in the **format string** parameter matches any amount of white space (spaces, tabs, and form feeds) in the input string.



RELATED INFORMATION

[Backslash \('\'\) Codes Display](#) on page 1359

Replacing Text in a String at a Specified Point

Complete the following steps to replace text in a string.

1. Add the Replace Substring function to the block diagram.
2. Wire the string in which you want to replace text to the **string** input of the Replace Substring function.
3. Wire the string containing the replacement text to the **substring** input of the Replace Substring function.
4. In the **offset** input of the Replace Substring function, enter the number of characters into **string** where you want to start replacing text. For example, to start replacing text after the fifth character in **string**, enter 5 in **offset**.
5. Wire the **result string** output of the Replace Substring function to a string indicator or VI or function that accepts a string as an input.
6. Run the VI. The **replaced substring** output of the Replace Substring function returns the text you replaced.

You also can [search for and replace text in a string](#).

Searching for and Replacing Text in a String

Complete the following steps to search for and replace text in a string.

1. Add the Search and Replace String function to the block diagram.
2. Wire the string you want to search to the **input string** input of the Search and Replace String function.
3. Wire the string you want to search for to the **search string** input.
4. Wire the replacement string to the **replace string** input.
5. To replace all occurrences of the text, set the **replace all?** input of the Search and Replace String function to TRUE. To replace the first occurrence, set **replace all?** to FALSE.
6. Wire the **result string** output of the Search and Replace String function to a string indicator or VI or function that accepts a string as an input.
7. [Wire all error in and error out terminals](#) on the block diagram.
8. Run the VI. The **offset past replacement** output of the Search and Replace String function indicates the point in the string where LabVIEW found the first match of the text in **replace string**.

You also can *replace text at a specified point in the string* and *find and replace text in VIs*.

Special Characters for Match Pattern

Use the following special characters in the **regular expression** input of the Match Pattern function.



NOTE
The Match Pattern function is compatible with a limited set of regular expressions and does not support character grouping, alternate pattern matching, backreferences, or non-greedy quantification. Instead of the Match Pattern function, use the Match Regular Expression function for more complex pattern matching.

Special Character	Description	Example(s)
. (period)	Matches any single character except a newline character. Within square brackets, . is literal.	string: Welcome to LabVIEW. regular expression: t... match substring: to La If you input [z .] as the regular expression, the period is literal and matches either . or z. In this example, [z .] returns . as the match.
*	Marks the single preceding character or character class as one that can appear zero or more times in the input. Because an asterisk can match a pattern that appears zero times, regular expressions that include an asterisk can return an empty string if the whole pattern is marked with an asterisk. This quantifier matches as many characters as possible.	string: Hello LabVIEW! regular expression: e1* match substring: e11 Expressions such as w* or (welcome)* match an empty string if the function finds no other matches.

Special Character	Description	Example(s)
+	Marks the single preceding character or character class as one that can appear one or more times in the input. This quantifier matches as many characters as possible.	string: Hello LabVIEW! regular expression: e1+ match substring: e11
?	Marks the single preceding character or character class as one that can appear zero or one time in the input. This quantifier matches as many characters as possible by default.	string: Hello LabVIEW! regular expression: e1? match substring: e1
[]	Creates a character class, which allows you to match any one of a set of characters that you specify. For example, [abc] matches a, b, or c. The Match Pattern function interprets special characters inside square brackets literally, with the following exceptions:	string: version=14.0.1 regular expression: [0-9]+\.[0-9]+\.[0-9]+ match substring: 14.0.1 The expression [0-9] matches any digit. The plus sign matches the previous character class, [0-9], one or more times but as many times as possible. The expression \. matches a literal . character. The plus sign matches the previous character class, [0-9], one or more times but matches as many times as possible. You can use this regular expression to match a three-part version number.
- (dash)	Indicates a range when used between digits, or lowercase or uppercase letters; for example, [0-5], [a-g], or [L-Q].	

Special Character	Description	Example(s)
~	Matches any character, including non-displayable characters, except for the characters or range of characters in brackets. For example, [~0-9] matches any character other than 0 through 9.	
^	Matches any displayable character, including the space character, except the characters or range of characters enclosed in the brackets. For example, [^0-9] matches all displayable characters, including the space character, except 0 through 9.	
^	If ^ is the first character of regular expression, it anchors the match to the offset in string. The match fails unless regular expression matches that portion of string that begins with the character at offset. If ^ is not the first character, it is treated as a regular character.	string: Hello LabVIEW! regular expression: ^[^]+match substring: Hello From the beginning of the input string, this regular expression matches as many characters—other than a space character—as possible. You can use this regular expression to isolate the first word, numeral, or other character combination of a string.

Special Character	Description	Example(s)
\$	If \$ is the last character of regular expression , it anchors the match to the last element of string . The match fails unless regular expression matches up to and including the last character in the string. If \$ is not last, it is treated as a regular character.	string :Hello LabVIEW! regular expression : [^] + \$ match substring : LabVIEW! From the end of the input string, this regular expression matches as many characters—other than a space character—as possible. You can use this regular expression to isolate the last word, numeral, or other character combination of a string.
\	Cancels the interpretation of any special character in this list. For example, \? matches a question mark, \. matches a period, and \\ matches a backslash. You also can use the following constructions for the space and non-displayable characters:	string :Welcome to the LabVIEW Help! regular expression : come\nto\tthe\sLabVIEW\sHelp\21 match substring :come to the LabVIEW Help! The expression come\n matches the literal letters followed by a newline character. The expression to\t matches the literal characters to followed by a tab. The two \s expressions match the spaces between the and LabVIEW and LabVIEW and Help!. The expression \21 matches the exclamation point because 21 is the hexadecimal code for an exclamation point.
\b	backspace	
\f	form feed	
\n	newline	
\s	space	
\r	carriage return	
\t	tab	
\xx	any character, where xx is the hex code using 0 through 9 and upper case A through F	

Special Characters for Match Regular Expression and Search and Replace String

Regular expressions require specific combinations of characters for pattern matching. Use regular expressions to match or locate a portion or subset of a string. For example, you can use regular expressions to match a variable in a string of code or a specific value in a report.



NOTE
The Match Regular Expression and Search and Replace String functions do not support null characters in strings. Also, a regular expression can return a successful match of an empty string. If a regular expression does not return a successful match, the **offset past match** will return -1.

Regular expression support is provided by the PCRE library package. Refer to the <National Instruments>_Legal Information directory for more information about the license under which the PCRE library package is redistributed.

Refer to the Perl 5 Regular Expression documentation at www.pcre.org for more information about Perl Compatible Regular Expressions.

Special Characters for Regular Expression and Search String Inputs

Use the special characters in the following table in the **regular expression** input of the Match Regular Expression function and the **search string** input of the Search and Replace String function. To use special characters with the Search and Replace String function, right-click the function and select **Regular Expression** from the shortcut menu.

	Note The following special characters are compatible only with the Match Regular Expression and Search and Replace String functions. The Match Pattern function uses a more limited set of regular expressions but performs more quickly than the Match Regular Expression function. Refer to the Special Characters for Match Pattern topic for more information about regular expressions for the Match Pattern function.

Special Character	Description	Example(s)
. (period)	Matches any single character except a newline character. Within square brackets, . is literal.	Input String: Welcome to LabVIEW. Regular Expression: t to L. If you input [z .] as the regular expression, the period is literal. The expression matches either . or z. In this example, [z .] returns . match.

Special Character	Description	Example(s)
*	Marks the single preceding character, character group, or character class as one that can appear zero or more times in the input. Because an asterisk can match a pattern that appears zero times, regular expressions that include an asterisk can return an empty string if the whole pattern is marked with an asterisk. This quantifier matches as many characters as possible by default.	Input String: Hello LabVIEW! Regular Expression: e1* Match: e1 Expressions such as w* or (welcome)* match an empty string if the function finds no other matches.
+	Marks the single preceding character, character group, or character class as one that can appear one or more times in the input. This quantifier matches as many characters as possible by default.	Input String: Hello LabVIEW! Regular Expression: e1+ Match: e1
?	Marks the single preceding character, character group, or character class as one that can appear zero or one time in the input. This quantifier matches as many characters as possible by default. When used immediately after a quantifier, ? modifies the quantifier to match as few times as possible. Modifiable quantifiers include *, +, and { }.	Input String: Hello LabVIEW! Regular Expression: e1? Match: e1 String: HelloLabVIEW Regular Expression: . +? Match: Hello In the second example, if you remove ? from the regular expression, the new match becomes HelloLabVIEW because + matches as many characters as possible unless you include ? immediately after +. You can use this regular expression to match any string within to

Special Character	Description	Example(s)
{n,N}	Marks the single preceding character, character group, or character class as one that can appear the number of times you specify, where n is the minimum and N is the maximum. You also can specify a single number. If you specify a range, this quantifier matches as many times as possible.	Input String: <u1>HelloLabVIEW! Regular Expression: (.+?){2} Match: HelloLab Input String: <u1>HelloLabVIEW! Regular Expression: (.+?){1,3} Match: HelloLabVIEW In the second ex the minimum match limit is one and the maximum is three. Because regular expression matches as many times as possible within the lim specify, the regular expression returns three matches.
[]	Creates a character class, which allows you to match any one of a set of characters that you specify. For example, [abc] matches a, b, or c. You can use - to specify a range of characters. For example, [a-z] matches any single lowercase letter. The Match Regular Expression and Search and Replace String functions interpret special characters inside square brackets literally, with the exception of ^, -, and \.	Input String: version=14.0.1 Regular Expression: [0-9]+(\.[0-9]+)* Match: 14.0.1 The expression [0-9] ma any digit. The plus sign matches the previous character class, [0-9], more times but as many times as possible. The parentheses create a ch group, which creates a submatch of the . character and all following The expression \. matches a literal . character. The plus sign match previous character class, [0-9], one or more times but matches as times as possible. The asterisk matches the previous character gro (\.[0-9]+), zero or more times, so the regular expression still ma integers if there is no . character. You can use this regular expression to any integer, decimal number, version number, IPv4 address, or other n sequence separated by . characters.

Special Character	Description	Example(s)
()	Creates a character group, which allows you to match an entire set of characters that you specify. A quantifier that immediately follows a character group quantifies the entire group. Parentheses also create submatches where each individual character group returns a submatch. If you nest a character group within another character group, the regular expression creates a submatch for the outer group before the inner group. Expand the Match Regular Expression function node to access submatch outputs. You also can refer back to submatches later in an expression using backreferences. Refer to the Backreferences section below for more information about using backreferences in regular expressions.	Input String: Hello LabVIEW! Regular Expression: (e1.) . . (L Match: e11o Lab Submatch 1: e11 Submatch 2: Lab Input String: H LabVIEW! Regular Expression: (. (e1.) . .) (L . .) Match: Hello Submatch 1: Hello Submatch 2: e11 Submatch 3: Lab

Special Character	Description	Example(s)
	Separates alternate possible matches. This character is useful when you want to match any of a number of character groups. A regular expression that contains () returns the first match that the function finds in the input string regardless of the order of your possible matches. For example, both regular expressions dog cat and cat dog match dog in the dog chased the cat.	Input String: value=FALSE total=12.34 token=TRUE Reg Expression: (value token)=(TRUE FALSE) Match: value=F Submatch 1: value Submatch 2: FALSE The regular expression retur first possible match in the input string. If token=TRUE appeared b value=FALSE in the input string, the regular expression would m token=TRUE instead.

Special Character	Description	Example(s)
^	Anchors a match to the beginning of a string when used as the first character of a pattern. If you set the multiline? input to TRUE on the Match Regular Expression or Search and Replace String functions, ^ matches the beginning of any line within the string using the line endings of the current platform. You also can match any character not in a given character class by adding ^ to the beginning of a character class. For example, [^0-9] matches any character that is not a digit. [^a-zA-Z0-9] matches any character that is not a lowercase or uppercase letter and also not a digit.	Input String: Hello LabVIEW! Regular Expression: ^ [^] + Match: Hello LabVIEW! The regular expression matches as many characters as possible—other than a space character—from the beginning of the input string. You can use this regular expression to isolate the first word, numeral, or other character combination of a string. Input String: Hello LabVIEW Regular Expression: ^LabVIEW Match: LabVIEW The regular expression matches LabVIEW only if you set multiline? to TRUE.
\$	Anchors a match at the end of a string when used as the last character of a pattern. If you set the multiline? input to TRUE on the Match Regular Expression or Search and Replace String functions, \$ matches the end of any line within the string using the line endings of the current platform.	Input String: Hello LabVIEW! Regular Expression: [^] +\$ Match: Hello LabVIEW! The regular expression matches as many characters as possible—other than a space character—from the end of the input string. You can use this regular expression to isolate the last word, numeral, or other character combination of a string. Input String: Hello LabVIEW Regular Expression: Hello\$ Match: Hello The regular expression matches Hello only if you set multiline? to TRUE.

Special Character	Description	Example(s)
	Cancels the special meaning of any special character in this list that immediately follows the backslash and instead matches the literal character. The following escaped expressions have special meanings: <code>\b</code> - Represents a word boundary. A word boundary is a character that is not a word character adjacent to a character that is a word character and vice versa. A word character is an alphanumeric character or an underscore (<code>_</code>). For example, <code>\bhat</code> matches <code>hat</code> in <code>hatchet</code> but not in <code>that</code>. <code>hat\b</code> matches <code>hat</code> in <code>that</code> but not in <code>hatchet</code>. <code>\bhat\b</code> matches <code>hat</code> in <code>hat</code> but not in <code>that</code> or <code>hatchet</code>. <code>\c</code> - Matches any control or non-printing character; includes any code point in	



Tip To anchor a match at the beginning and end of a string, use ^ as the first character in a pattern and \$ as the last character of a pattern. For example, ^LabVIEW\$ matches LabVIEW in LabVIEW but not in LabVIEW! or Hello LabVIEW. Anchoring the match at the beginning and end of the string requires the whole string to match.

Special Characters for Replace String Input

Use the following special characters in the **replace string** input of the Search and Replace String function.

Special Character	Description	Example(s)
\$n or \${n}	Inserts a string you specify before or after the submatch you specify. Refer to the Backreferences section below for more information about using backreferences in the replace string input of the Search and Replace String function. Use \${n} if you have more than nine submatches in a regular expression and want to refer to a submatch after the ninth. \$12 inserts only the first submatch because the function reads only the first digit that immediately follows \$. However, \${12} inserts the twelfth submatch.	input string: Welcome LabVIEW search string: (LabVIEW) replace string: to \$1! result string: Welcome to LabVIEW! input string: Welcome to the LabVIEW Help! search string: We(1)(co(m)e)((to)(t(he)(Lab)(VIE(W))) He(1p)(!) replace string: \$7\${11}ful\${12} result string: helpful!
\\$n	Cancels the interpretation of any special character you use in the replace string.	input string: total=123 per day search string: (\d+)(per \w*)? replace string: \\$\$1\$2 result string: total=\$123 per day As in this example, use \\$ to insert the literal character \$. Use \\ to indicate a literal backslash.

Backreferences

Use backreferences to refer to previous submatches within the same regular expression. You can use backreferences to create a submatch using a character group in one part of an expression and then match that exact submatch in a later part of the expression.

To specify a back reference in the **regular expression** input of the Match Regular Expression function or the **search string** input of the Search and Replace String function, use `\1` to refer to the first submatch, `\2` to refer to the second, and so on. For example, consider the following regular expression:

```
(["*$])(\w+)\1\2\1
```

The first character group contains a character class that matches `"`, `*`, or `$`. The second character group matches one or more word characters. The first backreference, `\1`, matches the same submatch as the first character group, `(["*$])`. The second backreference, `\2`, matches the same submatch as the second character group, `(\w+)`. The third backreference, `\1`, is identical to the first backreference and matches the same submatch as the first character group.

This example matches strings such as `"foo"foo"`, `*bar*bar*`, and `$baz$baz$`, but does not match strings such as `"foo$foo"` or `"foo*bar"`.

Backreferences in Replace String

When you use the Search and Replace String function in **Regular Expression** mode, you can specify a backreference in the **replace string** input that refers to submatches in the **search string** input. Use `$1` to refer to the first submatch, `$2` to refer to the second submatch, and so on. You can use these special characters only in the **replace string** input of the Search and Replace String function. Use these special characters to insert a string you specify before or after the submatch you specify. Consider the following example:

```
input string: $value "TRUE"TRUE" *NULL
search string: (["*$])(\w+)\1\2\1
replace string: $1$2value$1
result string: $value "TRUEvalue"
*NULL
```

In this example, the `$1` backreferences in **replace string** refer to the first submatch in **search string**. Likewise, the `$2` backreference refers to the second submatch in **search string**.

Splitting Strings

Splitting strings divides a single string into two substrings. You can divide the string at a certain character, such as a carriage return, or at a word or phrase.

1. Add the Search/Split String function to the block diagram.
2. Wire the string you want to split to the **string** input of the Search/Split String function.
3. Enter the text that indicates where **string** should be split in the **search string/char** input of the Search/Split String function.
4. In **offset**, enter the number of characters in **string** to skip before the Search/Split String function starts searching for the text in **search string/char**.

5. Wire the **substring before match** and the **match + rest of string** outputs of the Search/Split String function to string indicators or VIs or functions that accept a string as an input.
6. Run the VI.

Using Format Specifiers in Numeric Strings

Use format specifiers, also known as percent codes, to specify how you want LabVIEW to display numbers. You can use the **Numeric Properties** dialog box or the Format String property to specify how you want to format the numeric control. Format specifiers include the following items:

- **Format Specifier** —String that begins with a % and ends with a letter. Letters are case sensitive. Format specifiers can include the following elements between the % and the letter:
 - **Modifiers** —A symbol that modifies the format code.
 - **Minimum Width** —Number greater than zero that specifies the field width.
 - **Precision** —String that begins with a . and is followed by a number greater than or equal to zero.
 - **Significant Digits** —String that begins with a _ and is followed by a number greater than or equal to zero.
 - **Embedded Format Information** —String that LabVIEW uses to further format a number. Enclose string within < and >.
- **Delimiter** —Text between, before, or after format codes. % and > are the two delimiter characters. To embed a % character in a string, use %%. To embed a > character in a string, use \>. Refer to the [Backslash \('\'\) Codes Display](#) topic for more information about how LabVIEW interprets backslash codes.
- **Literal strings** —Text you want to appear in the string.



NOTE

You cannot use precision and significant digits together in a single format specifier.

Refer to the [Format Specifiers Examples](#) table for more examples of format specifiers.

Entering a Carriage Return into a String

Complete the following steps to enter a carriage return into a string.

1. Add a string control to the front panel. The default mode is [Normal Display](#).
2. Right-click the string control and select ['\'\) Codes Display](#) from the shortcut menu.
3. Use the Operating tool or the Labeling tool to add the cursor to the string control where you want to enter a carriage return.

4. Type `\r`.

Entering a Linefeed into a String

Complete the following steps to enter a linefeed into a string.

1. Add a string control to the front panel. The default mode is *Normal Display*.
2. Right-click the string control and select `'\'` *Codes Display* from the shortcut menu.
3. Use the Operating tool or the Labeling tool to add the cursor to the string control where you want to enter a linefeed.
4. Type `\n`. You also can enter a linefeed into a string by pressing the <Enter> key.

Entering a Tab Character into a String

Complete the following steps to enter a tab character into a string.

1. Add a string control to the front panel. The default mode is *Normal Display*.
2. Right-click the string control and select `'\'` *Codes Display* from the shortcut menu.
3. Use the Operating tool or the Labeling tool to add the cursor to the string control where you want to enter a tab character.
4. Type `\t`.

Entering or Changing Text in a String Control

Complete the following steps to enter or change text in a string control.

1. Add a string control to the front panel. The default mode is *Normal Display*.
2. Use the Operating tool or the Labeling tool to add the cursor to the string control and enter or edit the text you want. When the text reaches the right border of the string display, the string wraps to the next line, breaking on a natural separator such as a space or tab character.



NOTE

You can double-click the text to highlight a word or triple-click the text to highlight the entire string.

You also can use several *keyboard shortcuts* in a string control.

Selecting and Entering Strings in Combo Box Controls at Run Time

Use the following ways to select and enter strings in a combo box control while a VI is in run mode:

- Click the arrow button to the right of the control and select a string from the shortcut menu that appears.
- Click the arrow button to the right of the control and use the up and down arrow keys to navigate to the string you want to select. After you highlight the string in the shortcut menu, press the <Enter> key or click the arrow button to select the string.
- Place the cursor in the text box, press the down arrow key to display the shortcut menu, and use the up and down arrow keys to navigate to the string you want to select. After you highlight the string in the shortcut menu, press the <Enter> key or click the arrow button to select the string.
- Click the arrow button to the right of the control and type the beginning letters of the string you want to select. LabVIEW highlights the first, shortest string in the control that begins with the letters you type. Press the <Enter> key or click the arrow button to select the string. If no strings match the letters you type and the control does not allow *undefined string values*, LabVIEW does not accept or display the letters you type in the control. If no strings match the letters you type and the control allows undefined string values, press the <Enter> key to enter the new string value in the control.
- Place the cursor in the text box and type the beginning letters of the string you want to select. LabVIEW displays the first, shortest string in the control that begins with the letters you type. Press the <Enter> key to select the string. If no strings match the letters you type and the control does not allow undefined string values, LabVIEW does not accept or display the letters you type in the control. If no strings match the letters you type and the control allows undefined string values, press the <Enter> key to enter the new string value in the control.
- If the control allows undefined string values, place the cursor in the text box and type the string you want to enter. Press the <Enter> key to enter the new string value in the control.

Specifying Custom Values for Combo Box Strings

When you *configure the list of strings for a combo box control*, you can specify a custom value for each string, which is useful if you want the string that appears in the front panel combo box control to be different than the string value the combo box terminal on the block diagram returns.

Complete the following steps to specify a custom value for a string in a combo box control.

1. Right-click the combo box control and select **Edit Items** from the shortcut menu to display the Edit Items page of the **Combo Box Properties** dialog box.
2. Remove the checkmark from the **Values match Items** checkbox.

3. Double-click in the cell in the **Values** column that contains the value you want to change.
4. Enter the string value you want to use on the block diagram when the user selects the corresponding label in the control.
5. Repeat steps 3–4 for each value you want to change.
6. Click the **OK** button.

You also can use the Strings And Values [] property to specify custom values programmatically.

Adding Color Ramps to Dials, Knobs, and Gauges

A color ramp uses color to display its numeric value. You configure a color scale that consists of at least two arbitrary markers, each with a numeric value and a corresponding display color. As the input value changes, the color displayed changes to the color that corresponds to that value. Color ramps are useful for visually indicating data ranges, such as a warning range for when a gauge reaches dangerous values. For example, you can use a color ramp to set the color scale for intensity [graphs](#) and [charts](#).

Complete the following steps to add and configure a color ramp:

1. Right-click a dial, knob, or gauge and select **Visible Items»Ramp** from the shortcut menu to add a color ramp.
2. Right-click the color ramp or a marker and select **Marker Spacing»Arbitrary** from the shortcut menu to switch the color ramp to arbitrary marker spacing.
3. To change the value of an arbitrary marker on a color ramp, use the Operating tool to drag the marker to the value you want or use the Labeling tool to highlight the text of the marker and enter a new value. When you drag a marker, the cursor changes to  to indicate the tool is over a marker.
4. To add a new marker to the color ramp, right-click the position on the color ramp where you want to add a new marker and select **Add Marker** from the shortcut menu. LabVIEW adds a new marker to that location. You also can press the <Ctrl> key while you drag an existing marker to the location where you want to add a marker. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.
5. Use the Operating or Positioning tools to right-click the marker next to the color ramp and select **Marker Color** from the shortcut menu to change the color associated with a marker.
6. Select the color you want from the [color picker](#) that appears.
7. Resize the color ramp by moving the Positioning tool over one of the handles that appear on the midpoints of the color ramp. Drag the handle to the width you want.



NOTE

Meters have a visible color ramp by default.

Adding Sliders or Needles

Complete the following steps to add a slider or needle to a numeric object with a scale.

1. Right-click the slide or rotary object and select **Add Slider** or **Add Needle** from the shortcut menu.
2. Use the Coloring tool to change the color of the slider or needle if you want to distinguish between the sliders or needles.
3. (Optional) For slide controls or indicators, *change the fill options* to **Fill To Value Below** or **Fill To Value Above**.

You also can *customize* the slider or needle.

Adding Text Labels to Numeric Objects with Scale

You can add text labels to slide or rotary objects for uses such as associating strings with integer values and selecting specific integer values.

Complete the following steps to create text labels on numeric objects with scales.

1. Right-click the slide or rotary object and select **Text Labels** from the shortcut menu. The default text labels, **min** and **max**, appear on the scale. A text display also appears and displays the text labels.
2. Use the Labeling tool to edit the labels.
3. Right-click the text display and select **Visible Items»Digital Display** from the shortcut menu to display the numeric values associated with the text labels you create. These values always start at zero for the minimum value of the scale and increase by one for each text label.
4. Complete the following steps to add more text labels and corresponding markers to the scale.
 1. Right-click the text display and select **Edit Items** from the shortcut menu to display the Edit Items page of the **Slide Properties** dialog box or the **Knob Properties** dialog box.
 2. Double-click in an empty cell in the **Items** column and enter the string you want to display next to the scale marker. You also can place the cursor in a cell that already contains a string, click the **Insert** button to create an empty row above the cell, and enter a string in the new cell.
 3. (Optional) If the *marker spacing is arbitrary*, you can assign a specific numeric value to each item by removing the checkmark from the **Sequential values** checkbox and entering the numeric value you want to use. By default, after you enter a string in the **Items** column, LabVIEW adds a numeric value for the marker to the corresponding cell in the **Values** column. LabVIEW assigns sequential values that correspond to the order of the items in the list, starting with a value of 0 for the first item.
 4. Click the **OK** button.

You can edit the text label for a marker by placing the cursor in the text display, editing the label, and pressing the <Ctrl-Enter> keys. (macOS) Press the <Option-Enter> keys. (Linux) Press the <Alt-Enter> keys.

You also can use the Text Labels[] property to edit text labels programmatically.

You can right-click the text display and select **Disable Item** from the shortcut menu to disable a text label and marker so you cannot select the item at run time.

Changing Scale Limits

The outer two markers on slide and rotary controls or indicators are the scale limits. Scale limits do not have to match range limits. Scale limits can be a subset of the range of the control or indicator. For example, the default range of a 16-bit signed integer control or indicator is -32,768 to 32,767. However, you can set the data range to -1,000 to 1,000 and then set the scale limits to 0 and 500.

Complete the following steps to change scale limits and increments.

1. Use the Operating tool or the Labeling tool to click the upper or lower limit on the scale of the slide or rotary object.
2. Highlight the digits you want to change and type the new number.
3. Press the <Enter> key, click the **Enter** button on the toolbar, or click outside the scale to enter the new value. The other limit stays the same and LabVIEW recalculates the increment automatically.
4. Repeat steps 1 through 3 to change the value of an intermediate marker on the scale. LabVIEW adjusts the increments of the scale automatically. If the increments of the scale do not adjust automatically, the intermediate marker is too close to another marker, so you need to *increase the size of the slide or rotary object* before you change the value of the intermediate marker.

You also can use the Range properties to change scale limits and increments programmatically.

Changing Scrolling Ranges of Scroll Bar Controls

You can change the scrolling range of scroll bar controls by using the Data Entry page of the **Scrollbar Properties** dialog box. For example, if **Minimum** is 0, **Maximum** is 10, and **Page size** is 2, the scrolling range is 0 to 8. The top or left position of the scroll box determines the value of the scroll bar.

Complete the following steps to set the scrolling ranges for a scroll bar control.

1. Right-click a scroll bar control and select **Data Entry** from the shortcut menu to display the **Data Entry** page of the **Scrollbar Properties** dialog box.
2. Remove the checkmark from the **Use Default Limits** checkbox.

3. In the **Minimum** field, set the low value limit and decide whether to select the **Coerce** option from the pull-down menu. If you do not coerce **Minimum**, you can decrement the scroll bar value out of the scrolling range by clicking the decrement arrow.
4. In the **Maximum** field, set the high value limit and decide whether to select the **Coerce** option from the pull-down menu. If you do not coerce **Maximum**, you can increment the scroll bar value out of the scrolling range by clicking the increment arrow.
5. In the **Increment** field, set the incremental value. When you click the increment and decrement arrows, the scroll bar value adds or subtracts the increment value to move the scroll box toward the arrow that you click.
6. In the **Page size** field, set the page size. When you click the spaces between the scroll box and the arrows, the scroll bar value changes by the page size.

You also can use the Scrollbar properties to set the scrolling range programmatically.

Characters Numeric Controls and Indicators Accept

Numeric controls and indicators accept only the following characters.

Character(s)	Description
Hexadecimal digits	0 through F
Octal digits	0 through 7
Binary digits	0 and 1
Decimal digits	1, 1.0, 2, 3.5, and so on
.	Decimal point
+	Positive symbol
–	Negative symbol
E or e	For scientific or engineering notation format
Inf	Infinity
NaN	Not a number
/	For use in absolute time format, which changes based on the local settings for the computer
:	For use in absolute time format, which changes based on the local settings for the computer
AM, am, PM, pm	For use in absolute time format, which changes based on the local settings for the computer
SI prefixes	

Character(s)	Description
y	yocto (10^{-24})
z	zepto (10^{-21})
a	atto (10^{-18})
f	femto (10^{-15})
p	pico (10^{-12})
n	nano (10^{-9})
u	micro (10^{-6})
m	milli (10^{-3})
c	centi (10^{-2})
d	deci (10^{-1})
da	deka (10^1)
h	hecto (10^2)
k	kilo (10^3)
M	mega (10^6)
G	giga (10^9)
T	tera (10^{12})
P	peta (10^{15})
E	exa (10^{18})
Z	zetta (10^{21})
Y	yotta (10^{24})

Configuring the Time and Date in a Time Stamp Control

Use the time stamp control to use, view, and store absolute time with high precision. This data type can accurately store 18 digits of precision each in whole second and 19 digits of precision in fractions of a second. The time stamp wire data type stores values in UTC, but the time stamp control displays values in local time.

You can use a numeric control to display time stamp values, but the numeric control holds a relative quantity. The time stamp control holds an absolute quantity.

Complete the following steps to configure the time and date value in a time stamp control.

1. Add a time stamp control to the front panel window.
2. Use the Operating tool to click the **Time/Date Browse** button  to display the Set Time and Date dialog box. You also can right-click the time stamp control and select **Data Operations»Set Time and Date** from the shortcut menu to display the **Set Time and Date** dialog box.
3. Configure the time and date you want as the time stamp for the data.



NOTE

To configure the time, click the portion of time, such as minutes or seconds, you want to configure. LabVIEW highlights the selected portion of time. Use the up arrow and down arrow keys to increment or decrement accordingly. You also can enter a new value in the selected portion of time.

You also can right-click a time stamp control and select **Data Operations»Set Time to Now** from the shortcut menu to set the time and date value to the current time and date.

You can change the time stamp control value programmatically by [creating a Property Node](#) from the time stamp control and wiring a time stamp constant to the Value property.



Note You cannot use units with the time stamp control. In addition, National Instruments recommends you do not use units with a numeric control formatted for absolute time.

Displaying Higher Digits of Precision in a Time Stamp Control

Complete the following steps to display higher digits of precision for the [time stamp control](#).

1. Right-click the time stamp control and select **Display Format** from the shortcut menu to display the Display Format page of the **Time Stamp Properties** dialog box.
2. Select **Custom time format** in the **Time Type** pull-down menu located at the top of the time format section.
3. Click the **HH:MM:SS** button.
4. Enter the digits of precision in the **Digits** field. Although you can enter any number in this field, LabVIEW can accurately store up to 19 digits of precision in the fractions of a second.
5. Click the **OK** button. The time stamp control updates to the number of digits of precision you specified. Resize the control if necessary to view all digits.

If you do not want to display the time or the date, select **Time unused** or **Date unused** from the **Time Type** or **Date Type** pull-down menus on the **Display Format** page of the **Time Stamp Properties** dialog box.

Placing Arbitrary Markers on Scales

By default, scale markers are spaced evenly. You can indicate exactly where to place inner markers for a scale. This is useful for marking a few specific points on a slide, such as a set point or threshold.

Complete the following steps to specify specific points on a scale.

1. Right-click the scale and select **Marker Spacing»Arbitrary** from the shortcut menu.
2. Right-click the position on the scale where you want to add a new marker and select **Add Marker** from the shortcut menu. LabVIEW adds a new marker to that location. You also can press the <Ctrl> key while you drag an existing marker to the location where you want to add a marker. On slide controls and indicators, you also can drag one of the end markers to the location where you want to add a marker. On rotary controls and indicators, press the <Ctrl> key and drag one of the end markers to the location where you want to add a marker. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.

To delete an arbitrary marker, right-click the marker and select **Delete Marker** from the shortcut menu.

To change the scale representation, right-click the scale and select **Display Format** from the shortcut menu.



NOTE

The arbitrary markers affect only the inner markers, not the two end markers. If no arbitrary markers are visible in the current range, the scale reverts temporarily to uniform markers.

In uniform mode, dragging an inner marker changes the marker distribution. In arbitrary mode, dragging an inner marker moves only the current marker, without changing the other markers.

Scaling Slide and Rotary Controls and Indicators

You can change the **scale limits**, use **text labels**, adjust scale markers, and select linear or logarithmic spacing for numeric objects with a scale.

Complete the following steps to customize the scale of a slide or rotary object.

1. Right-click the slide or rotary object.
2. Select **Scale** from the shortcut menu and select from the following items:

- **Marker Spacing** —Use this option to select uniform or arbitrary scale markers. By default, scale markers are uniform, or evenly spaced. By selecting arbitrary marker spacing, you can indicate specific inner markers for a scale. *Arbitrary marker spacing* is useful for marking a few specific points on a slide or rotary control, such as a set point or threshold.
- **Add Marker** — *Adds a marker* to the scale. This option is not available if **Marker Spacing** is **Uniform**.
- **Delete Marker** — *Deletes a marker* from the scale. This option is not available if **Marker Spacing** is **Uniform**, and if the scale contains only the two end markers.
- **Display Format** —Displays the Display Format page of the **Slide Properties** dialog box. You can change the *format and precision* of numeric objects with a scale just as you do for digital objects.
- **Style** —Use this palette to select the style of the scale display. You can display a scale with or without tick marks, with or without scale values, or you can hide the scale altogether.
- **Mapping** —Use this option to select linear or logarithmic scale spacing. If you select logarithmic spacing, and the low scale limit is less than or equal to 0, the limit automatically becomes a positive number, and LabVIEW recalculates the value of the other markers accordingly. Keep in mind that scale options, including the mapping functions, are independent of the slide data range values, which you change using the **Data Entry** shortcut menu item. If you want to limit the data to logarithmic values, change the *data range* to eliminate values less than or equal to 0.

Using Scroll Bar Controls

You can use *scroll bar controls* to add custom scroll bars to a control with scrollable data. Complete the following steps to use a scroll bar control with a listbox.

1. Add a listbox to the front panel and *enter items in the listbox*.
2. Add a vertical scroll bar to the front panel.
3. Right-click the listbox, select **Create»Property Node»Top Row** from the shortcut menu, and add the Top Row property to the block diagram.
4. Right-click the Top Row property and select **Change All To Write** from the shortcut menu.
5. Wire the scroll bar terminal to the Top Row property.
6. Customize the scroll bar by right-clicking the front panel scroll bar, selecting **Data Entry** from the shortcut menu, and configuring options on the **Data Entry** page.
7. Click the **Run Continuously** button and change the value of the scroll bar control to scroll the items in the listbox.



NOTE

Using a custom scroll bar to scroll a control disables the built-in scroll bars of that control.

You also can use the Scrollbar properties to configure the scroll bar programmatically.

Using the Increment and Decrement Buttons

The increment and decrement buttons usually change the ones digit in a numeric control. If you click an increment or decrement button and hold the mouse button down, the display increases or decreases until you release the mouse button. If you click the increment or decrement button, hold down the mouse button, and hold down the <Shift> key, the size of the increment increases by successively higher orders of magnitude. The value first increments by ones, then by tens, by hundreds, and so on. As the range limit approaches, the increment decreases by orders of magnitude, slowing down as the value reaches the limit.

To use the arrow keys to change a numeric display by a digit other than the default increment, use the Operating tool to add the insertion point to the right of the target digit.

Complete the following steps to practice using the increment and decrement buttons.

1. Add a numeric control to the front panel window.
2. Using the Operating tool, click the increment and decrement buttons. The value increases and decreases by one.
3. Click and hold down the increment button to increase the value repeatedly.
4. Using the Positioning tool, resize the numeric control. Notice that the control can be increased in length only horizontally.
5. Using the Operating tool again, hold down the increment button and press the <Shift> key. Notice that the value begins to increase by one, but then by tens, hundreds, and so on.
6. Change the value of the numeric control to 10 by double-clicking inside the display window, typing 10, and pressing the <Enter> key.
7. Click the increment button two times. The value increases by one with each click and the value is now 12.
8. To increment by tens, use the Operating tool to add an insertion point to the right of the tens digit. Press the up arrow key to increase the value to 22. By placing the insertion point to the right of the tens digit, the increment value is changed to tens.
9. Move the insertion point to the right of the ones digit, and click the increment button. The value increments to 23. Notice that the ones digit again becomes the increment digit.

Right-click a numeric control and select **Visible Items»Increment/Decrement** from the shortcut menu to display or hide the increment and decrement buttons. You also can use the Increment/Decrement Visible? property to display or hide the buttons programmatically.

Valid Format Specifiers for Numeric Data Types

The following table shows which format specifiers are valid for each data type. If a cell contains X, the format is invalid for that data type. If the cell contains ✓, the format is valid

for that data type. If the data type changes and the current format is not allowed with the new data type or if the cell is labeled ✓-+, LabVIEW changes the format to the default format for that data type. For example, setting the format of a double-precision, floating-point data type to absolute time is valid, but when you change the double-precision, floating-point data type to a long signed integer data type, the format changes to decimal format.

Format Specifier	T	t	x	o	b	d	u	f	e	F
Data Types	Absolute Time	Relative Time	Hexadecimal	Octal	Binary	Signed	Unsigned	Floating-Point	Scientific Notation	Floating-Point
Time Stamp	✓	X	X	X	X	X	X	X	X	X
Complex types (CXT, CDB, CSG)	X	X	X	X	X	✓-+	✓-+	✓	✓	
Real floating-point types (EXT, DBL, SGL)	✓-+	✓-+	X	X	X	✓	✓-+	✓-+	✓	
Integer types (U64, I64, U32, I32, U16, I16, U8, I8)	✓-+	✓-+	✓	✓	✓	✓	✓	✓-+	✓-+	
Fixed-point type (FXP)	X	X	✓	✓	✓	✓	✓	✓	✓	

Empty Paths

An empty path is either an empty absolute path or an empty relative path.

Windows

An absolute path contains the complete address of a file's location within the file hierarchy. An empty absolute path refers to the highest point in the file hierarchy. An empty absolute path in a path control appears as an empty string and when you wire it to a file I/O function, it refers to the list of drives associated with the computer.

A relative path contains the address of a file's location relative to the position of the user within the file hierarchy. An empty relative path refers to the current position of the user within the file hierarchy. This path is represented as a period ..

macOS and Linux

An empty absolute path in a path control appears as a slash / and when you wire it to a file I/O function, the empty path refers to the root directory.

An empty relative path appears as a period . and refers to the current position of the user within the file hierarchy.

Symbolic Paths

A symbolic path or pseudopath is a predefined short form of a path that you can use in LabVIEW. Use a symbolic path to point to a location that changes with the execution environment, such as where on disk LabVIEW is installed or which user is logged in.

For example, the <vilib> symbolic path represents the <vi.lib> directory in the labview directory. At run time, LabVIEW interprets the <vilib> symbolic path as an absolute path, such as C:\Program Files (x86)\National Instruments\LabVIEW 2009\vi.lib or D:\Program Files\National Instruments\LabVIEW 2009\vi.lib, depending on where LabVIEW is installed.

The following table includes information about the symbolic paths that you can use in LabVIEW.

Symbolic path	Description	Usage
<osdatadir>	Path to the default file directory on the operating system.	Use this symbolic path when setting the VI Search Path on the Paths page of the Options dialog box. To use this symbolic path, enter <osdatadir>: \ then the filename or path and filename to the VI.
<vilib>	The vi.lib directory in the labview directory.	Use this symbolic path when setting the VI Search Path on the Paths page of the Options dialog box. To use this symbolic path, enter <vilib>: \ then the filename or path and filename to the VI.
<userlib>	The user.lib directory in the labview directory.	Use this symbolic path when setting the VI Search Path on the Paths page of the Options dialog box or the Help path on the Documentation page of the VI Properties dialog box. To use this symbolic path, enter <userlib>: \ then the filename or path and filename to the VI or help file.

Symbolic path	Description	Usage
<instrlib>	The <code>instr.lib</code> directory in the <code>labview</code> directory.	Use this symbolic path when setting the VI Search Path on the Paths page of the Options dialog box or the Help path on the Documentation page of the VI Properties dialog box. To use this symbolic path, enter <instrlib>: \ then the filename or path and filename to the VI or help file.
<helpdir>	The help directory in the <code>labview</code> directory.	Use this symbolic path when setting the Help path on the Documentation page of the VI Properties dialog box. To use this symbolic path, enter <helpdir>: \ then the filename or path and filename to the help file.
<resource>	The <code>resource</code> directory in the <code>labview</code> directory, which is intended to store support VIs for a particular version of LabVIEW.	Use this symbolic path when creating VIs that are meant to be additions to the LabVIEW editor and are not meant to be used in user applications. The APIs of the <code>resource</code> directory are subject to change between successive versions of LabVIEW without maintaining backward compatibility. To use this symbolic path, enter <resource>: \ then the filename or path and filename to the VI.
<ExtraVILib>	The <code>ExtraVILib</code> directory in the LabVIEW Data directory.	Use this symbolic path to store VIs instantiated from templates for various LabVIEW features. For example, LabVIEW stores channel instances for <i>channel wires</i> in this symbolic path. The channel instances are generated from channel templates. When you load VIs using a different LabVIEW version and bitness other than the LabVIEW version or bitness that you used to create the VIs, LabVIEW recreates a separate copy for the instances that the loading VIs need so that you can use the VIs in the loading LabVIEW version and bitness. To use this symbolic path, enter <ExtraVILib>: \ then the filename or path and filename to the VI.

Related Information

File Paths

Display Options

Backslash ('\') Codes Display

Right-click a string constant, control, or indicator and select **'\ ' Codes Display** from the shortcut menu to instruct LabVIEW to interpret characters that immediately follow a

backslash (\) as a code for non-displayable characters. The backslash mode is useful for debugging VIs and for sending non-displayable characters to instruments, serial ports, and other devices. The following list shows how LabVIEW interprets these codes.

Code	LabVIEW Interpretation
\00 – \FF	Hex value of an 8-bit character; must be uppercase
\b	Backspace (ASCII BS, equivalent to \08
\f	Form feed (ASCII FF, equivalent to \0C
\n	Linefeed (ASCII LF, equivalent to \0A—The Format Into File function automatically converts this code into the platform-dependent end-of-line character.
\r	Carriage return (ASCII CR, equivalent to \0D
\t	Tab (ASCII HT, equivalent to \09
\s	Space (equivalent to \20
\\	Backslash (ASCII \, equivalent to \5C
\\	

Use uppercase letters for hexadecimal characters and lowercase letters for special characters, such as form feed and backspace.

LabVIEW interprets the sequence \BFare as hex BF followed by the word are. It interprets \bFare and \bfare as a backspace followed by the words Fare and fare. In the sequence \Bfare, \B is not the backspace code, and \Bf is not a valid hex code. In a case like this, when a backslash is followed by only part of a valid hex character, LabVIEW assumes a 0 follows the backslash, so LabVIEW interprets \B as hex 0B. Any time a backslash is not followed by a valid hex character, LabVIEW ignores the backslash character.

You can enter some non-displayable characters listed in the previous table from the keyboard, such as a carriage return, into a string control regardless of whether you select **'\ Codes Display**. However, if you enable the backslash mode when the display window contains text, LabVIEW redraws the display to show the backslash representation of any non-displayable characters and the \ character itself.

Complete the following steps to practice using the **'\ Codes Display** shortcut menu item.

1. Add a string control to the front panel window. The default mode is *Normal Display*.
2. *Resize* the string control to display multiple lines of text and enter the following string: LEFT\RIGHT\3F
3. Right-click the string control and select **'\ Codes Display** from the shortcut menu. The string appears as follows because the carriage return after LEFT and the

backslash characters around RIGHT are shown in backslash form. LEFT\n\\
\\RIGHT\\\\3F

4. Repeat step 1 to add a new string control to the front panel window.
5. Right-click the new string control and select **'\ ' Codes Display** from the shortcut menu.
6. **Resize** the string control to display multiple lines of text and enter the following string: LEFT\RIGHT\3F
7. Right-click the string control and select **Normal Display** from the shortcut menu. The string appears as follows because LabVIEW originally interpreted \r as a carriage return and now prints one. \3F is the ASCII representation of the question mark ?. LEFTRIGHT?
8. Right-click the string control again and select **'\ ' Codes Display** from the shortcut menu. The string appears as follows: LEFT\nRIGHT?

Indicators behave the same way. Notice that the data in the string does not change from one mode to the other. Only the displayed representation of certain characters changes.

Configuring String Controls and Indicators

To configure a string constant, control, or indicator, right-click the object and select from the following shortcut menu items:

Normal Display

Backslash ('\') Codes Display

Password Display

Hex Display

Limit to Single Line

Update Value while Typing

Enable Wrapping

You also can right-click a string control or indicator and select **Visible Items»Display Style** from the shortcut menu to display a glyph on the object that indicates the display type.



RELATED INFORMATION

[Normal Display](#) on page 1363

[Backslash \('\'\) Codes Display](#) on page 1359

[Password Display](#) on page 1363

[Hex Display](#) on page 1362

[Limit to Single Line](#) on page 1362

[Update Value while Typing](#) on page 1363

[Enable Wrapping](#) on page 1362

[Grouping Data with Strings](#) on page 456

[Tables](#) on page 457

Enable Wrapping

To disable word wrapping in a string constant, control, or indicator, right-click the string and remove the checkmark next to the **Enable Wrapping** shortcut menu item. Disabling word wrapping causes the string to wrap at line breaks instead. If you want to disable word wrapping, the string must be in [Normal Display](#) mode. After you disable word wrapping, you can display a horizontal [scroll bar](#) by right-clicking the string control or indicator and selecting **Visible Items»Horizontal Scrollbar** from the shortcut menu.

You also can use the Enable Wrapping property to disable word wrapping programmatically.

Hex Display

Right-click a string constant, control, or indicator and select **Hex Display** from the shortcut menu to display the ASCII value of each character in hex instead of the character itself. The **Hex Display** item is useful for debugging and communicating with instruments.

Limit to Single Line

Complete the following steps to configure a string constant, control, or indicator to limit the contents of the string to a single line.

1. Right-click a string constant, control, or indicator and select **Limit to Single Line** from the shortcut menu. This option prevents you from entering a carriage return in a string control or indicator.

2. Copy a multiline string and paste the strings in the string control or indicator. When you paste a multiline string into a string object that only allows a single line, LabVIEW pastes only the first line.

You also can use the Limit To Single Line? property to programmatically prevent entering a carriage return.

Normal Display

Right-click a string constant, control, or indicator and select **Normal Display** from the shortcut menu to display all characters as typed, with the exception of non-displayable characters. Non-displayable characters generally appear as boxes.

Password Display

Right-click a string constant, control, or indicator and select **Password Display** from the shortcut menu to display an asterisk * for each character you enter into a string control, including spaces. When you read the string data from the block diagram, you read the actual data the user entered. If you try to copy data from the control, LabVIEW copies only the * characters.

Update Value while Typing

Right-click a string constant, control, or indicator and select **Update Value while Typing** from the shortcut menu to update the value of a control as the user enters characters instead of waiting until the user presses the <Enter> key or otherwise ends text entry. Use this option to check the correctness of the input or to give user feedback.

You also can use the Update While Typing? property to update the value programmatically.

Labeling

Use the following types of labels to document VIs:

- Free labels —Document code on the block diagram and list user instructions on the front panel. On the block diagram, you can attach a free label to a block diagram object to create an attached comment.
- Owned labels —Label specific objects on the front panel or block diagram. Owned labels are owned by a specific object. You can hide owned labels, but you cannot copy or delete them independently of their owners.
- Captions —Provide instructions or information that you cannot include in the named label of an object. You also can use captions instead of labels to localize a VI without breaking the VI.

- Boolean text —Create labels for Boolean controls and indicators that change depending on the value of the control or indicator.

You can edit labels after you create them. You also can move owned labels and captions and change the text characteristics of labels.



RELATED INFORMATION

[Creating Free Labels](#) on page 1367

[Labeling Objects](#) on page 1368

[Creating Captions](#) on page 1366

[Editing the Labels of Buttons](#) on page 1190

[Editing Labels](#) on page 1368

[Moving Owned Labels and Captions](#) on page 1370

[Changing Text Characteristics](#) on page 1365

Aligning Labels and Captions to Preset Positions

Labels and captions align to preset positions directly surrounding the front panel or block diagram object. Each object has three preset positions on each side. [Labels for wires](#) do not have preset positions.



TIP

To change the default label positions of new front panel and block diagram objects, use the **Default Label Position** pull-down menus on the Front Panel and Block Diagram pages of the Options dialog box.

Complete the following steps to align a label or caption to a preset position around a front panel or block diagram object.

1. Select a label or caption in the front panel or block diagram window. As you drag the label or caption, a blue outline indicates how the object aligns with the preset positions.
2. Place the label or caption in the front panel or block diagram window, at the preset position you want.

You can disable the preset alignment positions temporarily by pressing the spacebar and then dragging the label or caption to the position you want.

Changing Captions Programmatically

Complete the following steps to change a caption programmatically.

1. Add a numeric control to the front panel window.

2. Right-click the control and select **Visible Items»Caption** from the shortcut menu.
3. On the block diagram, right-click the control, select **Create»Property Node»Caption** from the shortcut menu, and select an option from the list that appears.

You also can use Property Nodes to show, hide, and change the caption programmatically.

Related Information

Property Node

Numeric Controls and Indicators

Changing Text Characteristics

LabVIEW defines its *built-in fonts* as the default system fonts. When you move VIs between platforms, LabVIEW automatically updates its built-in fonts so that they match the default system fonts of the current platform. Additionally, if you attempt to open a VI that uses an unavailable font, LabVIEW substitutes the closest match. LabVIEW handles colors similarly to fonts. If a color is not available on a computer, LabVIEW replaces it with the closest match. You also can use *system colors* to adapt the appearance of a front panel to the system colors of any computer that runs the VI.

Complete the following steps to change the font, size, and color of text.

1. Select the specific objects for which you want to change the font, size, or color of the text. To set the font characteristics for all front panel or block diagram objects, deselect any objects that you have selected.
2. Select **Font Dialog** from the **Text Settings** pull-down menu on the toolbar to display the **Default Font** dialog box. This dialog box includes the same options as the Font Style dialog box.



NOTE

Changing the default font does not change the font of existing labels. It affects only those labels you create from that point on.

1. Select a font in the **Font** pull-down menu. LabVIEW displays all fonts currently installed on the computer.
2. Select a size by clicking the arrow button and choosing a size from the pull-down menu or by entering the size in the **Size** text box.
3. Select an alignment option in the **Align** pull-down menu.
4. Click the **Color** box and select a color for the text from the *color picker* that appears.



Tip (Windows and Linux) To make labels transparent, use the Coloring tool and right-click the T in the color picker. (macOS) To make labels transparent, use the Coloring tool and <Control>-click the T in the color picker.

1. Select how you want the text to appear by placing or removing checkmarks from the **Plain**, **Bold**, **Italic**, **Underline**, **Strikeout**, (macOS) **Outline**, and (macOS) **Shadow** checkboxes.
2. The front panel text with modifications appears in the text box showing how it will appear.
3. Place a checkmark in the **Panel default** or **Diagram default** checkboxes to use the new text settings as the default font for new front panel or block diagram labels, respectively.
4. Click the **OK** button to save these text settings or click the **Cancel** button to exit the **Default Font** dialog box without saving the changes.

You also can use the Font property to set the font programmatically, the Justification property to get and set the justification of text programmatically, and the Text Colors property to change the foreground and background color of a text string programmatically.

Creating Captions

Front panel objects also can have captions. You can use captions instead of labels to localize a VI without breaking the VI. Unlike a [label](#), a caption does not affect the name of the object, and you can use it as a more descriptive object label. The caption appears only in the front panel window.

If you [assign the object to a connector pane terminal](#), the caption appears in a tip strip when you use the Wiring tool to move the cursor over the terminal on the block diagram. The caption also appears next to the terminal in the **Context Help** window if you move the cursor over the connector pane or VI icon.

Right-click the object and select **Visible Items»Caption** from the shortcut menu to show or hide the caption.



NOTE

When the text of the caption changes, LabVIEW does not need to recompile the VI or its callers.

You also can use the Text property to change the text of a caption programmatically or use the Caption property to reference the caption.

To delete a caption, wire a reference from the Caption property to the Delete method.

Creating Free Labels

Complete the following steps to create a free label:

1. Use the Labeling tool to click any open space. If automatic tool selection is enabled, double-click any open space. You also can select a label on the **Controls** or **Functions** palette and add it to the front panel or block diagram, respectively. A small box appears with a text cursor at the left margin, ready to accept typed input.
2. Type the text you want to appear in the label. Add hashtags (#) to labels on your block diagram to create bookmarks. Add URLs (protocol :// domainname) to front panel or block diagram labels to create hyperlinks to files on the Internet or on a network. To disable hyperlinks in front panel labels, right-click the free label and deselect **Enable Hyperlinks** in the shortcut menu. You cannot disable hyperlinks in block diagram labels. You can press the <Enter> key on the keyboard to add a new line.
3. Click anywhere outside the label to complete the edit operation. You also can click the **Enter Text** button that appears on the toolbar when you begin text entry, press the <Enter> key on the numeric keypad, or press the <Ctrl-Enter> keys on the keyboard. (macOS) Press the <Option-Enter> keys. (Linux) Press the <Alt-Enter> keys.

Use free labels to add comments to a VI.



NOTE

You can complete edit operations with the <Enter> key if you place a checkmark in the **End text entry with Enter key** checkbox in the Environment page of the Options dialog box.

Creating Attached Comments

On the block diagram, you can attach a free label to a block diagram object. You can attach a comment to only one object, but you can attach multiple comments to a single object.

To create an attached comment, hover over a free label to reveal a glyph in the bottom right corner . Click and drag the glyph to a block diagram object.

You also can click the glyph and then click the object.

To detach a comment, right-click the comment and select **Detach Label From Object** from the shortcut menu.



RELATED INFORMATION

[Options Dialog Box - LabVIEW Programming Reference Manual](#)

[Managing Unfinished Tasks](#) on page 978

[Selecting a Tool](#) on page 908

Editing Labels

You cannot edit the label of a subVI. However, as you [create VI descriptions](#), you can edit function labels to reflect the use of the function on the block diagram. For example, you can use the label of an Add function to document what quantities are added or why they are added at that point on the block diagram.



TIP

Owned labels cannot be edited at run time. If you want to programmatically edit these labels, [replace each label with a caption](#). Captions look the same and can be edited at run time.

Complete the following steps to edit a free or owned label.

1. If necessary, right-click the object and select **Visible Items»Label** from the shortcut menu to display the label. The labels for structures, functions, constants, and wires are not visible by default.
2. Use the Labeling tool to click the label and highlight the text you want to edit. Double-click the label to highlight a word or triple-click the label to highlight the entire label. If [automatic tool selection is enabled](#), double-click the label to switch to the Labeling tool and highlight the text you want to edit.
3. Edit the label text. You can press the <Enter> key on the keyboard to add a new line.



NOTE

(Linux) When editing any control with text in it, such as a text label, you can use the middle mouse button to paste the most recently highlighted text without having to copy or cut the text first.

1. Click anywhere outside the label to complete the edit operation. You also can click the **Enter** button on the toolbar, press the <Enter> key on the numeric keypad, or press the <Ctrl-Enter> keys on the keyboard. (macOS) Press the <Option-Enter> keys. (Linux) Press the <Alt-Enter> keys. When the text of the owned label changes, LabVIEW must recompile the VI and possibly recompile the VI callers.



Note You can complete edit operations with the <Enter> key on the keyboard if you place a checkmark in the **End text entry with Enter key** checkbox in the Environment page of the Options dialog box.

Labeling Objects

Use owned labels to annotate specific front panel and block diagram objects. LabVIEW includes three kinds of owned labels: name labels, subdiagram labels, and unit labels.

You can [resize labels](#) and [change the text characteristics](#) of labels.

Labeling Specific Objects

Use name labels to identify specific objects, such as controls, indicators, functions, and wires. To create a name label, right-click the object, select **Visible Items»Label** from the shortcut menu, and *edit the label*.

Labeling Structures

Use subdiagram labels to describe the code contained in structures on the block diagram. For structures with multiple subdiagrams, such as Event structures and Flat Sequence structures, LabVIEW displays a label for each subdiagram. To create a subdiagram label, right-click the structure, select **Visible Items»Subdiagram Label** from the shortcut menu, and *edit the label*.

Associating Units of Measure with Numeric Controls and Indicators

Use *unit labels* to associate physical units of measure with numeric controls and indicators.

Locking Labels and Captions

You can create custom user interfaces by moving front panel control or indicator labels and captions to different positions around the front panel object. You also can move terminal and wire labels on the block diagram. After you move the label to the position you want, you can lock the label to the front panel or block diagram object. By locking the label to an object, you avoid losing the label position you select because when you move the label, the object also moves.



NOTE

You cannot lock labels to wires.

Complete the following steps to lock a label or caption to a front panel or block diagram object.

1. Move the front panel or block diagram object to the position you want.
2. Right-click the label of the control, indicator, or terminal and select **Lock Label** from the shortcut menu.

You can unlock a label or caption by completing the previous steps to remove the **Lock Label** checkmark in the shortcut menu.

You also can configure labels or captions to lock to objects automatically when you add a new control, indicator, or terminal to the front panel or block diagram. Complete the following steps to configure labels or captions to lock to new objects automatically.

1. Select **Tools»Options** to display the Options dialog box.

2. Select **Front Panel** and/or **Block Diagram** from the **Category** list.
3. Place a checkmark in the **Labels locked by default** checkbox.
4. Click the **OK** button to apply the changes and close the **Options** dialog box.
5. Add an object to the front panel or block diagram. The label is locked.

You also can use the Lock property to lock a label programmatically.

Moving Owned Labels and Captions

You can create custom user interfaces by moving front panel control or indicator labels and captions to different positions around the front panel object. You also can move terminal and wire labels on the block diagram. Complete the following steps to move a label or caption around a front panel or block diagram object.

1. Click the label or caption.
2. (Optional) Press the spacebar key to disable the *preset alignment positions*.



NOTE

Labels for wires do not have preset positions. You can move an owned label anywhere along the wire.

1. Drag the label to the position around the object you want.
2. (Optional) *Lock the label or caption* to the front panel or block diagram object.



Note Placing a label or any other object over or partially covering a control or indicator slows down screen updates and can make the control or indicator flicker.

Rotating Text Labels

You can use rotated text labels to display rotated and stacked vertical text. You can rotate any text label.

To rotate a text label, right-click the label, select **Vertical Arrangement** from the shortcut menu, and select from one of the following options:

- **None** —Label appears as normal text.
- **Stacked** —Label appears as stacked text.
- **Clockwise** —Label text appears rotated 90° clockwise. This results in vertical text that reads downward.
- **Counterclockwise** —Label text appears rotated 90° counterclockwise. This results in vertical text that reads upward.

Creating a Dialog Box

Use the Display Message to User Express VI or the Prompt User for Input Express VI to display a standard dialog box that contains an alert or a message to users or prompts users to enter information, such as a user name and password.



TIP

To keep from overusing dialog boxes, use a status text window to display less serious warnings or messages.

Complete the following steps to create a custom *dialog box* that looks like a standard dialog box across all platforms.

1. Select **File»New** to display the New dialog box, which lists all available built-in template VIs.
2. Select the Dialog Using Events template to assist in building dialog boxes. The template contains front panel and block diagram objects commonly used in dialog boxes.
3. Click **OK** to close the **New** dialog box and open the custom dialog box for you to edit.
4. Use the following guidelines while building the front panel and block diagram of the VI.
 - Use system controls to design the dialog box.
 - Use Dialog Font for all front panel text.
5. *Assign key navigation* for the front panel controls. Assign **<Enter>** for the **OK** button, **<Escape>** for the **Cancel** button, and **<F1>** to the **Help** button.



NOTE

Do not assign **<Enter>** for the **OK** button if the front panel has multi-line string controls.

1. Reduce the front panel window size to show only the objects you want to appear in the dialog box.
2. Set the front panel window appearance as **Dialog** and enter a title for the dialog box title bar.
 1. Select **File»VI Properties** and select Window Appearance from the **Category** pull-down menu of the VI Properties dialog box.
 2. Select the **Dialog** option.
 3. Remove the checkmark from the **Same as VI Name** checkbox in the **Window Title** section to create a custom title that appears in the dialog box title bar. Enter the dialog box name in the field next to the checkbox.
 4. Click the **OK** button to save the changes and close the **VI Properties** dialog box.

Aligning Objects Using the Alignment Grid

Complete the following steps to use the alignment grid to align objects as you place them.

1. If necessary, complete the following steps to display and enable the grid alignment.
 1. Select **Tools»Options** to display the Options dialog box.
 2. Select Front Panel or Block Diagram from the **Category** list.
 3. Place checkmarks in the **Show front panel grid** or **Show block diagram grid** checkboxes.
 4. Place a checkmark in the corresponding **Enable grid alignment** checkbox.
 5. Click the **OK** button to close the **Options** dialog box and save the changes.
2. Select an object from the **Controls** or **Functions** palette. In the front panel window, as you move objects before you place them, a red outline indicates how the object aligns with the grid. Press the <G> key as you drag, drop, or resize a control to enable or disable the grid alignment.
3. Add the object to the front panel or block diagram window.

Select **Edit»Disable Panel Grid Alignment** or **Edit»Disable Diagram Grid Alignment** to disable the grid alignment and use the visible grid to align objects manually. Select **Edit»Enable Panel Grid Alignment** or **Edit»Enable Diagram Grid Alignment** to enable the grid alignment again. You also can press the <Ctrl-#> keys to enable or disable the grid alignment. On French keyboards, press the <Ctrl-"]> keys.

(macOS) Press the <Command-#> keys. (Linux) Press the <Alt-#> keys.

You can use the grid options on the **Front Panel** page or the **Block Diagram** page of the **Options** dialog box to set grid options for all new VIs, and you can use the **VI Properties** dialog box to [set the size of the grid for the current VI](#).

To [align objects along axes after you place them](#), use the **Align Objects** pull-down menu on the toolbar or select **Edit»Align Items**.

Creating Splitter Bars and Panes

You can use splitter bars, such as [toolbars](#), or [status bars](#) to create professional user interfaces in the front panel window. You can add a splitter bar to a front panel that already has controls on it, or you can add a splitter bar to the front panel window of a new, blank VI. Complete the following steps to add a splitter bar to the front panel window and create multiple panes.

1. Create a new VI or open an existing one.
2. Select **Horizontal Splitter Bar** or **Vertical Splitter Bar** from the **Containers** palette.
3. Add the splitter bar to the front panel window where you want it to appear. As you drag the splitter bar to the position you want, a dashed line indicates the position of the splitter bar on the pane. If the dashed line is red, the [alignment grid](#) is enabled.

- Two scrollable panes appear in the front panel window.

Refer to the Resizable Panel with Splitter Bars VI in the `labview\examples\Controls and Indicators\Containers` directory for an example of using splitter bars and creating panes.

Creating Status Bars

You can use splitter bars such as toolbars or status bars to create professional user interfaces in the front panel window. Complete the following steps to create a status bar using a splitter bar.

- Create a new VI or open an existing one.
- Select **Horizontal Splitter Bar** from the **Containers** palette.
- Add the splitter bar to the front panel window near the bottom of the pane. As you drag the splitter bar to the position you want, a dashed line indicates the position of the splitter bar on the pane. If the dashed line is red, the **alignment grid** is enabled.
- Two scrollable panes appear in the front panel window.
- Add the controls you want to use to the status bar in the lower pane.
- Right-click the splitter bar and select **Locked**, and **Splitter Sizing»Splitter Sticks Bottom** from the shortcut menu.
- Right-click the splitter bar, select **Lower Pane»Horizontal Scrollbar»Always Off**, and select **Lower Pane»Vertical Scrollbar»Always Off** from the shortcut menu, to hide the lower scroll bars of the pane.



NOTE

You can display the scroll bars by right-clicking the splitter bar and selecting **Lower Pane»Horizontal Scrollbar»Always On** and **Lower Pane»Vertical Scrollbar»Always On** from the shortcut menu.

You can add objects to the upper pane without affecting the status bar you just created.

Creating Toolbars

Add a toolbar to your front panel window to help you create professional user interfaces.

About this task

Complete the following steps to create a toolbar using a splitter bar.

Procedure

- Create a new VI or open an existing VI.
- Select **Horizontal Splitter Bar** from the **Layout** palette

3. Add the splitter bar to the front panel near the top of the pane. As you drag the splitter bar, a dashed line indicates the position of the splitter bar on the pane. If the dashed line is red, the **alignment grid** is enabled.

The splitter bar creates two scrollable panes. Create your toolbar in the upper pane.

4. Use the **Controls** palette to add controls to the upper pane.
5. Right-click the splitter bar and select **Splitter Sizing»Splitter Sticks Top**.

LabVIEW repositions the splitter bar based on the location of the top of the front panel window.

6. Turn off the scrollbars in the upper pane.
 - a) Right-click the splitter bar and select **Upper Pane»Horizontal Scrollbar»Always Off**.
 - b) Right-click the splitter bar and select **Upper Pane»Vertical Scrollbar»Always Off**.
7. To lock the splitter bar in place, right-click the splitter bar and select **Locked**.

Configuring Splitter Bars

You can use splitter bars such as toolbars or status bars to create professional user interfaces in the front panel window. You can configure splitter bars by right-clicking the splitter bar and selecting options, such as **Splitter Sizing**, from the shortcut menu.

You also can use the Splitter properties to configure splitter bars programmatically.

Configuring Panes

You can use splitter bars to create professional user interfaces, such as toolbars or status bars. By adding splitter bars to the front panel, you separate the front panel into multiple regions, called panes. Each pane acts similar to a unique front panel. You can scroll each pane individually. Each pane has its own set of controls on it. The splitter bar can separate the controls on one pane from those on another pane, yet the terminals for all controls are on the same block diagram. You can configure panes by right-clicking the visible scroll bar and selecting options from the shortcut menu.

You also can use the Pane properties to configure panes programmatically.

Viewing Run-Time Front Panel Bounds in Edit Mode

LabVIEW can show a border representing run-time front panel bounds while you modify the front panel in edit mode.

Complete the following steps to view run-time front panel bounds and set the panel size:

1. Locate the controls and indicators you want to show on the run-time front panel.
2. Navigate to the **Window Run-Time Position** page of the VI Properties dialog box, and place a checkmark in the **Scroll panes to origin at run time** checkbox.
3. Set **Position** to **Centered** or **Custom**.
4. Remove the checkmark from the **Use Current Panel Size** checkbox.
5. Configure the size of the run-time front panel by setting **Width** and **Height**.
6. Click **OK**. A rectangular border appears at the pane origin, representing run-time front panel bounds.

**TIP**

If the pane origin is not visible in the current pane area, you can select **Edit»Adjust Pane Origin** to adjust the pane origin to the top-left corner of the current visible pane area.

Drag and drop controls and indicators into the run-time front panel bounds. When the VI is running, the front panel shows only the area within the bounds.

Creating Template Controls

Complete the following steps to create a template control.

1. Right-click a front panel *control or indicator* that you want to use as a template and select **Advanced»Customize** from the shortcut menu to display the *Control Editor* window.
2. Create a custom control to include only the components you want to reuse.
3. Select **File»Save As** to display a file dialog box.
4. Select **Template Controls** from the **Save as type** pull-down menu.
5. Enter the name of the template control and click the **OK** button.

**CAUTION**

Do **not** save your own VIs and other files in *.llb files installed with LabVIEW*.

Defining a Minimum Window Size

Complete the following steps to specify a minimum window size for front panels and block diagrams.

1. Select **File»VI Properties** to display the VI Properties dialog box
2. Select Window Size from the **Category** pull-down menu.

3. In the **Minimum Panel Size** section, enter the minimum **Width** and **Height**, in pixels, that you want to define for the window. You cannot size the window smaller than one pixel in either dimension.

**NOTE**

For single-pane front panel windows, minimum size refers to the content area of that pane, not including the scroll bars. For multi-pane front panel windows, minimum size refers to the entire front panel, including any visible scroll bars.

1. Click the **OK** button to save the changes.

Deleting Objects

To delete a front panel object, other than a splitter bar, use the Positioning tool to *select the object* and take one of the following actions:

- Select **Edit»Cut**.
- Press the <Backspace> key.
- Press the <Delete> key.

**NOTE**

Deleting a structure in this way also deletes its contents. You can *remove a structure without deleting its contents*.

You can remove a splitter bar with adjoining splitters or you can remove its entire splitter hierarchy. Complete the following steps to delete a splitter bar from the front panel.

1. Right-click the splitter bar.
2. Select **Remove with Adjoining Splitters** or **Remove Entire Splitter Hierarchy** from the shortcut menu.

If the splitter bar you select is not a parent splitter bar, LabVIEW dims the **Remove Entire Splitter Hierarchy** option on the shortcut menu.

Maintaining Window Proportions with Screen Resolution

LabVIEW can maintain its front panel window proportions relative to the resolution of the monitor. When you set a VI to maintain window proportions, the percentage of the screen the front panel window uses stays the same regardless of the user's screen resolution.

Complete the following steps to maintain front panel window proportions relative to screen resolution.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Window Size from the **Category** pull-down menu at the top of the **VI Properties** dialog box.
3. Place a checkmark in the **Maintain proportions of window for different monitor resolutions** checkbox.
4. Click the **OK** button to save the changes.

Selecting this option maintains only window proportions. To make front panel objects maintain proportions, you also must place a checkmark in the **Scale all objects on front panel as the window resizes** checkbox.

You also can use the Keep Window Proportions property to maintain front panel window proportions relative to screen resolution programmatically.

Moving Objects

Complete the following steps to move an object.

1. Use the Positioning tool to *select the object*.
2. Drag the object to a new location. You can restrict the direction and increments of movement using the following methods:
 - To restrict the direction of movement horizontally or vertically, press the <Shift> key and drag an object. Movement is restricted to the direction you first move the object.
 - To move selected objects in small, precise increments, press the appropriate arrow key once for each pixel you want to move the objects. Hold down an arrow key to repeat the action. Press the <Shift> key along with an arrow key to move the object more rapidly.



NOTE

If the *alignment grid is enabled*, you can move an object only along grid lines or dots unless you use the arrow keys to move the object. Press the <Shift> key along with an arrow key to align an object with a grid line or dot. If the object already is aligned, press the <Shift> key along with an arrow key to move the object one grid space.

To cancel a move operation, press the <Esc> key. Then release the mouse button. The object remains in its original location.



Note The Positioning tool also resizes objects. To avoid resizing an object, place the cursor in the center of the object to move it. Click the corners of an object to *resize it*.

To *align objects along axes after you place them*, use the **Align Objects** pull-down menu on the toolbar or select **Edit»Align Items**.

Resizing Labels

Use the Positioning tool to resize labels. Labels normally autosize, meaning the text rectangle automatically resizes to accommodate the text you enter. Label text remains on one line unless you enter a carriage return or resize the label box.

Right-click the label and select **Size to Text** from the shortcut menu to toggle autosizing.

When you resize a label, LabVIEW automatically turns autosizing off.

You can resize a string constant the same way you resize a label.

You also can use the Size to Text? property to set programmatically whether the size of the text object changes according to the text included.

Resizing Multiple Objects to the Same Size

Use the **Resize Objects** pull-down menu to resize multiple front panel objects to the same size. You can resize all the selected objects to the width or height of the largest or smallest object, and you can resize all the selected objects to a specific size in pixels.



NOTE

If you add splitter bars to the front panel and create panes, you can select objects on multiple panes by pressing the <Shift> key and selecting the objects you want to resize.

Complete the following steps to resize multiple front panel objects.

1. Select the objects you want to resize to the same size.
2. Select the **Resize Objects** pull-down menu, located between the *Distribute Objects* and *Reorder* pull-down menus on the toolbar on the front panel.
3. Select from the following options:
 - **Maximum Width** —Resizes the width of the selected objects to the width of the widest object.
 - **Maximum Height** —Resizes the height of the selected objects to the height of the tallest object.
 - **Maximum Width and Height** —Resizes the width of the selected objects to the width of the widest object, and resizes the height of the selected objects to the height of the tallest object.
 - **Minimum Width** —Resizes the width of the selected objects to the width of the narrowest object.
 - **Minimum Height** —Resizes the height of the selected objects to the height of the shortest object.

- **Minimum Width and Height** —Resizes the width of the selected objects to the width of the narrowest object, and resizes the height of the selected objects to the height of the shortest object.
- **Set Width and Height** —Displays the Resize Objects dialog box, which you can use to resize the selected objects to a specific size in pixels.

Some objects change size only horizontally or vertically when you resize them, such as digital numeric controls and indicators. Others keep the same proportions when you resize them, such as knobs. For example, if the selected objects whose height you want to resize include a digital numeric control, LabVIEW resizes the height of only the objects that can change size vertically and does not resize the height of the digital numeric control.

Resizing Objects

Complete the following steps to resize most front panel objects.

1. Move the Positioning tool over the object. Resizing handles ■ or ● appear at the points where you can resize the object. If the object is not resizable, resizing handles do not appear.
2. Move the cursor over a resizing handle to change the cursor to the resizing cursor ↕.
3. Click and drag the resizing handle until the border outlines the object size you want.
4. Release the mouse button. The object reappears in its new size.

To restrict the vertical or horizontal dimensions or to maintain the current proportions of the object, press the <Shift> key while you click and drag the resizing handles.

To resize an object around its center point, press the <Ctrl> key while you click and drag the resizing handles.

To restrict the vertical or horizontal dimensions or to maintain the current proportions of the object and to resize an object around its center point, press the <Ctrl-Shift> keys while you click and drag the resizing handles. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.

To resize an object as you add it to the front panel, click to place the object, hold the mouse button down, and drag the mouse until the border outlines the object size you want.



NOTE

If the *alignment grid is enabled*, you can resize an object only in units the size of the grid. To disable the alignment grid temporarily, press the <G> key before you add the object to the front panel.

Complete the following steps to cancel a resizing operation.

1. Continue dragging the resizing handle outside the window until the border disappears.

2. Release the mouse button. The object maintains its original size.



Tip You also can press the <Esc> key to cancel a resizing operation.

Use the Housing Size property to resize numeric controls and indicators with scales programmatically.

You also can [resize multiple front panel objects](#) to the same size.

Selecting Objects

For many editing operations, such as [moving](#), [copying](#), and [deleting](#), you must select an object. Clicking an unselected object or clicking an open area deselects everything currently selected. You cannot select a front panel object and a block diagram object at the same time.

Complete the following steps to select an object.

1. Move the Positioning tool until the point of the arrow is on the object you want to select. If [automatic tool selection is enabled](#), you can press the <Shift> key to temporarily switch to the Positioning tool.
2. Click the mouse button. A moving dashed outline called a marquee highlights the object.

You also can create a selection rectangle around the object(s).

Complete the following steps to create a selection rectangle.

1. Use the Positioning tool to click an open area in the front panel window or block diagram window.
2. Drag diagonally until the selection rectangle includes or touches all the objects you want to select. When you select objects, the area covered by the selection rectangle displays in gray and a marquee highlights the selected objects. Selected structures appear with a darker background to indicate your selection includes them.



NOTE

By default, when you create a selection rectangle around objects, you must enclose the entire structure or the midpoint of a wire segment to select it. If you press the spacebar when you create the selection rectangle, you select any objects that the selection rectangle touches. To restore the default selection behavior, press the spacebar again.

Press the <Shift> key and click objects to select additional objects or deselect an object.

If automatic tool selection is enabled, you must deselect the object the cursor is over to switch from the Positioning tool to another tool.

Setting a Single Front Panel Object to Scale

Right-click a front panel object and select **Scale Object with Pane** from the shortcut menu to set the object to scale automatically when the pane resizes.

After you designate a single front panel object to scale automatically, gray lines outline several front panel regions. The regions define the positions of the other front panel objects in relation to the object you want to scale. When you resize the pane, the object you set to scale automatically resizes and repositions itself relative to its original location. The gray lines disappear when you run the VI.

To set an object not to scale with the pane, right-click the front panel object and select **Scale Object with Pane** from the shortcut menu to remove the checkmark.

You also can use the Set Scaling Mode method to set a single front panel object to scale programmatically.

Setting All Objects on a Front Panel to Scale

Complete the following steps to set all front panel objects to scale automatically when the front panel window resizes.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Window Size from the **Category** pull-down menu.
3. Place a checkmark in the **Scale all objects on front panel as the window resizes** checkbox.



NOTE

LabVIEW disables this option when you [add a splitter bar](#) to the front panel. You must configure the panes individually by right-clicking the scroll bar of the pane and selecting **Pane Sizing»Scale all Objects with Pane** from the shortcut menu.

1. Click the **OK** button to save the changes. After you save the changes, resizing the front panel causes all front panel objects to resize and reposition themselves relative to their original locations.

If you select **Scale all objects on front panel as the window resizes** and then attempt to set a single object to scale, a dialog box appears that prompts you to select between setting automatic scaling for one particular object or all front panel objects.

You also can use the Set Scaling Mode method to set all front panel objects to scale programmatically.

Fitting a Single Front Panel Object to a Pane

Configure a control to be the same size as the pane by using the **Fit Control to Pane** option. The control becomes the same size as the pane and scales automatically when you resize the pane. The scroll bars on the pane also disappear.

1. Right-click the object and select **Fit Control to Pane** from the *shortcut menu*.
2. The object resizes to fit the pane.

You also can configure a control to scale automatically while the pane resizes. Right-click a *splitter bar* and select **Pane Sizing»Scale Objects While Resizing** from the shortcut menu. Any item that scales on the pane scales while you resize the pane instead of when you are finished resizing the pane.

Using Decorations on the Front Panel

Use the decorations located on the **Decorations** palette to group or separate front panel objects with boxes, lines, or arrows. These objects are for decoration only and do not display data.

After you add a decoration to the front panel, use the **Reorder** pull-down menu to *reorder objects* that are stacked on top of each other.

You also can use decorations on the block diagram.

Creating Type Definitions and Strict Type Definitions

A *type definition or a strict type definition* is a custom control or indicator linked to the saved file for that custom control or indicator. When you save a custom control or indicator as a type definition, any data type changes you make to the type definition affect all instances of the type definition in all the VIs that use it. When you save a custom control or indicator as a strict type definition, any data type and cosmetic changes you make to the strict type definition affect all front panel instances of the strict type definition. However, if you change a strict type definition, instances *placed on block diagrams* as constants act as if the link is not strict, and LabVIEW updates the constant only with data type changes.

Complete the following steps to create a type definition or a strict type definition.

1. Right-click a control, indicator, or constant you want to make a type definition and select **Make Type Def.** from the shortcut menu.
2. Right-click the instance of the type definition and select **Open Type Def.** from the shortcut menu to display the *Control Editor* window.
3. (Optional) To create a strict type definition, select **Strict Type Def.** from the **Control Type** pull-down menu on the toolbar.

4. Make the changes you want to the control or indicator. You can change the size, color, and relative position of the elements of a control or indicator and import images into the control or indicator.
5. Select **File»Apply Changes** to apply the changes to the control or indicator. The **Apply Changes** menu item is available only after you make changes to the control or indicator. If you make changes to a control or indicator and try to close the Control Editor window without selecting **File»Apply Changes**, LabVIEW displays a message asking if you want to save the changes.
6. Select **File»Save** to save the custom control or indicator as a type definition or a strict type definition. You can save the type definition or strict type definition in a directory or in an [LLB](#).

Editing Saved Type Definitions and Saved Strict Type Definitions

Complete the following steps to edit a saved [type definition](#) or a saved [strict type definition](#).

1. Select **File»Open** to display a file dialog box.
2. Navigate to the type definition or strict type definition you want to edit.
3. Select the type definition or strict type definition and double-click the type definition or strict type definition or click the **Open** button to display the type definition or strict type definition in the [Control Editor](#) window.
4. Make the changes you want to the type definition or strict type definition. Only changes to the data type of a type definition affect instances of the type definition. Almost any change you make to a strict type definition, such as changing the data type, making cosmetic changes, or changing item names in a ring control, affects all instances of the strict type definition.
5. Select **File»Save** to save the type definition or strict type definition.

Manually Updating Type Definition Instances

If LabVIEW is unable update a type definition instance automatically, LabVIEW places the instance in an unresolved state. An unresolved instance appears dimmed, and the **Run** button of the VI that contains the instance appears broken.

Complete the following steps to resolve the data mapping discrepancies and manually update unresolved type definition instances:

1. Right-click an unresolved type definition instance on the front panel or the block diagram and select **Review and Update from Type Def** from the shortcut menu to display the **Review and Update from Type Def** dialog box.
2. Select an instance from the **Instance** listbox to display the previous default values and the projected new default values of that instance.

3. Review the default values in **New Default Value** and correct any values that are incorrect. LabVIEW highlights the controls or constants that contain data that the LabVIEW mapping process may have lost or incorrectly placed.
4. Click the **Approve** button to mark the current instance as approved in the **Instance** listbox.

**NOTE**

You can still view and edit default values after approving changes. LabVIEW updates approved instances with the last values that appear in the controls or constants in **New Default Value** even if you edit the values after approving changes.

5. Repeat steps 2–4 for each instance you want to update from the type definition.

Note You also can click the **Approve All** button to approve all changes to all of the instances listed in the **Instance** listbox without reviewing each instance individually.

6. Click the **Apply Changes** button to update the current and default values of the instances you approved and close the dialog box. Instances that you do not approve remain in an unresolved state until you approve them or disconnect them from the type definition.

You also can use the Update From Typedef method to approve and apply all type definition instance updates programmatically. However, LabVIEW may lose or incorrectly preserve the default values of instances LabVIEW updates programmatically. Avoid updating type definition instances programmatically if the instances have custom default values and you plan to make extensive changes to the type definition.

Reestablishing the Link to a Type Definition or Strict Type Definition

If you want to reestablish the link between a control or indicator and a [type definition or strict type definition](#), you must replace the control or indicator with an instance of the type definition or strict type definition.

Complete the following steps to replace a control or indicator with a type definition or a strict type definition.

1. Right-click the control or indicator you want to replace.
2. Select **Replace** from the shortcut menu to display a temporary version of the **Controls** palette.
3. Click the **Select a Control** icon to display a file dialog box.
4. Navigate to the type definition or strict type definition you want to place.
5. Select the type definition or strict type definition and double-click the type definition or strict type definition or click the **OK** button to replace the front panel control or indicator.

SubVI and Express VI

Displaying SubVIs and Express VIs as Icons or Expandable Nodes

You can display VIs and Express VIs as icons or as expandable nodes. Expandable nodes appear as icons surrounded by a colored field. When displayed as expandable nodes, subVIs appear as icons surrounded by a yellow field and Express VIs appear as icons surrounded by a blue field. Use icons if you want to conserve space on the block diagram. Use expandable nodes to make wiring easier and to aid in documenting block diagrams. By default, subVIs appear as icons on the block diagram, and Express VIs appear as expandable nodes.



NOTE

If you display a subVI or Express VI as an expandable node, you cannot [enable database access](#) for that node.

Complete the following steps to display a subVI or an Express VI as an icon.

1. Right-click the subVI or Express VI.
2. Select **View As Icon** from the shortcut menu. The subVI or Express VI appears as an icon.

Remove the checkmark from **View As Icon** option to display the subVI or Express VI as an expandable node.



Note Only new subVIs you add to the block diagram appear as expandable nodes. You cannot convert all existing subVIs displayed as icons to expandable nodes. You cannot add Express VIs to the block diagram as icons by default.

You can [resize](#) an expandable subVI or Express VI. When you resize an expandable subVI or Express VI, the input and output terminals of the subVI or Express VI appear below the icon.

You also can use the Expand When Dropped As SubVI property to display a subVI as an expandable node programmatically.

Inserting and Removing Inputs and Outputs

You can insert an input or output terminal into the expandable field of a subVI or Express VI. The terminals you can insert depend on the inputs and outputs the subVI or Express VI includes. For example, you must first [assign a front panel control or indicator](#) to the connector pane of a subVI before you can insert the control or indicator into the expandable field of the subVI.

Complete the following steps to insert an input or output in a subVI or Express VI.

1. Display the subVI or Express VI as an *expandable node*.
2. Right-click an input or output in the expanded field of the subVI or Express VI and select **Insert Input/Output** from the shortcut menu. A new input or output terminal appears above the input or output you right-clicked.

To remove an input or output, right-click the terminal you want to remove and select **Remove Input** or **Remove Output** from the shortcut menu.

Express VIs

Creating an Express VI from an Existing VI

Complete the following steps to create an *Express VI* from an existing standard VI.

1. Select **Tools»Advanced»Create or Edit Express VI** to display the Create or Edit Express VI dialog box.
2. Click the **New** button to display the Initialize page of the Express VI Creator wizard.
3. Select the **Create from a VI** option.
4. Click the **Next** button.
5. In the file dialog that appears, navigate to the standard VI on which you want to base the new Express VI and click the **OK** button. The Convert VI page of the **Express VI Creator** wizard appears.
6. Determine whether you want the VI terminals to be *configurable or expandable* and place or remove checkmarks in the appropriate checkboxes.
7. Click the **Next** button to display the Setup Express VI page of the **Express VI Creator** wizard.
8. Enter a name for the new Express VI in the **Express VI name** text box.
9. Specify a location for the new Express VI in the **Express VI path** text box. The default directory is `labview\user.lib\_express`. If you save Express VIs in this directory, they appear on the Express User Libraries palette and in the **Create or Edit Express VI** dialog box.



NOTE

(macOS and Linux) The default location for Express VIs you create is the LabVIEW default data directory because not all users can write to the `user.lib` directory.

10. Click the **Finish** button to display the **Create or Edit Express VI** dialog box. The new Express VI appears in the **Express VIs** tree.
11. From the **Express VIs** list, expand the Express VI item to display the *source VI* and *configuration dialog box VI*.
12. Open the source VI to finish building it. Notice that the source VI contains all the controls and indicators from the original VI that you set as expandable, the

connector pane, and the VI icon. The VI icon of the source VI also appears as the icon for the Express VI.

13. Open the configuration dialog box VI to finish building the *front panel* and *block diagram*.
14. Save and close the VIs.

Creating an Express VI from an Existing Express VI

Complete the following steps to create an *Express VI* from an existing Express VI.

1. Select **Tools»Advanced»Create or Edit Express VI** to display the Create or Edit Express VI dialog box.
2. Click the **New** button to display the Initialize page of the Express VI Creator wizard.
3. Select the **Create from an Express VI** option.
4. Click the **Next** button to display the Copy Express VI page of the **Express VI Creator** wizard.
5. Select an Express VI or click the **Browse** item in the **Express VIs** list to navigate to the Express VI on which you want to base the new Express VI.
6. Click the **Next** button to display the Setup Express VI page of the **Express VI Creator** wizard.
7. Enter a name for the new Express VI in the **Express VI name** text box.
8. Specify a location for the new Express VI in the **Express VI path** text box. The default directory is `labview\user.lib\_express`. If you save Express VIs in this directory, they appear on the Express User Libraries palette and in the **Create or Edit Express VI** dialog box.



NOTE

(macOS and Linux) The default location for Express VIs you create is the LabVIEW default data directory because not all users can write to the `user.lib` directory.

1. Click the **Finish** button to display the **Create or Edit Express VI** dialog box. The new Express VI appears in the **Express VIs** list.
2. From the **Express VIs** list, expand the Express VI item to display the *source VI* and *configuration dialog box VI*.
3. Open the source VI to finish building it.
4. Open the configuration dialog box VI to finish building the *front panel* and *block diagram*.
5. Save and close the VIs.

Creating an Express VI from a Blank VI

Complete the following steps to create an *Express VI* from scratch.

1. Select **Tools»Advanced»Create or Edit Express VI** to display the Create or Edit Express VI dialog box.
2. Click the **New** button to display the Initialize page of the Express VI Creator wizard.
3. Select the **Create from a blank VI** option.
4. Click the **Next** button to display the Setup Express VI page of the **Express VI Creator** wizard.
5. Enter a name for the new Express VI in the **Express VI name** text box.
6. Specify a location for the new Express VI in the **Express VI path** text box. The default directory is `labview\user.lib\_express`. If you save Express VIs in this directory, they appear on the Express User Libraries palette and in the **Create or Edit Express VI** dialog box.

**NOTE**

(macOS and Linux) The default location for Express VIs you create is the LabVIEW default data directory because not all users can write to the `user.lib` directory.

7. Click the **Finish** button to display the **Create or Edit Express VI** dialog box. The new Express VI appears in the **Express VIs** list.
8. From the **Express VIs** list, expand the Express VI item to display the *source VI* and *configuration dialog box VI*.
9. Open the source VI to finish building it.
10. Open the configuration dialog box VI to finish building the *front panel* and *block diagram*.
11. Save and close the VIs.

Building Configuration Dialog Box VIs

After you *create* a new *Express VI* with the Express VI Creator wizard and design the *source VI*, you can *design the front panel* of the *configuration dialog box VI* to complete the user interface and then *modify the provided block diagram template* to configure the run-time behavior of the Express VI.

Complete the following steps to build the front panel of a configuration dialog box VI. Also use the *LabVIEW style checklist* and other development guidelines when building the user interface.

1. Open the configuration dialog box VI from the Create or Edit Express VI dialog box.
2. Place controls and indicators that you want to appear in the configuration dialog box in the **Main Selection** frame of the front panel, except for the **OK**, **Cancel**, and **Help** buttons.

**NOTE**

If you *created the Express VI from a standard VI*, the controls and indicators from the standard VI that you selected as *configurable* appear automatically on the front panel.

3. Resize the front panel to display only the **Main Selection** frame and the **OK**, **Cancel**, and **Help** buttons. The configuration dialog box maintains this setting after you save the VI.
4. Save the configuration dialog box VI.

Complete the following steps to build the block diagram of a configuration dialog box VI.

1. In the Initialize section of the block diagram, specify the configurable parameters for the Express VI in the **Control Data** array. Each element in the **Control Data** array has three fields.
 1. In the first field, **Ctl Type**, specify the type of the control, such as a front panel control, a constant, the Express VI title, or a dynamic data type control, in the source VI.
 2. In the second field, **SourceVI Ctl Label**, enter the label of the control as it appears on the front panel of the source VI.

Note You must enter the exact string, including the correct case, as it appears in the control label in the source VI. To ensure accuracy, copy and paste from the front panel label in the source VI.

3. In the third field, **ConfigDiag Ctl Label**, enter the label of the control as it appears on the front panel of the configuration dialog box VI. An empty string indicates the label is the same as the source VI label.
2. Configure the Main While Loop section of the block diagram to handle user interface feedback such as updating graphs or using radio buttons. The Event structure in this section is for any code related to direct user events such as value changes.
3. In the Reconfigure section of the block diagram, specify the *expandable terminals* for the Express VI in the **Terminals to Grow** array.
4. Save and close the configuration dialog box VI.



Note If you run the configuration dialog box VI by itself instead of launching it from the Express VI, you receive an error message. Disregard this error, which occurs because you are not running the VI as part of the Express VI.

Displaying Configuration Settings in the Context Help Window

Use the genHelp VI to *display the current configuration* of an *Express VI* in the **Context Help** window. The genHelp VI specifies the controls whose values you want to display in the **Context Help** window when you hover over the Express VI on the block diagram. You also can specify text, such as the control names, to precede the control values in the **Context Help** window. The genHelp VI appears as a subVI in the *configuration dialog box VI* for all Express VIs you *create* using the Express VI Creator wizard. The name of this subVI is genHelp ExpressVIName.

Complete the following steps to edit the genHelp VI so the **Context Help** window updates automatically to display the current configuration of the Express VI.

1. In the Reconfigure section of the configuration dialog box *block diagram*, double-click the genHelp subVI to open it.
2. On the block diagram of the genHelp VI, in the **Context Help Name** constant, enter the text that you want to precede the value of the first control in the **Context Help** window. For example, you can enter the name of the first control, followed by a colon and a space.
3. In the **Control Name** constant, enter the name of the first control whose value you want to display. Delete any other text in the constant.

**NOTE**

You must enter the exact string, including the correct case, as it appears in the **Control Data** array. To ensure accuracy, copy and paste from the front panel label.

4. If the control is enumerated, create a constant from the **Enum & T/F Strings** input of the ex_PPGetProperty subVI and enter the enumerated strings in the exact order they appear in the control.
5. If you want to display the configuration for a second control, copy the ex_PPGetProperty subVI and place the copy to the right of the original subVI. Otherwise, skip to step 14.
6. Wire the **Output Cluster** control to the **Output Cluster** input of the copy of the subVI.
7. From the original subVI, wire the output that corresponds to the data type of the first control to the **Previous String** input of the copy of the subVI. For example, if the first control is enumerated, wire the **Enum Result** output of the original subVI to the **Previous String** input of the copy of the subVI.

Note If the first control is a double-precision, floating-point numeric data type and you want to display the value of the control directly, wire the **dbl String** output of the original subVI to the **Previous String** input of the copy of the subVI. However, you might want to format the value before displaying the value in the **Context Help** window. In that case, format the **dbl** output of the original subVI, for example using the Format Into String function, and wire the resulting string to the **Previous String** input of the copy of the subVI.

8. Create a constant from the **Prefix** input of the copy of the subVI.
9. In the **Prefix** constant, enter the text that you want to precede the value of the second control in the **Context Help** window.
10. Create a constant from the **Control:** input of the copy of the subVI.
11. In the **Control:** constant, enter the name of the second control whose value you want to display.
12. Repeat step 4 if the control is enumerated.
13. Repeat steps 5 through 12 for all controls whose values you want to display in the **Context Help** window.
14. From the last ex_PPGetProperty subVI, wire the output that corresponds to the data type of the last control to the **Help Description** indicator.
15. Save and close the genHelp VI and the configuration dialog box VI.

When you place the Express VI on a block diagram, the **Context Help** window updates with the latest control values each time you close the configuration dialog box.

Validating Express VIs

You can run validation tests to detect common problems in *Express VIs* that you *create*. Complete the following steps to run validation tests on an Express VI.

1. Select **Tools»Advanced»Create or Edit Express VI** to display the Create or Edit Express VI dialog box.
2. Select the Express VI you want to validate from the **Express VIs** list.
3. Click the **Validate** button to display the Validation dialog box. The **Validation** dialog box displays a list of test results for the Express VI. A green checkmark indicates the Express VI passed the corresponding test. A red X indicates the Express VI failed the test.
4. Click a specific test name to view a description of the test.
5. Click the **OK** button to return to the **Create or Edit Express VI** dialog box.

Adding a Custom Title Option for Express VI Instances

Each instance of an *Express VI* has a different name. LabVIEW uses consecutive numbers to name Express VI instances uniquely. On the block diagram, double-click the name of the Express VI on the expandable node to edit the name of that instance. You also can create an option in the configuration dialog box to add custom titles for instances of an Express VI.

Complete the following steps to create an option for adding custom titles to instances of an Express VI.

1. Open the *configuration dialog box VI* of the Express VI.
2. Place a system string control on the front panel in the **Main Selection** frame. Label the control Express VI Name.
3. In the Initialize section of the block diagram, locate the first unused element in the **Control Data** array and select **Title** as the control type.
4. Enter Express VI Name in the **SourceVI Ctl Label** control for that element.



NOTE

You must enter the exact string, including the correct case, as it appears in the string control label on the front panel. To ensure accuracy, copy and paste from the front panel label.

5. Save and close the configuration dialog box VI.
6. Place a copy of the Express VI on a block diagram.
7. In the configuration dialog box, enter the title you want in the **Express VI Name** control.
8. Click the **OK** button to close the configuration dialog box. The new title appears on the Express VI. If you edit the Express VI to *display current configuration settings*, the new title also appears in the **Context Help** window.

Renaming and Moving Express VIs

Express VI components are linked through specific paths and naming conventions, so manually renaming or moving components breaks the Express VIs. You must use the Create or Edit Express VI dialog box to rename or move an Express VI.

Complete the following steps to rename an Express VI or move it to another location on the same computer.

1. Select **Tools»Advanced»Create or Edit Express VI** to display the **Create or Edit Express VI** dialog box.
2. Select the Express VI you want to rename or move from the **Express VIs** list.



NOTE

When you rename an Express VI, LabVIEW renames both the **source VI** and the **configuration dialog box VI**.

1. Click the **Rename** button to display the Rename Express VI dialog box.
 - If you want to rename the Express VI, enter the new name in the **New Express VI Name** text box. By default, LabVIEW appends the next consecutive number to the original Express VI name.
 - If you want to move the Express VI, enter the original name of the Express VI in the **New Express VI Name** text box and specify a new location in the **Parent Directory** text box.
2. Click the **OK** button to return to the **Create or Edit Express VI** dialog box. The renamed Express VI appears in the **Express VIs** list. If you moved the Express VI, the updated path appears in the **Path** text box.

Converting Data to the Dynamic Data Type

Complete the following steps to convert numeric, Boolean, waveform, and array data to dynamic data for use with Express VIs.

1. Add the Convert to Dynamic Data Express VI to the block diagram.
2. In the configuration dialog box that appears, specify the kind of data you want to convert to dynamic data and click the **OK** button.
3. Wire the data you want to convert to the input of the Convert to Dynamic Data Express VI.
4. Wire the output of the Convert to Dynamic Data Express VI to another Express VI, VI, or function that accepts dynamic data.



NOTE

Some VIs, such as VIs converted from Express VIs, and functions, such as the Merge Signals and Split Signals functions, accept dynamic data as an input.

Converting Dynamic Data to Other Data Types

Complete the following steps to convert dynamic data to numeric, waveform, and array data types for use with other VIs and functions.

1. Add the Convert from Dynamic Data Express VI to the block diagram.
2. In the configuration dialog box that appears, specify the kind of data to which you want to convert the dynamic data and click the **OK** button.
3. Wire any output that returns dynamic data to the **Dynamic Data Type** input of the Convert from Dynamic Data Express VI.



NOTE

Most Express VIs return dynamic data as an output.

1. Wire the output of the Convert from Dynamic Data Express VI to the VI, function, or indicator that accepts the resulting data type.

Mapping Data Using the Sensor Mapping Express VI

Map real-world data onto a 3D model using free sensors and DAQmx channels. Use the Sensor Mapping Express VI to wire an array of data values that represent sensors you configure in the Configure Sensors dialog box. You can configure sensors to represent DAQmx channels or create free sensors to represent data you wire to the Express VI. Refer to the NI-DAQmx Help for more information about using DAQmx channels.

Complete the following steps to use the Sensor Mapping Express VI to map real-world data onto a 3D model.

1. Add the Sensor Mapping Express VI to the block diagram.
2. Click the **Load Model** button in the **Configure Sensors** dialog box to display a file dialog box.
 1. Navigate to the 3D model you want to load.
 2. Click the **OK** button to select the model and close the file dialog box. LabVIEW displays the path to the 3D model in the **Model Path** field.



NOTE

LabVIEW supports the Virtual Reality Modeling Language (VRML) and stereolithography (STL) file formats for 3D models.

3. (Optional) Select a DAQmx task from the **DAQmx Task Name** pull-down menu. LabVIEW populates the **Channels List** with the channels for the DAQmx task.

Note The **DAQmx Task Name** option does not appear unless you install NI-DAQmx.

4. Drag the channels from the list onto the 3D model to add sensors. When you create sensors from a DAQ channel, the sensor represents the channel name, not the

channel data. You also can right-click anywhere on the model to add free sensors. The sensors you add to the 3D model affect the color of the 3D model at the sensor point according to the data value the sensor represents.

5. (Optional) Change the draw style of the 3D model by configuring the **Draw Style** option. Use this option to ensure accurate placement of the sensors by viewing the points, lines, and vertices of the model.
6. (Optional) Resize the sensors on the model by increasing or decreasing the **Sensor Size** option. Use this option to ensure accurate placement of the sensors.
7. Right-click the **Color Ramp** control and use the shortcut menu items to customize the appearance, size, colors, and number of colors. You also can configure the minimum and maximum value ranges for incoming data using the control. Highlight the color map minimum or maximum value, type a new value and press the <Enter> key to change the value range. After you set the value range, click the top and bottom out-of-range colors to configure the colors to indicate if a value is outside of the value range you set.
8. Click the **OK** button to save the configuration and close the dialog box.
9. [Create an array of data values](#) and wire the array to the **Sensor Values** input of the Express VI. The array positions correspond with the order of the sensors you configure in the **Configure Sensors** dialog box.

Note If the array you wire contains fewer values than the amount of sensors you add to the model, LabVIEW maps the extra sensors to the top out-of-range color.

10. Add a 3D picture control to the front panel window.
11. On the block diagram, wire the **Scene Ref Out** output of the Sensor Mapping Express VI to the 3D picture control terminal.
12. Right-click the 3D picture control terminal and select **Create»Invoke Node»AutoFocus** to create an Invoke Node. Add the node to the right of the 3D picture control terminal. The AutoFocus method ensures the camera automatically positions itself according to the 3D model.
13. Wire all [error in and error out terminals](#) on the block diagram.
14. Run the VI. LabVIEW displays the 3D model with the sensor data in the 3D picture control.

In the **Configure Sensors** dialog box, you can hold down the <Shift> key while clicking and dragging the model to zoom in and out. Press the <Ctrl> key while clicking and dragging the model to pan the display. (macOS) <Command-Shift>-click. (Linux) <Alt-Shift>-click.

To delete sensors, select a sensor in the **Channels List** and click the **Delete Sensor** button to remove the sensor from the model and the **Channels List**. If the sensor is a DAQmx channel, LabVIEW removes the sensor from the model but not from the **Channels List**.

SubVIs

Placing SubVIs on Block Diagrams

Complete the following steps to place a subVI on the block diagram.

1. Display the block diagram of a new or existing VI by selecting **Window»Show Block Diagram**.
2. If necessary, display the **Functions** palette by selecting **View»Functions Palette**.
3. Click the **Select a VI** icon or text on the **Functions** palette.
4. Navigate to and double-click the VI you want to use as a subVI, and place it on the block diagram.
5. Wire the subVI terminals to other nodes on the block diagram.

**NOTE**

If the subVI does not have a connector pane, you cannot wire the subVI to other nodes.

You can double-click the subVI node to display the front panel of the subVI.

You also can place an open VI on the block diagram of another open VI. Use the Positioning tool to click the icon in the upper right corner of the front panel or block diagram of the VI you want to use as a subVI, and drag it to the block diagram of the other VI.



Tip (Windows and macOS) Drag a VI or control from the file system to a LabVIEW block diagram to create a subVI call to that VI.

If you are using a LabVIEW project, you also can place a VI from the [Project Explorer](#) window on the block diagram of another open VI. Select the VI you want to use as a subVI from the **Project Explorer** window, and drag it to the block diagram of the other VI.

Replacing SubVI Nodes with SubVI Contents

You can insert the contents of a subVI into the block diagram at edit time by right-clicking the subVI node and selecting **Replace with SubVI Contents**. In the **Replace with SubVI Contents** dialog box that appears, configure the following options:

- **Make space in calling diagram to fit subVI contents** —Specifies whether to adjust the block diagram to accommodate the contents of the subVI.
- **Wrap subVI contents in a Sequence structure** —Specifies whether to place the contents of the subVI in a [Stacked Sequence](#) structure.

**NOTE**

The **Replace with SubVI Contents** menu item is not available if the subVI includes an indicator inside a structure.

Opening and Editing SubVIs

To open and edit a subVI, use the Operating or Positioning tool to double-click the subVI on the block diagram. LabVIEW displays the front panel of the subVI. To display the block diagram and front panel of the subVI, press the <Ctrl> key and use the Operating or Positioning tool to double-click the subVI on the block diagram. (macOS) Press the <Option> key. (Linux) Press the <Alt> key.

Creating SubVIs from Selections

Convert a section of a VI into a subVI by using the Positioning tool to select the section of the block diagram you want to reuse and selecting **Edit»Create SubVI**. An icon for the new subVI replaces the selected section of the block diagram. LabVIEW creates controls and indicators for the new subVI and wires the subVI to the existing wires.

Creating a subVI from a selection is convenient but still requires careful planning to create a logical hierarchy of VIs. Consider which objects to include in the selection and avoid changing the functionality of the original or resulting VI.

Consider the following issues when you create subVIs from VI selections:

- Creating a subVI from a selection is the same as removing the selected objects and replacing them with a subVI. In both situations, LabVIEW does the following:
 - LabVIEW does not remove any of the block diagram terminals included in the selection from the original VI. The controls or indicators remain on the front panel of the original VI and the terminals are wired to the new subVI.
 - For each front panel object with a Property Node or local variable in the selection, LabVIEW adds a control reference to the original block diagram and wires the reference to the subVI. In the subVI, LabVIEW wires the reference to a Property Node.
- Do not select more than 28 objects to create a subVI because 28 is the *maximum number of connections* on a connector pane.
- Avoid creating *cycles on the block diagram*.
- Avoid including a *structure that contains a block diagram terminal* in the selection.

Avoiding SubVI Selections That Create Cycles

Avoid creating cycles on the block diagram when you *create a subVI from a selection*. Cycles occur if a data flow originates from an output of the subVI and terminates as an input of the subVI.

Identifying cycles while making selections is difficult, but LabVIEW detects them. If LabVIEW detects a cycle, it displays a dialog box describing why it cannot create a subVI. However, you can copy the selection into a new, separate VI.

Avoiding SubVI Selections That Contain Terminals inside Structures

When you [create a subVI from a selection](#), avoid including a structure that contains a block diagram terminal in the selection. If you create a subVI from the selection, LabVIEW displays a dialog box that warns you of the potential problem and prompts you to continue or cancel the conversion.

If you create a subVI from a For Loop or a While Loop that contains a block diagram terminal, the functionality of the original VI might change. Because the terminal remains on the original block diagram and the terminal is wired to the new subVI, the subVI does not update the value of the terminal on every iteration of the loop inside the subVI. The value of the terminal updates only after all iterations of the loop execute and the subVI finishes running.

Also, if you create a subVI from a Case structure that contains an output terminal, the functionality of the original VI might change. Because the output terminal remains on the original block diagram and the indicator is wired to the new subVI, LabVIEW always writes a value to the indicator after the subVI runs. Before you created the subVI, LabVIEW wrote a value to the indicator only when the case containing the indicator executed. If you create a subVI from the Case structure, you must edit the subVI to pass a value to the indicator in all cases.

Avoiding SubVI Selections of More than 28 Objects that Require Terminals

Do not select more than 28 objects that require terminals to create a subVI from a selection because 28 is the maximum number of connections on a connector pane. Each block diagram terminal, each front panel object with a Property Node, and certain local variables you select require a terminal on the connector pane. If you select more than 28 objects that require terminals, LabVIEW displays a dialog box describing why it cannot create a subVI.

To avoid exceeding the maximum number of terminals, select a smaller section of the block diagram or group some of the objects into a cluster.



RELATED INFORMATION

[Creating SubVIs from Selections](#) on page 1396

[Selecting a Connector Pane Pattern](#) on page 1401

[Creating Cluster Controls and Indicators](#) on page 1109

Terminals

Assigning Terminals to Controls and Indicators

After you [select a pattern](#) to use for the connector pane, you can assign a front panel control or indicator to each of the connector pane terminals. When you assign controls and indicators to the connector pane, place inputs on the left and outputs on the right to prevent complicated or confusing wiring patterns.

Complete the following steps to assign terminals to controls and indicators in a connector pane.

1. Ensure that you have [selected a pattern](#) sufficient for the number of controls and indicators you want to assign to the connector pane.
2. Click a terminal of the connector pane. The tool automatically changes to the Wiring tool, and the terminal turns black.



NOTE

Although you use the Wiring tool to assign terminals on the connector pane to front panel controls and indicators, no wires are drawn between the connector pane and these controls and indicators.

1. Click the front panel control or indicator you want to assign to the terminal. A marquee highlights the object, and the terminal changes to the data type color of the control to indicate that you connected the terminal.



Note If the connector pane terminal turns white, a connection was not made. Repeat steps 2 and 3 until the connector pane terminal changes to the proper data type color.

1. Repeat steps 2 and 3 for each control and indicator you want to assign to a terminal. If you need to change the control or indicator assigned to a terminal, you must first [delete the connection](#) and repeat steps 2 and 3 to assign another control or indicator to the terminal.
2. If necessary, [confirm each terminal connection](#).

If [automatic tool selection is disabled](#), you can use the Wiring tool to select the control or indicator first and then select the terminal.

You can specify which terminals are [required](#), [recommended](#), and [optional](#).



Tip (Windows) To switch the location of two terminals, hold down the <Ctrl> key and use the Positioning tool to select the two terminals you want to switch. In this keyboard shortcut, the <Ctrl> key corresponds to the (macOS) <Option> key or to the (Linux) <Alt> key.

Confirming Terminal Connections

To confirm which control or indicator is assigned to a connector pane terminal, click the terminal on the connector pane. A marquee highlights the assigned object.

You also can use the Wiring tool to click the control or indicator. The color of the assigned terminal on the connector pane darkens.



RELATED INFORMATION

[Assigning Terminals to Controls and Indicators](#) on page 1398

Deleting Terminal Connections on the Connector Pane

You can delete [connections between terminals and the corresponding controls or indicators](#) individually or all at once.

To delete a terminal connection individually, right-click the terminal you want to disconnect on the connector pane and select **Disconnect This Terminal** from the shortcut menu. The terminal turns white to indicate that the connection no longer exists.

To delete all connections on the connector pane, right-click anywhere on the connector pane and select **Disconnect All Terminals** from the shortcut menu.

Notice that disconnecting a terminal is different from removing a terminal from the connector pane pattern. **Disconnect This Terminal** deletes the connection, but **Remove Terminal** removes the terminal from the pattern.

Opening SubVI Front Panels During Execution

A single front panel sometimes is too restrictive to present numerous options or displays. To solve this problem, organize VIs so the top-most VI presents high-level options, and subVIs present related options.



TIP

You also can use [tab controls](#) to make the front panel more usable.

When LabVIEW calls a subVI, ordinarily the subVI runs without opening its front panel. If you want a single instance of the subVI to open its front panel when called, use the SubVI Node Setup dialog box.

If you want every instance of the subVI to open its front panel when called, use the VI Properties dialog box.

Single Instance

Complete the following steps to have a single instance of a subVI front panel open when called.

1. Place a *subVI* on the block diagram.
2. Right-click the subVI and select **SubVI Node Setup** from the shortcut menu.
3. Place checkmarks in the **Show front panel when called** and **Close afterwards if originally closed** checkboxes.
4. Click the **OK** button.
5. Run the top-level VI. The subVI front panel appears when the top-level VI calls it.

Every Instance

Complete the following steps to have every instance of a subVI's front panel open when called.

1. Double-click the subVI to open it and select **File»VI Properties** to display the VI Properties dialog box.
2. Select Window Appearance from the **Category** pull-down menu.
3. Click the **Customize** button and place checkmarks in the **Show front panel when called** and **Close afterwards if originally closed** checkboxes.
4. Click the **OK** button.
5. Save and close the subVI.
6. Run the top-level VI. Every instance of the subVI front panel opens when called.

You also can use the Show Front Panel On Call and the Show Front Panel On Load properties to display the front panel programmatically.

Determining the Current Instance of a SubVI

Complete the following steps to determine the current instance of a subVI.

1. *Suspend execution of the subVI.*
2. Select the **Call list** pull-down menu that appears on the toolbar. The menu lists the chain of callers from the top-level VI down to the subVI. This is not the same list you

see when you select **View»Browse Relationships»This VI's Callers**, which lists all calling VIs regardless of whether they are currently running.

3. Select a VI from the **Call list** menu to display its block diagram and highlight the current instance of the subVI.

You also can use the Call Chain function to list the chain of callers programmatically.

Selecting a Connector Pane Pattern

To use a VI as a subVI, you need to build a **connector pane**. Complete the following steps to select a connector pane pattern for a VI.

1. The default connector pane pattern is $4 \times 2 \times 2 \times 4$. To select a different pattern, right-click the connector pane and select **Patterns** from the shortcut menu.



NOTE

Select a connector pane pattern with more terminals than necessary. Including extra terminals in the VI allows you to add additional connectors to the VI.

2. After you select a connector pane pattern, you can customize it to suit the VI by adding, removing, or rotating the terminals.
 - To add a terminal to the pattern, place the cursor where you want to add the terminal, right-click, and select **Add Terminal** from the shortcut menu.
 - To remove an existing terminal from the pattern, right-click the terminal and select **Remove Terminal** from the shortcut menu.
 - To change the spatial arrangement of the connector pane patterns, right-click the connector pane and select **Flip Horizontal**, **Flip Vertical**, or **Rotate 90 Degrees** from the shortcut menu.
3. **Assign a front panel control or indicator** to each of the connector pane terminals.
4. If you placed the VI as a subVI on another block diagram, you must relink the subVI to the VI whose connector pane you changed by right-clicking the subVI and selecting **Relink To SubVI** from the shortcut menu. Otherwise, the VI containing the subVI is broken and will not run.

National Instruments recommends you avoid using more than 16 terminals. A connector pane with more than 16 terminals can be difficult to wire. If you need to pass more data, use **clusters**.

Setting Required, Recommended, and Optional Inputs and Outputs

You can designate which inputs and outputs are required, recommended, and optional to prevent users from forgetting to wire subVI terminals.

For terminal inputs, required means that the block diagram on which you place the subVI will be broken if you do not wire the required inputs. Required is not available for terminal

outputs. For terminal inputs and outputs, recommended or optional means that the block diagram on which you place the subVI can execute if you do not wire the recommended or optional terminals. If you do not wire the terminals, the VI does not generate any warnings.

Complete the following steps to set a terminal to required, recommended, or optional.

1. Right-click a terminal on the connector pane and select **This Connection Is** from the shortcut menu.
2. Select **Required**, **Recommended**, or **Optional**.

You can configure LabVIEW to set all new terminals you configure on the connector pane to required instead of recommended. Select **Tools»Options»Front Panel** and place a checkmark in the **Connector pane terminals default to required** checkbox. This option also applies to connections made using the **Edit»Create SubVI** menu item.

You can connect only one control or indicator to a terminal.

Related Information

[Block Diagram Objects](#)

[Assigning Terminals to Controls and Indicators](#)

[Confirming Terminal Connections](#)

[Deleting Terminal Connections on the Connector Pane](#)

Icons

Creating Icons

The primary way to create icons is to use the Icon Editor dialog box. You also can drag and drop a graphic from the file system or use the VI Icon methods in the VI class to create icons.



RELATED INFORMATION

[Using Icons](#) on page 216

[Dragging and Dropping VIs, Pictures, and Text](#) on page 919

Creating a Library Icon

Create an icon to represent a [project library](#), statechart, [class](#), or [XControl](#) graphically. Use the Icon Editor dialog box to create or edit icons.

LabVIEW applies a library icon to the icon of all objects in the library. National Instruments recommends *creating only a banner for the library icon*. The icon for each object in the library then includes the banner, and you can *modify the body of the icon* to provide information about the specific object.

Creating a Project Library Icon

Complete the following steps to create an icon for a new project library.

1. Right-click the project library item in the **Project Explorer** window and select **Properties** from the shortcut menu to display the Project Library Properties dialog box. LabVIEW creates a banner as the default icon for a new project library. Notice the default project library icon in the **VI Icon Template** section of the General Settings page.
2. Click the **Edit Icon** button on the **General Settings** page to display the **Icon Editor** dialog box.
3. (Optional) Use the Fill tool on the right side of the **Icon Editor** dialog box to fill the banner of the icon with a different color.
4. Use the Text tool to enter text in the banner of the icon. While the text is active, you can move the text by pressing the arrow keys.
5. (Optional) Select **File»Save As»Template** to display the Save Icon As dialog box and save the icon as a template for later use. LabVIEW saves icon templates as 256-color .png files.
6. Click the **OK** button to save the icon information with the project library and close the **Icon Editor** dialog box. Notice that the **256 Colors** icon on the **General Settings** page of the **Project Library Properties** dialog box displays the icon you created.
7. Click the **OK** button to close the **Project Library Properties** dialog box.



TIP

You can use the **Apply Icon to VIs** button in the General Settings page of the **Project Library Properties** dialog box to automatically apply an existing project library icon to VIs in the project library.

Creating an Icon for a VI in the Project Library

Complete the following steps to create an icon for a new VI in the project library.

1. Right-click the project library item in the **Project Explorer** window and select **New»VI** from the shortcut menu to create and open a new VI. Notice that the VI icon in the upper right corner of the front panel window includes the project library icon.
2. Double-click the VI icon to display the **Icon Editor** dialog box.
3. Select **Layers»Show Layers Page** to display the **Layers** page if it is not already showing. Notice the following items on this page:

- The project library icon appears as a *user layer* called **NI_Library**.

**NOTE**

LabVIEW reverts any changes you make to the opacity or visibility of the **NI_Library** user layer the next time you modify the project library icon. If you want to modify the opacity or visibility of the **NI_Library** user layer permanently, apply the changes to the project library icon directly.

- A user layer called **NI_IconNumber** contains a number that indicates how many new VIs, up to nine VIs, you have opened since launching LabVIEW.

Note To disable this numbering, select **Tools»Options»Front Panel** and remove the checkmark from the **Use numbers in icons of new VIs (1 through 9)** checkbox.

- A user layer called **VI Icon** contains a one-pixel black frame for the icon.
 - The VI icon uses the **White w border** template, as shown in the **Icon Template** section of the **Layers** page.
4. (Optional) Click the preview of the **NI_IconNumber** user layer and press the <Delete> key to delete the layer.
 5. On the **Icon Text** page, enter up to four lines of icon text for the body of the icon. You can configure the font, alignment, size, and color of the text. If you place a checkmark in the **Center text vertically** checkbox, the **Icon Editor** dialog box centers the icon text vertically within the body of the icon.
 6. On the **Glyphs** page, drag and drop glyphs onto the **Preview** area. Press the <F> key or the <R> key to flip a glyph horizontally or rotate a glyph clockwise, respectively, as you move the glyph. You also can double-click a glyph to place the glyph in the top-left corner of the icon. You can browse glyphs by category or by keyword.
 7. Use the Move tool to move any glyph. Each glyph is on a separate layer and therefore moves separately. Notice that the rest of the icon becomes dimmed when you select a glyph so you can identify which selection you are moving.
 8. Use the editing tools on the right side of the **Icon Editor** dialog box to edit the icon further if necessary. The **Icon Editor** dialog box creates a new user layer for each non-consecutive use of the editing tools. Select **Layers»Create New Layer** to create a new user layer during consecutive uses of the editing tools.

Note You cannot modify the icon template or icon text with the editing tools on the right side of the **Icon Editor** dialog box. Use the **Templates** page and the **Icon Text** page to modify the icon template and icon text, respectively.

9. Use the **Layers** page to configure the name, opacity, visibility, and order of the layers of the icon.
10. Click the **OK** button to save the icon information with the VI and close the **Icon Editor** dialog box.

Creating a VI Icon

Create an icon to represent a VI or *custom control* graphically. Use the Icon Editor dialog box to create or edit icons.

You can use banners to identify related VIs. National Instruments recommends *creating and saving a banner as a template*. You then can use this template for a related VI icon and *modify the body of the VI icon* to provide information about the specific VI.

If you want to create an icon for an object in a library, *create a library icon* instead of using an icon template. LabVIEW applies a library icon to the icon of all objects in the library.

Saving a Banner as a Template

Complete the following steps to save a banner as an icon template for a VI.

1. Double-click the icon in the upper right corner of the front panel window or block diagram window, or right-click the icon and select **Edit Icon** from the shortcut menu, to display the **Icon Editor** dialog box.
2. Press the <Ctrl-A> keys to select all *user layers* of the icon, and press the <Delete> key to delete the selection. The default icon is a single user layer called **VI Icon**.
3. On the **Templates** page, select the `_blank.png` icon template from the **VI»Frameworks** category. You can browse templates by category or by keyword.
4. Use the Fill tool on the right side of the **Icon Editor** dialog box to fill the banner of the icon with a color.
5. Use the Text tool to enter text in the banner of the icon. While the text is active, you can move the text by pressing the arrow keys.
6. Select **File»Save As»Template** to display the Save Icon As dialog box and save the icon as a template for later use. LabVIEW saves icon templates as 256-color .png files.

Creating a VI Icon from a Template

Complete the following steps to create a VI icon that uses the template you created.

1. Press the <Ctrl-A> keys to select all *user layers* of the icon, and press the <Delete> key to delete the selection.
2. On the **Templates** page, select the template you created. You can browse templates by category or by keyword.
3. On the **Icon Text** page, enter up to four lines of icon text for the body of the icon. You can configure the font, alignment, size, and color of the text. If you place a checkmark in the **Center text vertically** checkbox, the **Icon Editor** dialog box centers the icon text vertically within the body of the icon.
4. On the **Glyphs** page, drag and drop glyphs onto the **Preview** area. Press the <F> key or the <R> key to flip a glyph horizontally or rotate a glyph clockwise, respectively, as you move the glyph. You also can double-click a glyph to place the glyph in the top-left corner of the icon. You can browse glyphs by category or by keyword.
5. Use the Move tool to move any glyph. Each glyph is on a separate layer and therefore moves separately. Notice that the rest of the icon becomes dimmed when you select a glyph so you can identify which selection you are moving.
6. Use the editing tools on the right side of the **Icon Editor** dialog box to edit the icon further if necessary. The **Icon Editor** dialog box creates a new user layer for each

non-consecutive use of the editing tools. Select **Layers»Create New Layer** to create a new user layer during consecutive uses of the editing tools.



NOTE

You cannot modify the icon template or icon text with the editing tools on the right side of the **Icon Editor** dialog box. Use the **Templates** page and the **Icon Text** page to modify the icon template and icon text, respectively.

7. (Optional) Select **Layers»Show Layers Page** to display the **Layers** page. Use this page to configure the name, opacity, visibility, and order of the layers of the icon.
8. Click the **OK** button to save the icon information with the VI and close the **Icon Editor** dialog box.

You also can *drag a graphic from anywhere in the file system* and drop it in the upper right corner of the front panel window to use the graphic as a VI icon. You can drag and drop .png, .bmp, or .jpg files.



Note If you modify an icon by dragging and dropping a graphic from the file system, LabVIEW creates a *user layer* called **VI Icon** for the graphic and deletes any other existing icon information from the **Icon Editor** dialog box.

You also can use the VI Icon:Set From File method to set the VI icon from an image programmatically. LabVIEW creates a user layer called **VI Icon** for the image and deletes any other existing icon information from the **Icon Editor** dialog box.

Creating a Custom-Shaped Icon

You can create a custom-shaped icon that is smaller than the full 32 × 32 pixel icon area. Complete the following steps to create a custom-shaped icon for a new VI.

1. Double-click the icon in the upper right corner of the front panel window or block diagram window to display the **Icon Editor** dialog box.
2. Press the <Ctrl-A> keys to select all *user layers* of the icon, and press the <Delete> key to delete the selection. The default icon is a single user layer called **VI Icon**.
3. *Create an icon* with a closed shape border. For example, use the Ellipse tool to draw a closed ellipse and add glyphs or text only within the ellipse.
4. Click the **OK** button to save the icon information with the VI and close the **Icon Editor** dialog box.

When you select a custom-shaped icon on the block diagram, a selection marquee appears around each individual graphical element in the icon. For example, suppose the icon has no border but consists of a circle and a square that do not overlap. If you select the icon on the block diagram, a selection marquee appears around both the circle and the square.

Synchronizing with the Icon Library Manually

When you select **Tools»Synchronize with ni.com Icon Library** from the Icon Editor dialog box, LabVIEW attempts to download the most recent glyphs from the Icon Library at `ni.com`. If LabVIEW cannot reach the National Instruments Web server, LabVIEW displays the Synchronize Manually dialog box. Use this dialog box to synchronize with the Icon Library manually.

Complete the following steps to synchronize with the Icon Library manually when the **Synchronize Manually** dialog box appears.

1. Search for *Glyphs and Templates Are Missing in LabVIEW Icon Editor* on `ni.com` or click the link in *Related information*.
2. From the article on `ni.com`, download the `Icon_Library.zip` file to an easily accessible location.
3. Copy and save the Icon Library XML file to an easily accessible location. The Icon Library XML file determines the structure of the files in the `Icon_Library.zip` file.
4. In the **Synchronize Manually** dialog box, enter the path to the `Icon_Library.zip` file in the **Icon library ZIP file** field.
5. Enter the path to the Icon Library XML file in the **Icon library XML file** field.
6. Click the **OK** button. LabVIEW populates the Synchronize with Icon Library dialog box with all icons from the `.zip` file and with a structure that the XML file defines.



RELATED INFORMATION

[Glyphs and Templates Are Missing in LabVIEW Icon Editor](#)

Menus

Customizing Menus

You can create custom menus for every VI you build using the Menu Editor dialog box, and you can configure VIs to show or hide menu bars. You also can associate an existing run-time menu `.rtm` file with a VI.



NOTE

Custom menus appear only while the VI runs.

You can build custom menus or modify the default LabVIEW menus statically when you edit the VI or programmatically when you run the VI.



RELATED INFORMATION

[Showing and Hiding Menu Bars](#) on page 1412

[Associating Existing Menus](#) on page 1408

[Customizing Menus Statically](#) on page 1410

[Customizing Menus Programmatically](#) on page 1408

Associating Existing Menus

You can use an existing run-time menu `.rtm` file with more than one VI.

Complete the following steps to associate an existing `.rtm` file with a VI.

1. Select **Edit»Run-Time Menu** to display the Menu Editor dialog box.
2. Select **File»Open** in the **Menu Editor** dialog box.
3. Navigate to the menu file you want to associate with the VI and click the **OK** button. Notice that the path to the menu file is displayed in the title bar of the **Menu Editor** dialog box.
4. Close the **Menu Editor** dialog box.
5. When asked if you want to change the run-time menu to the selected menu file, click the **Yes** button.

Complete the following steps to disassociate a run-time menu from a VI.

1. Select **Edit»Run-Time Menu** to display the Menu Editor dialog box.
2. Select **Default** from the **Menu Type** pull-down menu. Notice that a file path no longer appears on the title bar of the **Menu Editor** dialog box.
3. Close the **Menu Editor** dialog box.
4. When asked if you want to change the run-time menu to the **Default** menu, click the **Yes** button.

You can customize menus *statically* and *programmatically*.

Customizing Menus Programmatically

Complete the following steps to customize a menu programmatically.

1. Select **Edit»Run-Time Menu** to display the Menu Editor dialog box.

2. Select **Custom** from the **Menu Type** pull-down menu. The question marks in the **Hierarchy** listbox indicate that you are creating a new item that does not have a name yet.
3. Select **File»Save** and name the custom menu. You can save the menu file wherever you choose.
4. Close the **Menu Editor** dialog box.
5. Add the Current VI's Menubar function to the block diagram.
6. Add the Insert Menu Items function to the block diagram.
7. Wire the Current VI's Menubar function to the **menu reference** input of the Insert Menu Items function.
8. Right-click the **item names** input of the Insert Menu Items function and select **Create»Constant** from the shortcut menu. An array constant appears.
9. Type **File** in the 0 slot and **Edit** in the 1 slot of the array.
10. Right-click the **item tags** input of the Insert Menu Items function and select **Create»Constant** from the shortcut menu. An array constant appears.
11. Again, type **File** in the 0 slot and **Edit** in the 1 slot of the array. Menu item tags can have the same name as menu item names.
12. Add a While Loop to the block diagram so the While Loop encloses the menu functions.
13. Add the Wait (ms) function inside the While Loop to delay execution.



NOTE

You must wire a value to the **milliseconds to wait** input of the Wait (ms) function for the VI to run.

1. Switch to the front panel window and click the **Run Continuously** button. A **File** and **Edit** menu appear in the front panel window. However, there are no submenu items.

Adding Submenu Items to Existing File and Edit Menus

Complete the following steps to add submenu items to existing **File** and **Edit** menus.

1. Add another Insert Menu Items function to the block diagram to the right of the first Insert Menu Items function.
2. Wire the **menu reference out** and **error out** output of the first Insert Menu Items function into the **menu reference** and **error in** input of the second Insert Menu Items function.
3. *Wire all error in and error out terminals* on the block diagram.
4. Right-click the **menu tag** input of the second Insert Menu Items function and select **Create»Constant** from the shortcut menu. A string constant appears.
5. Type **File** in the string to add items in the **File** menu you created in step 9.
6. On the second Insert Menu Items function, right-click the **item names** input and select **Create»Constant** from the shortcut menu. An array constant appears.

7. Type **Menu1** in the 0 slot and **Menu2** in the 1 slot. Repeat this step for the **item tags** input.
8. Switch to the front panel and click the **Run Continuously** button. The front panel window includes a **File** menu and an **Edit** menu with **Menu1** and **Menu2** as subitems in the **File** menu.
9. [Use a Case Structure](#) to execute the menu items.

**NOTE**

You also can add default LabVIEW menu items by providing the [application item tags](#) as inputs to the Insert Menu Item function.

Refer to the `labview\examples\Dialog and User Interface\Menu\Customizing the Menubar\Customizing the Menubar.lvproj` for examples of customizing menus programmatically.

Customizing Menus Statically

Complete the following steps to customize a menu statically.

1. Select **Edit»Run-Time Menu** to display the Menu Editor dialog box.
2. Select **Custom** from the **Menu Type** pull-down menu.
3. In the **Item Type** pull-down menu, select **User Item**. To add LabVIEW default menu items to a custom menu, select Application Item and select the LabVIEW menu items you want.

**NOTE**

If you want to use LabVIEW default [keyboard shortcuts](#) in a custom menu, you must add their corresponding default menu items to the custom menu. However, the default keyboard shortcuts for cutting an object, copying an object, pasting an object, displaying the Context Help window, locking the Context Help window, and displaying the LabVIEW Help are included in the custom menu by default.

1. In the **Item Name** text box, type a menu item name. The menu item name appears in the **Preview** field and the **Item Tag** field. You can modify the tag name to give the menu item a unique identifier. The block diagram uses the tag name to identify the menu item programmatically. Menu item tags can have the same name as menu item names. (Windows) In the **Item Name** text box, enter an underscore _ in front of the letter you want the user to press with the <Alt> key to open the menu from the keyboard. If the item name begins with two underscores __, LabVIEW removes the leading underscores when the item name appears on the menu but includes all other underscores you entered. Use this feature when menu items contain underscore characters but you do not want any characters to be underlined.

Note If you enter two underscores at the beginning of the item name, you prevent LabVIEW from assigning shortcut keys to the <Alt> key, but you can still assign a shortcut key to the item name in the **Shortcut** text box.

2. Place a checkmark in the **Enabled** checkbox to make the item available to the user. Removing the checkmark dims the menu item, making it unavailable to the user.
3. Place a checkmark in the **Checked** checkbox to cause a checkmark to appear next to the menu item. A checkmark is useful for operations that continuously run until the user removes the checkmark from the menu item.
4. Create a keyboard shortcut in the **Shortcut** text box. You can use the <F1> to <F24> function keys without using the <Ctrl> key as a modifier key. You must use the <Ctrl> menu key in addition to any other key. (macOS) Use the <Command> key. (Linux) Use the <Alt> key.



Note LabVIEW does not respond to keyboard shortcuts for dimmed menu items. (macOS) Keyboard shortcuts for the function keys work only if the function keys are not assigned as shortcut keys for Exposé.

1. Click the blue plus button  on the toolbar to add more items to the custom menu. Click the red X button  to delete items.
2. Select **File»Save As** to save the menu file. In the dialog box that appears, name the menu and click the **OK** button. When you load the VI, the customized menu automatically loads.
3. Use a **Case Structure** to execute each menu item.

You can arrange the menu hierarchy in the **Menu Editor** dialog box by clicking the arrow buttons on the toolbar, using the hierarchy manipulation options in the **Edit** menu, or by dragging and dropping. You can expand or collapse hierarchies by clicking the submenu glyphs or by choosing from the expand or collapse options in the **Edit** menu. To add separators, select **Separator** in the **Item Type** pull-down menu.

Handling Menu Selections

After you customize a menu *statically* or *programmatically* you must create a Case structure that executes each menu item in the custom menu.

Complete the following steps to handle menu items you created in a custom menu.

1. Add a While Loop to the block diagram and drag out a region on the block diagram.
2. Add the Current VI's Menubar function outside the While Loop.
3. Add the Get Menu Selection function inside the While Loop.
4. Wire the Current VI's Menubar function to the **menu reference** input of the Get Menu Selection function.
5. Right-click the **ms timeout** input of the Get Menu Selection function and select **Create»Constant** from the shortcut menu. A numeric constant with a value of 200 appears.
6. Add a Case structure inside the While Loop and drag out a region inside the While Loop.

7. Wire the **item tag** output of the Get Menu Selection function to the selector terminal on the border of the Case structure.
8. Wire the **menu reference out** output of the Get Menu Selection function to the border of the Case structure.
9. [Create a case](#) for each menu item you created in the custom menu.
10. Run the VI to view the custom menu.

Refer to the `labview\examples\Dialog and User Interface\Menu\Customizing the Menubar\Customizing the Menubar.lvproj` for examples of handling menu selections.

Showing and Hiding Menu Bars

Complete the following steps to show and hide menu bars.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Window Appearance from the **Category** pull-down menu.
3. Click the **Customize** button and place a checkmark in or remove a checkmark from the **Show menu bar** checkbox.
4. Click the **OK** button twice to save the changes.

Viewing Application Items

Knowing where to view application item tags might make it easier to call these items when you modify menus programmatically. You can use application item tags for run-time menus for VIs or for run-time shortcut menus.

Complete the following steps to view application item tags.

1. From the front panel window, select **Edit»Run-Time Menu** to display the Menu Editor dialog box. You also can right-click a control or indicator and select **Advanced»Run-Time Shortcut Menu»Edit** to display the Shortcut Menu Editor dialog box.
2. Select **Default** from the **Menu Type** pull-down menu.
3. In the **Menu Hierarchy** listbox, select the menu item for which you want to view the **Item Tag**, which appears on the right.

Customizing Run-Time Shortcut Menus

You can customize run-time shortcut menus of controls in your user interface. To customize a shortcut menu, right-click a control and select **Advanced»Run-Time Shortcut Menu»Edit** from the shortcut menu.

You also can disable run-time shortcut menus for a VI or an individual control or indicator.



RELATED INFORMATION

[Customizing Run-Time Shortcut Menus for Controls Statically](#) on page 1415

[Disabling Run-Time Shortcut Menus](#) on page 1416

Associating Existing Run-Time Shortcut Menus

You can use an existing run-time shortcut menu `.rtm` file with more than one control.

Complete the following steps to associate an existing `.rtm` file with a control.

1. Right-click a control and select **Advanced»Run-Time Shortcut Menu»Edit** from the shortcut menu to display the Shortcut Menu Editor dialog box.
2. Select **File»Open** in the **Shortcut Menu Editor** dialog box.
3. Navigate to the shortcut menu file you want to associate with the control and click the **OK** button. Notice that the path to the menu file is displayed in the title bar of the **Shortcut Menu Editor** dialog box.
4. Close the **Shortcut Menu Editor** dialog box.
5. When asked if you want to change the run-time shortcut menu to the selected menu file, click the **Yes** button.

Complete the following steps to disassociate a run-time shortcut menu from a control.

1. Right-click a control and select **Advanced»Run-Time Shortcut Menu»Edit** from the shortcut menu to display the Shortcut Menu Editor dialog box.
2. Select **Default** from the **Menu Type** pull-down menu. Notice that a file path no longer appears on the title bar of the **Shortcut Menu Editor** dialog box.
3. Close the **Shortcut Menu Editor** dialog box.
4. When asked if you want to change the run-time shortcut menu to the **Default** menu, click the **Yes** button.

You can customize run-time shortcut menus *statically* and *programmatically*.

Customizing Run-Time Shortcut Menus for Front Panels Programmatically

Complete the following steps to customize a shortcut menu for the front panel of a VI programmatically.

1. Add an Event structure to the block diagram and drag out a region on the block diagram.

2. Right-click the Event structure and select **Edit Events Handled by This Case** from the shortcut menu.
3. In the Edit Events dialog box, select **Pane** from the **Event Sources** list and select Shortcut Menu Activation? from the **Events** list.

**NOTE**

LabVIEW labels the first pane as **Pane**. LabVIEW labels subsequent panes that you create by adding splitter bars to the front panel as **Pane X**, where X numbers the pane according to the order in which you create the pane. You can view the label of the pane by right-clicking the scroll bar of the pane and selecting **Visible Items»Labels** from the shortcut menu.

4. Click the **OK** button to close the dialog box.
5. Add the Insert Menu Items function to the block diagram inside the Shortcut Menu Activation? Event structure.
6. Right-click the **item names** input of the Insert Menu Items function and select **Create»Constant** from the shortcut menu. An array constant appears.
7. Type `About this Application` in element 0 and `Help` in element 1 of the string array constant.
8. Wire the **MenuRef** event data field to the **menu reference** input of the Insert Menu Items function.
9. Click the **Run Continuously** button.
10. Right-click in the front panel window. The shortcut menu includes an **About this Application** item and a **Help** item.
11. [Configure a Shortcut Menu Selection event](#) to handle the selection of the items.

Customizing Run-Time Shortcut Menus for Controls Programmatically

Complete the following steps to customize a menu for a control programmatically.

1. Add a control to the front panel. This example uses a numeric control.
2. Add an Event structure to the block diagram. It is not necessary to enclose the numeric control in the Event structure.
3. Right-click the Event structure and select **Edit Events Handled by This Case** from the shortcut menu.
4. In the Edit Events dialog box, select **Numeric** from the **Event Sources** list and select Shortcut Menu Activation? from the **Events** list.
5. Click the **OK** button to close the dialog box.
6. Add the Insert Menu Items function to the block diagram inside the Shortcut Menu Activation? Event structure.
7. Right-click the **item names** input of the Insert Menu Items function and select **Create»Constant** from the shortcut menu. An array constant appears.
8. Type `Help` in element 0 and `Copy` in element 1 of the string array constant.

9. Wire the **MenuRef** event data field to the **menu reference** input of the Insert Menu Items function.

**TIP**

If you do not want the default shortcut menu items to appear, use the Delete Menu Items function. To remove the default shortcut menu items, wire the **MenuRef** event data field to the **menu reference** input of the Delete Menu Items function and then wire the **menu reference out** output to the **menu reference** input of the Insert Menu Items function.

1. Click the **Run Continuously** button.
2. Switch to the front panel and right-click the numeric control. The shortcut menu includes a **Help** item and a **Copy** item.
3. *Configure a Shortcut Menu Selection event* to handle the selection of the items.

**NOTE**

You also can add default shortcut menu items by providing application item tags as inputs to the Insert Menu Items function.

Customizing Run-Time Shortcut Menus for Controls Statically

Complete the following steps to customize a run-time shortcut menu for a control statically.

1. Right-click a control and select **Advanced»Run-Time Shortcut Menu»Edit** from the shortcut menu to display the Shortcut Menu Editor dialog box.

**NOTE**

Controls or indicators with customized run-time shortcut menus that you add into an array shell use the shortcut menu of the array.

1. Select **Custom** from the **Menu Type** pull-down menu of the dialog box.
2. Select **User Item** from the **Item Type** pull-down menu. To add LabVIEW default menu items to a custom run-time shortcut menu, select Application Item and select the LabVIEW shortcut menu items you want.
3. In the **Item Name** text box, type the text you want for this item. The same text appears in the **Item Tag** field. You can modify the tag name to give the shortcut menu item a unique identifier. The block diagram uses the tag name to identify the shortcut menu item programmatically. Shortcut menu item tags can have the same name as shortcut menu item names.
4. Click the blue plus button  on the toolbar to add more items to the shortcut menu. Click the red button  to delete items.

5. Select **File»Save**. In the dialog box that appears, select whether you want to save the shortcut menu with the control or save the shortcut menu to a file. If you select **Save to file**, the dialog box appears, allowing you to name the shortcut menu and select a location on disk to save to. When you load the VI, the customized shortcut menu automatically loads.
6. *Configure a Shortcut Menu Selection (User) event* to handle the selection of the items.

You can arrange the shortcut menu hierarchy in the dialog box by clicking the arrow buttons on the toolbar, using the hierarchy manipulation options in the **Edit** menu, or by dragging and dropping. You can expand or collapse hierarchies by clicking the submenu glyphs or by choosing from the expand or collapse options in the **Edit** menu. To add separators, select **Separator** in the **Item Type** pull-down menu.

Disabling Run-Time Shortcut Menus

Complete the following steps to disable all run-time shortcut menus for a VI so users cannot right-click objects and change their appearance.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Window Appearance from the **Category** pull-down menu.
3. Click the **Customize** button to display the Customize Window Appearance dialog box.
4. Remove the checkmark from the **Allow default run-time shortcut menus** checkbox.
5. Click the **OK** button twice to save the changes.

To disable the run-time shortcut menus for an individual control or indicator, right-click the object and select **Advanced»Run-Time Shortcut Menu»Disable** from the shortcut menu.

Handling Run-Time Shortcut Menu Selections

After you customize a run-time shortcut menu statically or programmatically, you must configure an Event structure to handle each menu item in the custom menu.

Complete the following steps to handle menu items you created in a custom menu.

1. Add an Event structure to the block diagram.
2. Right-click the Event structure and select **Edit Events Handled by This Case** from the shortcut menu.
3. In the Edit Events dialog box, select a control or pane from the **Event Sources** list and select **Shortcut Menu Selection (User)** from the **Events** list.
4. Click the **OK** button to close the dialog box.
5. Add a While Loop to the block diagram around the Event structure.

6. Add a Case structure inside the Shortcut Menu Selection (User) case of the Event structure.
7. Wire the **ItemTag** event data field to the selector terminal on the border of the Case structure.
8. Enter the tag name of the shortcut menu item in the case selector label.
9. Create a case for each shortcut menu item you created. For example, if you have a **Help** item, you can use the Control Online Help function to open a help file when a user selected **Help** from the shortcut menu.



RELATED INFORMATION

[Customizing Run-Time Shortcut Menus for Controls Statically](#) on page 1415

[Customizing Run-Time Shortcut Menus for Controls Programmatically](#) on page 1414

[Entering Values for each Case Structure Subdiagram](#) on page 1293

[Adding and Duplicating Subdiagrams](#) on page 1289

Troubleshooting

Probes

Caveats and Recommendations When Using Custom Probes

The following list describes some of the caveats and recommendations to consider when you create and use custom probes:

- Custom probes only execute when data flows through the wire. That is, although you can operate the controls on a custom probe at any time, programmatic responses to the changed control values occur only when the probe receives data from the wire.
- You can debug a custom probe similar to a VI. However, a probe cannot probe its own block diagram, nor the block diagram of any of its subVIs. When debugging probes, use the [generic probe](#).
- Save probes in the `labview\user.lib\_probes\default` or the [Default Data\Probes](#) directories. Probes saved in the `labview\user.lib\_probes\default` directory are available to anyone running LabVIEW on the computer remotely. This directory is useful for public probes. Because the Default Data directory changes based on which user is logged in to the computer, National Instruments recommends you save non-public probes in this directory.
- The main block diagram pauses until the probe finishes execution. Avoid using loops or Event structures in custom probes as these elements can increase the time it takes the probe to finish executing.
- Be careful when using uninitialized shift registers in custom probes. Even if a VI is non-reentrant, when that VI is used as a probe, all uninitialized shift registers

behave the same as [reentrant](#) VIs. Each probe retains the last value of the loop and does not share these values with other instances of the same probe.

Creating Custom Probes

Create a probe when you want to have more control over how LabVIEW probes the data that flow through a wire. When you create a new probe, the data type of the probe matches the data type of the wire you right-click. Refer to the [caveats and recommendations](#) before you begin creating custom probes.

Complete the following steps to create a custom probe.



NOTE

If you are creating a custom probe for a LabVIEW [class](#) or [interface](#) data type, some of the steps listed below may vary depending upon whether you are creating a custom probe that is a member VI of the class or interface or a custom probe that is not a member VI of the class or interface. The changes are noted where appropriate. A custom probe that is a member VI of a class is a member probe and has the ability to access the private data of the class directly. Member probes can display any aspect of the private class data. A non-member custom probe may only display information accessible via the public interface of the class.

1. Right-click a wire and select **Custom Probe»New** from the shortcut menu to display the Create New Probe dialog box.
2. Select whether to create a new probe or to create a probe based on an existing probe.
 - Select **Create a probe from an existing probe** to create a probe based on an existing probe. Existing probes can be the probes LabVIEW supplies or other probes you created using the **Create New Probe** dialog box.
 1. Click the **Next** button and select an existing probe on which you want to base the new probe. LabVIEW displays all [supplied probes](#) in the `labview\vi.lib\_probes` directory and any custom probes available.
 2. Click the **Next** button.
 3. Enter the filename, window title and shortcut menu name, and directory where you want to save the probe. If you do not specify a directory, LabVIEW stores custom probes in the Probes subdirectory of the [default data directory](#). If you want a custom probe to be the default probe for a particular data type, save the probe in the `labview\user.lib\_probes\default` directory. Do not save probes as existing files in the `labview\vi.lib\_probes` directory because LabVIEW overwrites the existing files when you uninstall or reinstall.

4. Click the **Save** button. LabVIEW opens a probe that matches the existing probe you selected.



NOTE

If you are creating a non-member probe for a LabVIEW class or interface, follow the instructions above. If you are creating a member probe for a LabVIEW class or interface, you can save the custom probe in the subdirectory where you save the other member VIs of the class or interface to ensure that the class or interface items remain together for easy distribution.

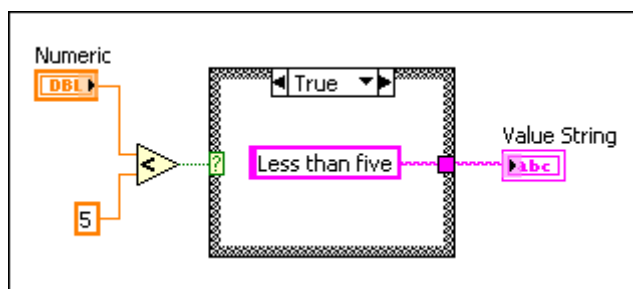
- Select **Create a new probe** to create a new probe.
 1. Click the **Next** button.
 2. Enter the filename, window title and shortcut menu name, and directory where you want to save the probe.
 3. Click the **Save** button. LabVIEW opens a probe that contains a control that is the data type of the wire you right-clicked and a Boolean indicator that acts like a conditional breakpoint. The probe also contains the **To Probe String** function and a string indicator you can use to configure the value LabVIEW displays in the **Value** column of the **Debug** window. Do not delete the control or indicators or change the connector pane because doing so breaks the probe.
 4. (Optional) Wire any data type to the **data value** input of the To Probe String function to see the data as a string in the **Value** column of the **Debug** window. The function breaks down the data and converts it into a string that the **Value String** indicator can display in the **Value** column. The To Probe String function also compresses and trims the data to fit the available space. Use this function to extract relevant information from complex data types.



NOTE

LabVIEW automatically wires the control to the To Probe String function. You can remove the To Probe String function to specify the string you want LabVIEW to display in the **Value** column.

5. (Optional) Specify the string you want LabVIEW to display in the **Value** column of the **Debug** window by wiring that string to the **Value String** indicator, as shown in the following block diagram.



The previous figure shows a custom probe of a wire that contains numeric data. You can use a [Case Structure](#) to tell LabVIEW to display

Greater than five or Less than five rather than the default, which is to display the data that passes through the wire.

6. When you finish customizing the probe, save and close it. If you want to edit that probe, you must open it from the directory where you saved it.



NOTE

If you are creating a probe for a LabVIEW class or interface data type, the **Create New Probe** dialog box does not make the new probe a member of the LabVIEW class or interface by default. If you want the probe to be a member probe of the class or interface, you should [add the member probe to the class or interface](#).

To use the probe you created, right-click the wire again and select **Probe** from the shortcut menu. After you create a probe, that probe becomes the default probe for that data type, and LabVIEW loads that probe when you right-click the wire and select **Probe** from the shortcut menu. To change the default probe for a data type, right-click a wire of that data type, select **Custom Probe** from the shortcut menu, and select a supplied, custom, or generic probe from the shortcut menu.



NOTE

If you create a probe for a LabVIEW class or interface and then move the LabVIEW class or interface to a different machine or you reinstall LabVIEW on the current machine, the default probe that users of the class or interface can use is the generic probe. A custom probe is not the default probe for any data type until you use it on the machine. However, if you create a member probe you can use the Class/Interface Properties dialog box to [set the custom probe as the initial default probe for the class or interface](#). Now when you distribute the class or interface to a new machine, the default probe the user of the class or interface can use is the custom probe rather than the generic probe.

To edit an existing probe, open the probe VI or use the **Create New Probe** dialog box to create a new probe based on the existing probe.



RELATED INFORMATION

[Debug Window - LabVIEW Programming Reference Manual](#)

Creating Probes after a VI Runs

You can configure a VI to retain wire values so that when you [create a probe](#) on the block diagram, the probe displays the data that flowed through the wire at the last VI execution. Using probes after a VI runs is useful when the block diagram is complex and you want to debug the VI.

Complete the following steps to create a probe after a VI runs.

1. Click the **Retain Wire Values** button  on the block diagram toolbar to retain wire values for use with probes.

2. Run the VI. The VI must execute at least once so that wire values can be available immediately to any probes you create.
3. Stop the VI.
4. Place a probe on a wire by right-clicking the wire and selecting **Probe**. Notice that the most recent value of the data that passed through the wire appears in the subpanel of the **Debug** window.

Click the **Do Not Retain Wire Values** button  at any time to refrain from saving wire values at each execution.



NOTE

Choose the **Do Not Retain Wire Values** option to reduce memory requirements and improve performance slightly.



RELATED INFORMATION

[Debug Window - LabVIEW Programming Reference Manual](#)

Using Indicators to View Probed Data

In addition to creating a [custom probe](#) or using one of the [supplied probes](#), you also can use an indicator to view the data points that pass through a wire.

Complete the following steps to use an indicator to view probed data.

1. Right-click a wire and select **Custom Probe»Controls** from the shortcut menu.
2. Select an indicator on the **Controls** palette to display the data. For example, you can use a chart to view numeric data. You also can click **Select a Control** and select any [custom control or type definition](#) saved on the computer or in a shared directory on a server or navigate to a custom control you created using the Create New Probe dialog box. The data type of the indicator you select must match the data type of the wire you right-clicked.

Using Supplied Probes

Use supplied probes to display comprehensive information about the data that flows through a wire. You also can use a supplied probe to respond based on the data that flows through the wire.

Complete the following steps to use a supplied probe.

1. Right-click a wire and select **Custom Probe** from the shortcut menu.
2. Select among the following supplied probes. All supplied probes that match the data type of the wire you right-clicked appear at the top of the shortcut menu.

- **.NET Refnum Probe** —Displays the result of calling the ToString() method on the object represented by the .NET refnum, the type of the object, the hash code of the object, and the hex value of the reference. You can use this probe to set a breakpoint if the value is an invalid refnum.

**NOTE**

This probe is only supported for .NET Framework and not .NET 8.0 (Core).

- **Analog Waveform Probe (DBL)** —Displays a graph of the analog data; a table with a time stamp that specifies when the probe received the data, the x value, and the time; and a table that displays the attributes of the waveform data.
- **Conditional Boolean Probe** —Displays the Boolean data and allows conditional breakpoints.
- **Conditional Double Array Probe** —Monitors an array of double-precision, floating-point numeric controls. The **Data** tab displays the size of the input array. Use the **Conditions** page to set a conditional breakpoint. You can set conditions based on the size of the array or any elements within the array. If you select multiple conditions, LabVIEW executes a breakpoint if any of those conditions are true.
- **Conditional Double Probe** —Displays the double-precision, floating-point data and allows conditional breakpoints. If you select multiple conditions, LabVIEW executes a breakpoint if any of those conditions are true.
- **Conditional Error Probe** —Displays the error status, code, and description. Allows conditional breakpoints to pause on an error, a warning, or on a particular error code.
- **Conditional Signed8 Probe** —Displays the 8-bit signed integer data and allows conditional breakpoints.
- **Conditional Signed16 Probe** —Displays the 16-bit signed integer data and allows conditional breakpoints.
- **Conditional Signed32 Array Probe** —Similar to the Conditional Double Array probe, this probe monitors an array of 32-bit signed integer numeric controls. The **Data** page displays the size of the input array. Use the **Conditions** page to set a conditional breakpoint. You can set conditions based on the size of the array or any elements within the array. If you select multiple conditions, LabVIEW executes a breakpoint if any of those conditions are true.
- **Conditional Signed32 Probe** —Displays the 32-bit signed integer data and allows conditional breakpoints.
- **Conditional Signed64 Array Probe** —Similar to the Conditional Double Array probe, this probe monitors an array of 64-bit signed integer numeric controls. The **Data** page displays the size of the input array. Use the **Conditions** page to set a conditional breakpoint. You can set conditions based on the size of the array or any elements within the array. If you select multiple conditions, LabVIEW executes a breakpoint if any of those conditions are true.
- **Conditional Signed64 Probe** —Displays the 64-bit signed integer data and allows conditional breakpoints.
- **Conditional String Probe** —Displays the string data and allows conditional breakpoints.

- **Conditional Unsigned8 Probe** —Displays the 8-bit unsigned integer data and allows conditional breakpoints.
- **Conditional Unsigned16 Probe** —Displays the 16-bit unsigned integer data and allows conditional breakpoints.
- **Conditional Unsigned32 Array Probe** —Similar to the Conditional Double Array probe, this probe monitors an array of 32-bit unsigned integer numeric controls. The **Data** page displays the size of the input array. Use the **Conditions** page to set a conditional breakpoint. You can set conditions based on the size of the array or any elements within the array. If you select multiple conditions, LabVIEW executes a breakpoint if any of those conditions are true.
- **Conditional Unsigned32 Probe** —Displays the 32-bit unsigned integer data and allows conditional breakpoints.
- **Conditional Unsigned64 Array Probe** —Similar to the Conditional Double Array probe, this probe monitors an array of 64-bit unsigned integer numeric controls. The **Data** page displays the size of the input array. Use the **Conditions** page to set a conditional breakpoint. You can set conditions based on the size of the array or any elements within the array. If you select multiple conditions, LabVIEW executes a breakpoint if any of those conditions are true.
- **Conditional Unsigned64 Probe** —Displays the 64-bit unsigned integer data and allows conditional breakpoints.
- **Control Refnum Probe** —Displays the class name, the control label, the hex value of the reference, the VI name of the owner, and the path of the VI. You can use this probe to pause if the value is an invalid refnum.
- **Default Picture Probe** —Displays the picture.
- **Default Pixmap Probe** —Displays the pixmap.
- **Digital Waveform Probe** —Displays a graph of the digital data; a table with a time stamp that specifies when the probe received the data, the x value, and the time; and a table that displays the attributes of the waveform data.
- **Dynamic Data Type Probe** —Displays the attributes for each waveform and graphs the data.
- **History - Boolean** —Displays the history of the Boolean data that flowed through the wire.
- **History - CDB** —Displays the history of the complex double-precision, floating-point data that flowed through the wire.
- **History - CSG** —Displays the history of the complex single-precision, floating-point data that flowed through the wire.
- **History - CXT** —Displays the history of the complex extended-precision, floating-point data that flowed through the wire.
- **History - DBL** —Displays the history of the double-precision, floating-point numeric data that flowed through the wire.
- **History - EXT** —Displays the history of the extended-precision, floating-point numeric data that flowed through the wire.
- **History - I8** —Displays the history of the 8-bit signed integer data that flowed through the wire.

- **History - I16** —Displays the history of the 16-bit signed integer data that flowed through the wire.
- **History - I32** —Displays the history of the 32-bit signed integer data that flowed through the wire.
- **History - I64** —Displays the history of the 64-bit signed integer data that flowed through the wire.
- **History - Path** —Displays the history of the path data that flowed through the wire.
- **History - SGL** —Displays the history of the single-precision, floating-point numeric data that flowed through the wire.
- **History - String** —Displays the history of the string data that flowed through the wire.
- **History - U8** —Displays the history of the 8-bit unsigned integer data that flowed through the wire.
- **History - U16** —Displays the history of the 16-bit unsigned integer data that flowed through the wire.
- **History - U32** —Displays the history of the 32-bit unsigned integer data that flowed through the wire.
- **History - U64** —Displays the history of the 64-bit unsigned integer data that flowed through the wire.
- **Picture Probe** —Displays the picture and includes the **Height** and **Width** of the image.
- **Pixmap Probe** —Displays the picture and includes information about the width, height, and image color depth of the input picture.
- **Rendezvous Refnum Probe** —Displays the rendezvous name and size, the number of VIs or functions waiting to acquire the rendezvous, and the hex value of the reference. You can use this probe to set a breakpoint if the value is an invalid refnum.
- **Semaphore Refnum Probe** —Displays the semaphore name and size, the number of VIs or functions waiting to acquire the semaphore, and the hex value of the reference. You can use this probe to set a breakpoint if the value is an invalid refnum.
- **String Notifier Refnum Probe** —Displays the notifier name, the latest notification, the number of VIs or functions currently waiting on the notifier, and the hex value of the reference. You can use this probe to set a breakpoint if the value is an invalid refnum.
- **String Queue Refnum Probe** —Displays the queue name, the elements of the queue, the number of elements in the queue, the number of pending insert items, the number of pending remove items, and the hex value of the reference. You can use this probe to set a breakpoint if the value is an invalid refnum.
- **Toggle Execution Highlighting** —Toggles the execution highlighting at the specified point when the VI is running. This probe is not supported on RT targets.
- **Translucent Boolean Probe** —Displays the Boolean data on the wire.

- **VI Refnum (Panel Image) Probe** —Displays the VI name, the VI path, the VI front panel image, and the hex value of the reference. You can use this probe to set a breakpoint if the value is an invalid refnum.
- **VI Refnum Probe** —Displays the VI name, the VI path, and the hex value of the reference. You can use this probe to set a breakpoint if the value is an invalid refnum.

After you select a probe, that probe becomes the default probe for that data type, and LabVIEW loads that probe when you right-click the wire and select **Probe** from the shortcut menu.

You can [create a new probe based on a supplied probe](#) and add it to the **Custom Probe** shortcut menu.

Using the Generic Probe

Use the generic probe to view the data that passes through a wire. The generic probe displays the data. You cannot configure the generic probe to respond to the data. Right-click a wire and select **Custom Probe»Generic Probe** from the shortcut menu to use the generic probe.

You can debug a custom probe similar to a VI. However, a probe cannot probe its own block diagram, nor the block diagram of any of its subVIs. When debugging probes, use the generic probe.

Use [supplied](#) or [custom probes](#) to respond to the data that passes through a wire.

Using Generic Probes with LabVIEW Classes

You can use generic probes on LabVIEW class wires. The generic probe on a LabVIEW class wire shows the name of the class at run time, the private class data, and the private class data for all of the parent classes. If you use the generic probe on a locked LabVIEW class, only the name of the class at run time and any unlocked private data within the class hierarchy is visible in the probe.

Using the Probe Tool

Use the Probe tool to check intermediate values on a wire as a VI runs. Use the Probe tool if you have a complicated block diagram with a series of operations, any one of which might return incorrect data. Use the Probe tool with [execution highlighting](#), [single-stepping](#), and [breakpoints](#) to determine if and where data is incorrect. If data is available, the probe immediately updates and displays the data during execution highlighting, single-stepping, or when you pause at a breakpoint. When execution pauses at a node because of single-stepping or a breakpoint, you also can probe the wire that just executed to see the value that flowed through that wire.

You can create a *custom probe*, use *supplied probes*, or use an *indicator to view the probed data*.

Complete the following steps to use the Probe tool .

1. (Optional) If you want a probe to display the data that flowed through the wire at the last VI execution, use the *Retain Wire Values* option on the block diagram.
2. Right-click a wire and select **Probe** from the shortcut menu or select **Custom Probe** and a probe from the shortcut menu. If you have not yet selected a supplied probe or *created a custom probe*, LabVIEW searches the `labview\vi.lib\_probes\default` and the `labview\user.lib\_probes\default` directory and finds a probe that matches the data type of the wire you right-clicked. After you select a custom or default probe, it becomes the default probe for that data type. The **Debug** window appears. LabVIEW automatically lists and numbers the probe in the **Debug** window and displays the same number in a glyph on the wire you clicked. You also can use the Probe tool to click a wire or select **View»Debug Window** to display the **Debug** window.
3. (Optional) Place more probes on other wires and objects in the VI where you expect the data to change. Move the **Debug** window out of the way if necessary.
4. Run the VI. The subpanel on the right side of the **Debug** window displays data passed along the wire. Select the probe with the data you want to see in the subpanel. You can open the probe in a separate **Probe** window by right-clicking the probe in the **Debug** window and selecting **Open Window** from the shortcut menu. You also can select the probe and click the **Open Separate Probe Window** button on the **Debug** window toolbar.
5. (Optional) To locate the subpanel or window that shows the value of the probed wire from the block diagram, right-click the wire associated with the probe and select **Find Probe** from the shortcut menu. A red border highlights the subpanel or **Probe** window.
6. Compare the data that appears in the subpanel or **Probe** window to what you expected.
7. To close a probe, right-click the probe or the VI the probe belongs to in the **Debug** window and select **Remove** from the shortcut menu. You also can select the probe in the **Debug** window and click the **Remove Selected Probes** button on the **Debug** window toolbar. You can select more than one probe or select the VI the probes belong to in order to remove all probes in that VI. All probes close automatically when you close the **Debug** window or the block diagram.

You cannot use the **Debug** window to change data. The probe has no effect on the way a VI runs, aside from custom probes optionally acting as breakpoints.



NOTE

If you want to provide help to users about how to use the custom probe, do not place a **Help** button on the probe because LabVIEW only executes probes when data flows through the wire. Instead, you can provide help by *adding text* to the front panel of the probe or by *adding description and tip information* to each front panel control. To provide help for the entire probe, consider placing the controls on a tab control or in a cluster and adding description and tip information for the tab control or cluster.

The following three options are available on the shortcut menu that appears when you right-click the data in the **Probe Display** subpanel of the **Debug** window or a **Probe** window.

- **Reinitialize to Default Value** —Resets the value the **Probe** window displays to its default value.
- **Copy Data** —Copies the data for pasting to other controls in the same VI or in other VIs. This option is also available on the shortcut menu that appears when you right-click a probe listed in the **Debug** window.
- **Find Wire** —Finds the associated wire. LabVIEW brings the block diagram that contains that wire to the front and highlights the wire. This option is also available on the shortcut menu that appears when you right-click a probe listed in the **Debug** window.



RELATED INFORMATION

[Debug Window - LabVIEW Programming Reference Manual](#)

Managing Breakpoints

Use the Breakpoint tool to place a breakpoint on a VI, node, or wire and pause execution at that location. When you set a breakpoint on a wire, execution pauses after data passes through the wire and the **Pause** button appears red. Place a breakpoint on the block diagram to pause execution after all nodes on the block diagram execute. The block diagram border appears red and blinks to reflect the placement of a breakpoint.

When a VI pauses at a breakpoint, LabVIEW brings the block diagram to the front and uses a marquee to **highlight** the node or wire that contains the breakpoint. When you move the cursor over an existing breakpoint, the black area of the Breakpoint tool cursor appears white.

Setting Breakpoints

Complete the following steps to place a breakpoint on a VI, node, or wire and pause execution at that location.

1. Use the Breakpoint tool  to click the VI, node, or wire where you want to pause execution. You also can right-click the VI, node, or wire and select **Breakpoint»Set Breakpoint** from the shortcut menu.
2. Run the VI. When you reach a breakpoint during execution, the VI pauses and the **Pause** button appears red. Also, the VI background and border begin blinking. You can take the following actions:
 - **Single-step** through execution using the single-stepping buttons.
 - Check intermediate values on **probes** that you placed on wires prior to running the VI.
 - Check intermediate values on probes that you place after running the VI if you have enabled the **Retain Wire Values** option.

- Change values of front panel controls.
- Check the [Call list](#) pull-down menu to see the list of VIs that call the VI that is stopped at a breakpoint.
- Click the Pause button to continue running to the next breakpoint or until the VI finishes running.

Disabling and Enabling Breakpoints

To disable a breakpoint so the VI continues executing when LabVIEW reaches the breakpoint, right-click a block diagram object with a breakpoint and select **Breakpoint»Disable Breakpoint** from the shortcut menu. To enable a breakpoint that you previously disabled, right-click the block diagram object and select **Breakpoint»Enable Breakpoint** from the shortcut menu. You also can disable or enable breakpoints individually or all at once using the **Debug** window.

Removing Breakpoints

Use the Breakpoint tool to click an existing breakpoint to remove it. You also can right-click a breakpoint using the Positioning tool and select **Breakpoint»Clear Breakpoint** from the shortcut menu to remove it. To remove all breakpoints in the VI hierarchy, select **Edit»Remove Breakpoints from Hierarchy**. You also can remove all breakpoints in the VI hierarchy using the **Debug** window.

You must manually remove breakpoints in dynamically called VIs or VIs referenced by the Static VI Reference function.

To locate and manage all breakpoints in the VI hierarchy, select **View»Debug Window**. Double-click any item in the listbox to highlight the breakpoint on the block diagram.



NOTE

You cannot view breakpoints on a password-protected block diagram if you do not know the password, but you can enable, disable, and remove breakpoints. After you run the VI, the VI returns to the state it was in before you made any changes to the breakpoints.



RELATED INFORMATION

[Debug Window - LabVIEW Programming Reference Manual](#)

Controlling Error Notification

Use exception control to treat what is normally an error as no error or to treat a warning as an error.

Complete the following steps to cancel an existing error or upgrade a warning to an error.

1. Add the General Error Handler VI to the block diagram.

2. Right-click the **[exception action]** input and select **Create»Control**. A control icon appears.
3. Double-click the **[exception action]** control to open the front panel. Set **[exception action]** to **cancel error on match** to cancel an error or **set error on match** to treat a warning as an error.
4. Right-click the **[exception code]** input on the block diagram and select **Create»Control**. A control icon appears.
5. Double-click the **[exception code]** control to open the front panel. Set **[exception code]** to match the error code you want to cancel or to the code value of **error in** for a warning you want to treat as an error.
6. (Optional) If you have the exact string value of the error source that normally appears when the error or warning occurs, right-click the **[exception source]** input on the block diagram and select **Create»Control**. Double-click the **[exception source]** control to open the front panel. Enter the exact string value in the **[exception source]** text box.
7. Select **File»Save As** to display a file dialog box. Navigate to the directory in which you want to save the VI. Enter the name of the VI and click the **Save** button.

Defining Custom Error Codes

LabVIEW installs built-in LabVIEW error codes and error codes for related drivers, toolkits, and modules as XML-based text files. The LabVIEW built-in error codes ship with specific error code ranges. You can, however, define custom error codes using the following ranges:

- -8999 through -8000
- 5000 through 9999
- 500,000 through 599,999

You can define custom error codes in the following ways:

- To create multiple error codes for use throughout your application and distribute them in built applications or shared libraries, use the Error Code Editor dialog box.
- To create custom error codes that do not need to be widely used in your application or distributed in built applications or shared libraries, use the General Error Handler VI.
- To create a single custom error code for one particular instance in your application, use the Error Ring.



RELATED INFORMATION

[Defining Custom Error Codes to Distribute throughout Your Application](#) on page 1431

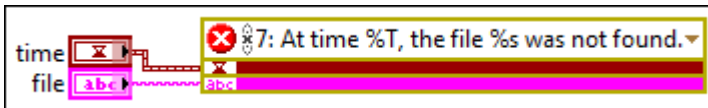
[Defining Custom Error Codes Using the General Error Handler VI](#) on page 1432

[Defining a Custom Error Code from the Error Ring](#) on page 1430

Defining a Custom Error Code from the Error Ring

Creating a custom error code is useful if you want to define a single error code or overwrite a single built-in error code. If you have multiple custom errors you want to use in the Error Ring, use the Error Code Editor dialog box. Complete the following steps to create a custom error from the Error Ring for a one-time use.

1. Add the Error Ring to the block diagram.
2. Click the drop-down arrow ▼ to display the Select Error dialog box.
3. In the **Error Code Range** drop-down listbox, select **Custom Error Code**.
4. Specify an error code in the **Error Code** field.
 - To define a custom error code, enter an error code from the following ranges of error codes specifically reserved for custom error messages: -8999 through -8000, 5000 through 9999, and 500,000 through 599,999.
 - To overwrite an existing error code message, enter the existing error code. Creating a custom error message for an already-defined error code is useful when you want to return the error code for a common error case but you want to provide specific information about this particular instance of the error. For example, you can use error code 7, "File not found." and replace that with a description about the specific file that was not found.
5. Enter a description in the **Error Description (Optional)** field. The description field accepts the same format specifiers, such as %s and %T, as the Format Into String function does. For example, for the custom error message "At time %T, the file %s was not found", the configured Error Ring adjusts to include inputs for a timestamp and a string, respectively, as displayed in the following figure.



6. Use the radio buttons to configure the Error Ring to return either an error or a warning.
7. Remove the checkmark from the **Include Call Chain** checkbox if you do not want to include the call chain information in the error cluster. If you include the call chain, the error description displays the chain of callers from the current VI to the top-level VI, which is useful when determining where in the VI hierarchy the error occurred. (Real-Time Module) Exclude the call chain to reduce jitter in real-time applications.
8. Click **OK**.

After you configure the Error Ring, you can adjust the configuration without returning to the dialog box.

- Click the two icons on the Error Ring to toggle the error/warning setting and the include/exclude call chain setting, respectively.
- Right-click the Error Ring and select **Generate Error**, **Generate Warning**, **Include Call Chain**, or **Exclude Call Chain** from the shortcut menu to affect the relevant configuration.

- Right-click the Error Ring and select **Visible Items»Error Explanation Text** from the shortcut menu to show or hide the error description.

Defining Custom Error Codes to Distribute throughout Your Application

Complete the following steps to define custom error codes using the Error Code Editor dialog box. After you create your custom error code file, you can select the errors in the Select Error dialog box and you can *distribute the file* in an application or shared library.

1. Select **Tools»Advanced»Edit Error Codes** to launch the **Error Code Editor**.
2. In the prompt that appears, click the **New** button to create a new error codes file or click the **Existing** button to browse to an existing error codes file.
3. Enter comments about the error codes file in the **Comments about this file** text box.
4. Click the error code to edit the code and select the space under the Text column to add text for the error. Click the **Add Error Code** button to add another error code to the error codes file.
5. To edit the error codes and descriptions, select the error code or text you want to change, and enter your edits. Select the **Sort Error Codes** button to sort the error codes in ascending order. If you want to delete an error code, select the error code and click the **Delete Selected Error Code** button.
6. When you are done editing the error, click the **Save** button to save the error codes file in the LabVIEW\user.lib\errors directory. You must save the error codes file in the format xxx-errors.txt, where xxx is a name that you supply.

You also can define custom error codes in the same range by creating an XML-based text file. You must name the text file xxx-errors.txt, where xxx is a name that you supply. The xxx-errors.txt file must use the following syntax exactly, including capitalization, spacing, and so on. You supply the italic text.



NOTE

You must manually create an errors folder in the LabVIEW\user.lib directory to organize your error code files.

```
<?xml version="1.0" encoding="Windows-1252"?> <nidocument>
<nicomment> This file describes custom errors for my VI. <nifamily
familyname=" Your Custom Errors " displayname=" Your Custom
Errors "><nifamily><nicomment> <nierror code=" 5000 "> Memory full.
Add more memory.<nierror> <nierror code=" 5001 "> Invalid name. Enter a
new name.<nierror> <nidocument>
```

You can add your own comment between the <nicomment><nicomment> tags. In each <nierror> tag, you must define the error code number. Define the error code message between the <nierror><nierror> tags. The <nifamily><nifamily> tag is optional. The text between these tags appears as the error category if you use the Select Error dialog box.

Changes to error code text files take effect the next time you start LabVIEW.

Distributing Custom Error Codes in Text Files

If a VI that you want to distribute as part of a stand-alone application or shared library uses custom [error codes in XML-based text files](#), you must distribute those files with the application or shared library so the error handling VIs can use them.

If you want to include custom error codes, add the error code files to the **Always Included** listbox on the Source Files page of the Application Properties dialog box or the Source Files page of the Shared Library Properties dialog box.

You also can include LabVIEW error codes. From the Advanced page of the **Application Properties** dialog box, place a checkmark in the **Copy error code files** checkbox to include LabVIEW error code files in the application. From the Advanced page of the **Shared Library Properties** dialog box, place a checkmark in the **Copy error code files** checkbox to include LabVIEW error code files in the shared library.

If you use the Application Builder to create an installer, you can place a checkmark in the **Install custom error code files** checkbox on the Advanced page of the Installer Properties dialog box to include LabVIEW error code files. The installer build includes all error code files from `labview\project\errors` and `labview\user.lib\errors` and installs them in the `Shared\LabVIEW Run-Time\x.x\errors` directory, where `x.x` is the version of LabVIEW.



NOTE

You must manually create an `errors` folder in the `labview\user.lib` directory to organize your error code files.

Defining Custom Error Codes Using the General Error Handler VI

Complete the following steps to define custom error codes using the General Error Handler VI.

1. Add the General Error Handler VI to the block diagram.
2. Right-click the **[user-defined codes]** input and select **Create»Constant** from the shortcut menu. An array appears.
3. Double-click the numeric constant and enter a number within the range of -8999 to -8000, 5000 to 9999, or 500,000 to 599,999 in the array. For example, enter 5008.
4. Right-click the **[user-defined descriptions]** input and select **Create»Constant** from the shortcut menu. Another array appears.
5. Double-click the string constant and enter a description in the user-defined description array. For example, enter `Ignore this message.`

6. Right-click the **error in** input and select **Create»Constant** from the shortcut menu. A cluster appears that contains a Boolean constant, a numeric constant, and an array.
7. Use the Operating tool to set the Boolean control to TRUE.
8. Double-click the numeric constant and enter the same number that appears in the **[user-defined codes]** constant.



NOTE

As an alternative to steps 6 through 8, you can wire a numeric constant with the specified error code to the Error Cluster From Error Code VI. Then wire the **error out** output of the Error Cluster From Error Code VI to the **error in** input of the General Error Handler VI.

1. Run the VI. An error dialog appears and displays the customized error code and description. Click the **Continue** button in this error dialog.
2. Double-click the General Error Handler subVI and view the front panel. Switch to the VI in which you defined your custom error code, and run the VI again. Click the **Continue** button again and notice that the **message** text box on the front panel of the General Error Handler VI displays the same customized error code and description as the error dialog.

You also can define custom error codes by [creating an XML-based text file](#) and adding the error codes and messages to the text file.

Finding Error Code Descriptions

LabVIEW installs built-in LabVIEW error codes and error codes for related drivers, toolkits, and modules as XML-based text files.

You can use the General Error Handler or Simple Error Handler VIs to obtain a description of these error codes.

- **General Error Handler**
- **Simple Error Handler**

You also can select **Help»Explain Error** to display an error code description in the **Explain Error** dialog box. Enter the error code number in the **Code** field or, if the error code number is in hexadecimal format, in the **Hex Code** field. Press the <Enter> key or click anywhere in the dialog box to display a description of the error code in the **Explanation** field. This description is the same description that appears in this help file. You also can [generate error cluster reports](#) to obtain descriptions of built-in error codes.

If the VIs do not return a description, the error code text files might have been moved, deleted, or corrupted. This problem usually occurs when you use [user-defined error codes](#), applications or shared libraries, or drivers released before LabVIEW 6.1.

Undefined Error Codes

Undefined errors can occur for many possible reasons. The following are some suggestions and ideas for ways to identify and handle undefined errors.

- Undefined errors might occur if a description does not exist for the error code. You can [create and define a custom error code](#) yourself. You also can contact National Instruments technical support for assistance with undefined error codes.
- An undefined error might occur if you wire a number that is not an error code to an error code input.
- Undefined error codes might occur because the error relates to a third-party object, such as the operating system or [ActiveX](#). For third-party errors, you can search the Web for a description of the undefined error code. You can search with the numerical error code or for the hexadecimal, or hex, representation of the code. For example, you can search for the error code 12341234 or for the [hex display](#) of the code, 0xBEEDF00F.



TIP

ActiveX error codes are documented by Microsoft in `winerror.h`.

Applications or Shared Libraries

If you are developing an application or shared library, you must [distribute any relevant error code text files](#) with the application or shared library.

Legacy or Third-Party Drivers

If you are using a driver that was released before LabVIEW 6.1, the driver installed a version of `error.llb` that contains all the driver error codes. The driver did not install error code text files to describe the driver error codes.

If you installed the legacy driver after installing LabVIEW 6.1, you must replace the driver's version of `error.llb` with the version of `error.llb` installed with LabVIEW 6.1 or LabVIEW 7.0.

Generating Error Cluster Reports

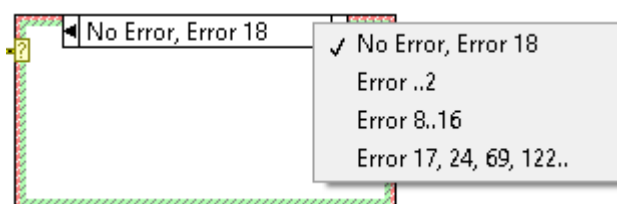
Complete the following steps to see how the error cluster reports an error.

1. Click the **status** button on either the **error in** or **error out** front panel control. The **status** button changes from a green checkmark to a red X.
2. Right-click the **status** button or the cluster border and select **Explain Error** from the shortcut menu. The **Explain Error** dialog box contains information about the error. If the VI contains [warnings](#) but no errors, the shortcut menu includes an **Explain Warning** option.

Handling Errors Using Case Structures

Complete the following steps to handle errors using **Case** structures.

1. Select an error cluster terminal.
2. Wire an error cluster to the selector terminal of a Case structure. The case selector label displays two cases—Error and No Error—and the border of the Case structure changes color—red for Error and green for No Error. The Case structure executes the Error subdiagram for all errors.
3. (Optional) To configure the case structure to execute a specific subdiagram for specific errors or list of errors, add more cases and **enter error values or ranges in the selector labels**. You can also add error values to the No Error case. The border of the No Error case is both green and red when the selector label includes any error codes.



NOTE

If you add more error cases without customizing the selector label of the Error case, the Error case changes to Default, which handles all the remaining errors that are not covered in existing error cases.

4. (Optional) Configure the display format of the error codes in the selector label by right-clicking the structure border and selecting **Radix**. If the display format is not decimal, a glyph representing the format appears on the left side of the selector label.
5. Add code to the error case(s) to define behaviors for different cases.

Handling Errors Using Conditional Terminals on Loops

You can control the iteration of a While Loop or a For Loop when an error passes to a conditional terminal. Complete the following steps to handle errors using conditional terminals.

1. Select the error cluster terminal.
2. Wire the error cluster to the conditional terminal of a While Loop or a For Loop.



NOTE

If you wire the error cluster to the conditional terminal, only the TRUE or FALSE value of the **status** parameter of the error cluster passes to the terminal. In addition, the shortcut menu items **Stop if True** and **Continue if True** change to **Stop on Error** and **Continue while Error**.

Selecting Built-In Errors from the Error Ring

Complete the following steps to select built-in errors from the Error Ring.

1. Add the Error Ring to the block diagram.
2. Click the drop-down arrow ▼ to display the Select Error dialog box.
3. In the **Error Code Range** drop-down listbox, specify the error code category you want to use. You also can [create a custom error message](#) for a one-time use from this dialog box.
4. Select the error from the list that appears.
5. Use the radio buttons to configure the Error Ring to return either an error or a warning.
6. Remove the checkmark from the **Include Call Chain** checkbox if you do not want to include the call chain information in the error cluster. If you include the call chain, the error description displays the chain of callers from the current VI to the top-level VI, which is useful when determining where in the VI hierarchy the error occurred. (Real-Time Module) Exclude the call chain to reduce jitter in real-time applications.
7. Click **OK**.

After you configure the Error Ring, you can adjust the configuration without returning to the dialog box.

- Click the two icons on the Error Ring to toggle the error/warning setting and the include/exclude call chain setting, respectively.
- Right-click the Error Ring and select **Generate Error**, **Generate Warning**, **Include Call Chain**, or **Exclude Call Chain** from the shortcut menu to affect the relevant configuration.
- Right-click the Error Ring and select **Visible Items»Error Explanation Text** from the shortcut menu to show or hide the error description.

Setting up Error I/O in a VI

Most VIs and functions provide error I/O information. This information is useful to identify the location of errors on the block diagram that occur at run time.

Complete the following steps to set up error I/O in a VI.

1. Create an error cluster control by right-clicking a subVI or function **error in** terminal and selecting **Create»Control** from the shortcut menu. You also can find the error cluster controls on the **Array, Matrix & Cluster** and **Classic Array, Matrix & Cluster** palettes.
2. With the first subVI or function, wire the **error out** output to the **error in** input of the next subVI or function.
3. Repeat step 2 for subsequent subVIs and functions until you reach the last subVI or function in the data flow.

4. On the last function or subVI, create an error indicator by right-clicking the **error out** parameter and selecting **Create»Indicator** from the shortcut menu.
5. *On the connector pane* of the VI, wire the **error in** control to a terminal and the **error out** indicator to another terminal.

**NOTE**

LabVIEW standard is to wire **error in** to the lower left terminal and **error out** to the lower right terminal on the connector pane.

Some VIs and functions return errors with numeric error codes instead of error clusters. If a VI uses the **error** or **error code** output instead of the **error out** output, wire an error handler VI to the VI.

Using Error Clusters

Use the **error in** and **error out** clusters in each VI you use or build to pass the error information through the VI. In most cases, the **source** cluster element also identifies where the error occurred. If **error in** detects an error, the VI returns the error information in **error out** and does not continue to run. By default, LabVIEW automatically handles any error when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box. You can *disable automatic error handling*.

Complete the following steps to see an example of how to use the error clusters in a VI.

1. Add the Create Report VI to the block diagram and double-click it to open it.
2. Select **Window»Show Block Diagram** to display the block diagram.
3. Use the Context Help window or the *VI Hierarchy* window to locate the Generate Report Create VI.
4. Double-click the Generate Report Create VI to open it. Notice that the front panel includes the **error in** and **error out** clusters. On the block diagram, these clusters are wired externally to a Case structure.

If an error occurs, the Case structure executes before reaching this VI. The VI does not execute the main code but instead passes the error information straight through the Case structure to the **error out** cluster. If no error occurs, the VI creates a new report.

Debugging Applications and Shared Libraries

You can debug stand-alone applications and shared libraries that you create with the Application Builder.

**NOTE**

The LabVIEW Professional Development System includes the Application Builder. If you use the LabVIEW Base Development System or Full Development System, you can purchase the Application Builder separately by visiting the National Instruments Web site.

Complete the following instructions to debug a stand-alone application or shared library.



Note When you debug applications and shared libraries, you cannot debug reentrant panels that an Open VI Reference function creates or reentrant panels that are entry points to LabVIEW-built shared libraries. You also cannot debug subVIs within Diagram Disable structures. SubVIs within Diagram Disable structures appear as question marks while you debug an application. However, because code in a Diagram Disable structure does not execute, this does not affect debugging.

1. Before you build the application or shared library, you must enable debugging in the build specification. For an application, place a checkmark in the **Enable debugging** checkbox on the Advanced page of the Application Properties dialog box. For a shared library, place a checkmark in the **Enable debugging** checkbox on the Advanced page of the Shared Library Properties dialog box. Enabling debugging retains the block diagrams of the VIs in the build so you can perform debugging.
2. Place a checkmark in the **Wait for debugger on launch** checkbox if you want the application or shared library to wait to run any VIs until you run the debugging tool.
3. Build the application or shared library.
4. Run the built application or call the shared library from outside of LabVIEW.
5. From the **Project Explorer** window, select **Operate»Debug Application or Shared Library** to display the Debug Application or Shared Library dialog box. The dialog box displays a list of applications and shared libraries with debugging enabled.
6. If the application or shared library you want to debug is running on a different computer, enter the computer name in the **Machine name or IP address** text box. Click the **Refresh** button to view the list of applications and shared libraries with debugging enabled on the remote computer.
7. Select the application or shared library you want to debug.
8. Click the **Connect** button to perform debugging. LabVIEW downloads the application or shared library and the front panel of the startup VI appears for debugging. If you enabled **Wait for debugger on launch**, you must click the **Run** button to start the application or shared library.
9. Use the block diagram of the startup VI to debug the application or shared library. You can use probes, breakpoints, and other [debugging techniques](#) to identify problems. You then can correct any problems found during debugging.



Note If you are debugging using a custom probe that uses a Call Library Function Node on a shared library that was not built with the application or shared library you want to debug, you must place the shared library in the same directory as the application or shared library you want to debug.

1. After you finish debugging, close the startup VI, which also closes the remotely controlled application or shared library. If you want to disconnect the remotely controlled application or shared library without closing the startup VI, right-click the startup VI and select **Remote Debugging»Quit Debug Session** from the shortcut menu.

The following scenarios might cause the `No debuggable applications or runtime libraries found` error to appear.

- The debuggable application or shared library was not loaded or the debuggable shared library was unloaded by the application.
- The configuration `ini` file of the shared library or application, created by the Application Builder, was not distributed with the shared library or application.



Building and Distributing Applications Home

Troubleshooting the Build Process

Complete the following to correct two common issues you might encounter while *Building and Distributing Applications* in LabVIEW.

[Conflict] appears next to one of the VIs in my project, preventing me from building an application from the project.

The Application Builder does not allow you to build an application if the project contains conflicts. A conflict in a LabVIEW Project is a potential *cross-link* that occurs when LabVIEW tries to load a VI that has the same qualified name as an item already in the project. To correct a conflict, select **Project>>Resolve Conflicts**.

The build process fails when I try to build my application, but I cannot identify the specific cause or appropriate fix.

You can use the following two methods to solve this problem. In some situations, you may need to use both methods.

1. Recompile your VIs.

1. Using the *VI Server*, the Call By Reference node, or the Start Asynchronous Call node, open one of the top-level VIs you want to use in the build as well as any VIs you want to load dynamically.
2. From an open VI, press the <Ctrl-Shift> keys while you click the **Run** button to recompile all VIs in memory. (macOS) Press the <Option-Shift> keys. (Linux) Press the <Meta-Shift> keys.
3. Select **File>Save All** to save all VIs in memory.

2. Clear the Application Builder Cache .

3. Right-click the build specification name of your application and select **Build** to rebuild the application.

Documenting in LabVIEW

Documenting VIs

You can use LabVIEW to document a finished VI and create instructions for users of VIs.

You can view documentation within LabVIEW, print it, and save it to HTML, RTF, or text files. You can import HTML and RTF files into most word processing applications, and you can use HTML and RTF files to create compiled help files.

To create effective documentation for VIs, create VI and object descriptions.



RELATED INFORMATION

[Printing Documentation for VIs, Templates, and Objects](#) on page 1448

[Generating HTML Documentation for VIs, Templates, and Objects](#) on page 1442

[Generating RTF Documentation for VIs, Templates, and Objects](#) on page 1443

[Generating Text Documentation for VIs, Templates, and Objects](#) on page 1444

[Creating Compiled Help Files](#) on page 1445

[Creating and Editing VI Descriptions](#) on page 1441

[Creating Object Descriptions and Tip Strips](#) on page 1441

Creating a Bookmark Manager

LabVIEW installs a Bookmark Manager in the `labview\resource\dialog\BookmarkManager\managers` directory. The Bookmark Manager that installs with LabVIEW is very basic and may not fulfill every requirement or use case. Therefore, you can create your own bookmark manager to better suit your application needs. Complete the following steps to create a bookmark manager.

1. Copy the `Default` folder found in the `labview\resource\dialog\BookmarkManager\managers` directory. You will use this source as the starting point for developing your bookmark manager.
2. Paste the folder in the same directory and rename `Default` to something else.
3. Locate the `Default.txt` file at the top level of the `managers` directory and rename it to match the new name of the folder.
4. Open the copy of the `*.txt` file and replace the word `Default` with the name of your new bookmark manager.
5. Edit the files in the new folder to accommodate your particular needs.

If you have more than one folder and text file under the manager's directory, LabVIEW prompts you to select the manager you want to use when you launch the Bookmark Manager window. If you place a checkmark in the **Do not show again** checkbox, LabVIEW launches the last manager you selected. If you add a new bookmark manager, LabVIEW reverts the checkbox setting and displays the Select a Bookmark Manager dialog box again.

Creating and Editing VI Descriptions

Create descriptions for VIs to describe the purpose of the VI and to give users instructions for using the VI. The VI description appears in the **Context Help** window when you move the cursor over the VI icon and in VI documentation you generate.

Complete the following steps to create and edit the description of a VI.

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Documentation from the **Category** pull-down menu.
3. Enter or edit a description of the VI in the **VI description** text box. You can *format the text* in the description to appear bold in the **Context Help** window.
4. (Optional) *Link from the VI* to an HTML file or compiled help file.
5. Click the **OK** button to save the changes.



NOTE

You cannot edit subVI descriptions from the calling VI block diagram. To edit a subVI description, *open the subVI*, select **File»VI Properties** in the subVI, and select **Documentation** from the **Category** pull-down menu.

You also can use the VI Description property to create and edit the VI description programmatically.

Creating Object Descriptions and Tip Strips

Create descriptions for objects, such as controls and indicators, to describe the purpose of the object and to give users instructions for using the object. The object description appears in the **Context Help** window when you move the cursor over the object and in VI documentation you generate. Tip strips are brief descriptions that appear when you move the cursor over an object while a VI runs.

Complete the following steps to enter a description and a tip strip for an object.

1. Add an object to the front panel or block diagram.
2. *Label the object.*
3. Right-click the object and select **Description and Tip** from the shortcut menu to display the Description and Tip dialog box.

4. Enter a description of the object in the **Description** text box. The **Description** text box uses the label you entered for the object. You do not have to enter description information for the tip strip to work. You can *add bold formatting* to the description.

**NOTE**

You can enter a description for a VI or function located on the Functions palette but you can only view the description in the **Description and Tip** dialog box. The description will not appear in the **Context Help** window.

1. Enter a tip strip for the object, up to 255 characters, in the **Tip** text box. Notice the **Tip** text box uses the label you entered for the object.



Note The **Tip** text box is available only for controls, indicators, or constants. You cannot enter tip information for VIs or functions.

1. Click the **OK** button to save the description and tip strip.

You also can use the Description and Tip Strip properties to change description and tip information programmatically.

Formatting Text in the Context Help Window

You can format text in the Context Help window to appear as bold for VI descriptions, palette descriptions, and control and indicator descriptions. Use `<B>` and `</B>` tags around any text in the description you want to format as bold.

If you want to display a carriage return in the **Context Help** window, you must separate paragraphs with two carriage returns.

**RELATED INFORMATION**

[Creating and Editing VI Descriptions](#) on page 1441

[Creating Object Descriptions and Tip Strips](#) on page 1441

Generating HTML Documentation for VIs, Templates, and Objects

You can import HTML files into most word processing applications, and you can use HTML files to *[create compiled help files](#)*.

Complete the following steps to generate HTML documentation for a VI, template, or object.

1. Open any VI, control, global variable, or template file and select **File»Print** to display the Print dialog box.
2. In the **Select VI(s)** page of the **Print** dialog box, select whether you want to generate HTML documentation for the current VI or for multiple VIs and click the **Next** button. If you select **Multiple VIs**, select the files for which you want to generate HTML documentation from the **Select VIs** page and click the **Next** button.
3. Select the components you want to include in the documentation from the **Print Contents** page and click the **Next** button. If you select **VI documentation**, select the components you want to include from the **VI Documentation** page and click the **Next** button.
4. Select **HTML file** on the **Destination** page and click the **Next** button to display the **HTML** page.
5. Set the options for the images.
6. Click the **Save** button to display a file dialog box. The dialog box prompts you to name the HTML and *image files* and select the directory where you want to save the files. The default name of any HTML documentation is the name of the VI with a `.html` extension.

You also can use the Print VI Panel or Documentation VI or the Print:VI To HTML method to generate VI documentation in HTML format programmatically.

(Statechart) You cannot generate HTML documentation for a statechart. However, you can generate XML documentation for a statechart and view the XML file in a Web browser.

Generating RTF Documentation for VIs, Templates, and Objects

You can import rich text format (RTF) files into most word processing applications, and you can use RTF files to *create compiled help files*.

Complete the following steps to generate RTF documentation for a VI, template, or object.

1. Open any VI, control, global variable, or template file and select **File»Print** to display the Print dialog box.
2. In the **Select VI(s)** page of the **Print** dialog box, select whether you want to generate RTF documentation for the current VI or for multiple VIs and click the **Next** button. If you select **Multiple VIs**, select the files for which you want to generate RTF documentation on the **Select VIs** page and click the **Next** button. Otherwise, continue to the next step.
3. On the **Print Contents** page, select the components you want to include in the documentation and click the **Next** button. If you select **VI documentation**, select the components you want to include in the documentation on the **VI Documentation** page and click the **Next** button. Otherwise, continue to the next step.
4. On the **Destination** page, select **Rich Text Format (RTF) file** and click the **Next** button.

5. On the **RTF** page, place a checkmark in the **Help compiler source** checkbox if you want to use the RTF documentation in a compiled help file. If you place a checkmark in this checkbox, LabVIEW saves images as *external BMP files* instead of embedding them in the RTF file.
6. Select a **Color depth** for the images.
7. Click the **Save** button to display a file dialog box. The dialog box prompts you to name the file(s) and select the directory where you want to save the file(s). The default name of any RTF documentation is the name of the VI with a `.rtf` extension.

You also can use the Print:VI To RTF method to generate VI documentation in RTF format programmatically.

Generating Text Documentation for VIs, Templates, and Objects

Complete the following steps to generate text documentation for a VI, template, or object.



NOTE

This method produces only the text sections of documentation, such as the description and revision history of a VI. It does not include graphics.

1. Open any VI, control, global variable, or template file and select **File»Print** to display the Print dialog box.
2. In the **Select VI(s)** page of the **Print** dialog box, select whether you want to generate text documentation for the current VI or for multiple VIs. If you selected **Multiple VIs**, click the **Next** button to display the **Select VIs** page and select the files for which you want to generate text documentation.
3. Click the **Next** button to display the **Print Contents** page.
4. Select the components you want to include in the documentation. If you select **VI documentation**, click the **Next** button to display the **VI Documentation** page and select the components you want to include in the documentation.
5. Click the **Next** button to display the **Destination** page.
6. Select **Plain text file** and click the **Next** button to display the **Plain Text** page.
7. Place a checkmark in the **Limit text per line to** checkbox to specify the number of characters that appear on each line of text.
8. Click the **Save** button to display a file dialog box. The dialog box prompts you to name the text file and select the directory where you want to save the file. The default name of any text documentation is the name of the VI with a `.txt` extension.

You also can use the Print:VI To Text method to generate VI documentation in text format programmatically.

Creating Compiled Help Files

You can use the HTML or RTF files LabVIEW generates to create your own compiled help files.

Complete the following steps to create a compiled help file.

1. Generate HTML or RTF files from the VI documentation:
 - **Generate HTML files** if you want to create a (Windows) compiled HTML Help file or (macOS) Apple Help.
 - **Generate RTF files** if you want to create a (Windows) WinHelp or (Linux) HyperHelp file.
2. Compile the source HTML or RTF documents into one file. You can use one of the following third-party utilities to compile the documents, but several utilities are available:
 - (Windows) You can use Microsoft HTML Help Workshop to compile HTML Help files and use Microsoft Help Workshop to compile WinHelp files.
 - (macOS) You can use Apple Help to view HTML files as a help system.
 - (Linux) You can use the HyperHelp compiler from Bristol Technology Incorporated to compile QuickHelp files.
3. **Link from the VIs** you documented to the compiled help file.

If you want to create an HTML-based help file for multiple platforms, you can use a compiled HTML Help file on Windows and use uncompiled HTML files on macOS and Linux. If you want to create an RTF-based help file for multiple platforms, you must compile the help file separately on each platform.

Linking VIs to Custom Documentation

Provide custom documentation from the **Context Help** window by linking a VI to a web-based or local help file. (Windows) You can use any file type except text files for custom documentation. (macOS and Linux) You can use only HTML .html and .htm and PDF files for custom documentation.



NOTE

(Linux) To use a PDF file for custom documentation, you must have one of the following PDF viewers installed:

- Acroread
- Xpdf
- KPDF
- GGV

Complete the following steps to link a VI to a custom help file:

1. Select **File»VI Properties** to display the VI Properties dialog box.
2. Select Documentation from the **Category** pull-down menu.
3. Select **Local Help File** or **Web-based Help** from the drop down menu.
4. If you select **Local Help File** in step 3, complete the following steps to link a VI to a local help file:
 1. Click the **Browse** button.
 2. Navigate to the custom help file you want to link to and click the **OK** button. The file path appears in the **Help path** text box.
 3. If **Help path** contains a path or *symbolic path* to a .htm or .html file, LabVIEW ignores the **Help tag** text box. If **Help path** contains a path to a compiled help file .chm or .hlp, use the **Help tag** text box to link the VI to a specific topic in that help file:
 - For .chm files, you can enter a .htm or .html filename or an index keyword in the **Help tag** text box. The .htm or .html filename must match the name of an individual, uncompiled HTML file in the HTML Help project. To link to a bookmark within an HTML file, add # followed by the name of the bookmark to the end of the filename. If you enter an index keyword in the **Help tag** text box, the index keyword must match a keyword in the index of the HTML Help project. If you want to link to a .chm file on Windows and link to the uncompiled HTML files on macOS and Linux, enter the HTML filename in the **Help tag** text box. (Windows) LabVIEW links to the HTML file inside the .chm file. (macOS and Linux) LabVIEW links to the HTML file, which must be in the help\html\ help.chm directory, where help.chm is the .chm filename you entered in the **Help path** text box in step 4.
 - For .hlp files, you can enter an index keyword in the **Help tag** text box. The index keyword must match a keyword in the index of the WinHelp or HyperHelp project.

You also can use the Help functions on the Help palette or the Document Path and Document Tag properties to link from VIs to custom help files programmatically.

5. If you select to link a VI to a web-based help file, enter the URL of a web-based help file in the **Help URL** text box. The file you specify links from the Detailed help link and Detailed help button in the Context Help window for the VI. You also can use the Use Web URL and Document Web URL properties to link a VI to web-based documentation programmatically.
6. Click the **OK** button to save the changes. When you move the cursor over the VI icon, a blue **Detailed help** link appears in the **Context Help** window, and the **Detailed help** button is enabled.
7. Click the blue link or the **Detailed help** button in the **Context Help** window to test the link from the VI to the HTML or compiled help file. You also can place the VI on a block diagram, right-click the VI icon, and select **Help** from the shortcut menu.



Note If you link from VIs to HTML files or compiled help files and you use the VIs to build a stand-alone application, you must include the files as source files when you build the application.

Naming Conventions for Image Files that LabVIEW Generates

When you [generate HTML](#) or [RTF documentation](#) with external graphics, LabVIEW saves the control and indicator data type terminals and other parts of the VI to image files with consistent names.

LabVIEW saves controls and indicators using the following information:

- The first letter is a **c** or an **i**, depending on if the terminal is a control or an indicator, respectively.
- If the terminal represents an array, the dimension, such as **2d**, follows the first letter.
- A short name that represents the data type follows the dimension.
- If the data type of the terminal is a LabVIEW class, then the name of the data type includes the full qualified name of the LabVIEW class.

The following table lists example image filenames for controls and indicators.

Name	Terminal
ci32.jpg	Control, 32-bit signed integer
i2di32.jpg	Indicator, 2D array of 32-bit signed integers
c3dstr.jpg	Control, 3D array of strings
cLibrary_Child_1lvclass.jpg	Control, Child 1.lvclass in Library.lvlib

When LabVIEW generates image files for other parts of the VI, such as the front panel, LabVIEW includes the full qualified name of the VI in the image filename. For example, if `Test.vi` is in a library, LabVIEW includes the library in the image filename. Also, if you select the JPEG graphic format to save a VI named `Test.vi`, LabVIEW saves the graphics as JPEG files.

The following table lists example image filenames for other VI objects.

VI Elements	JPEG Filename
Icon	Testi.jpg
Connector pane	Testc.jpg

VI Elements	JPEG Filename
Front panel	Testp.jpg
Front panel of a VI inside a library	Library_lvlib_Parent_lvclass_Testp.jpg
Block diagram	Testd.jpg
Block diagram of a VI inside a library	Library_lvlib_Parent_lvclass_Testd.jpg
VI Hierarchy window	Testh.jpg
SubVI icons	subVI_name i.jpg

Printing Documentation for VIs, Templates, and Objects

Complete the following steps to print documentation for a VI, template, or object.

1. Open any VI, control, global variable, or template file and select **File»Print** to display the Print dialog box.
2. In the **Select VI(s)** page of the **Print** dialog box, select whether you want to print documentation for the current VI or for multiple VIs. If you selected **Multiple VIs**, click the **Next** button to display the **Select VIs** page and select the files for which you want to print documentation.
3. Click the **Next** button to display the **Print Contents** page.
4. Select the components you want to include in the documentation. If you selected **VI documentation**, click the **Next** button to display the **VI Documentation** page and select the components you want to include in the documentation.
5. Click the **Next** button to display the **Destination** page.
6. Select **Printer** and click the **Next** button to display the **Page Setup** page.
7. Select the options to set the appearance of the documentation and click the **Next** button to display the **Printer** page.
8. Select from the print options available.
9. Click the **Print** button to print the documentation on the default printer on the system. If a system dialog box appears, you can set further printing options before you print the documentation.

You also can use the Print VI Panel or Documentation VI or the Print:VI To Printer method to print VI documentation to a printer programmatically.

Printing Documentation for VIs in Multiple Application Instances

A VI can appear differently if you [edit the VI in multiple application instances](#).

Complete the following steps to print documentation for a VI in multiple application instances.

1. From the current application instance, open the VI for which you want to print documentation and select **File»Print** to display the Print dialog box.
2. Use the **Print** dialog box to [print documentation](#) for the VI to a printer or to [generate HTML, RTF, or text documentation](#).
3. From a second application instance, open the same VI for which you want to print documentation and select **File»Print** to display the **Print** dialog box.
4. Use the **Print** dialog box to print documentation for the VI to a printer or to generate HTML, RTF, or text documentation.
5. Repeat the previous two steps for all application instances in which the VI for which you want to print documentation is open.

Creating HTML Documents and Embedding Front Panel Images

Complete the following steps to create an HTML document and embed static or animated images of a front panel from a VI or a stand-alone application. You also can embed images of a front panel in an [existing HTML document](#).



NOTE

When you embed the front panel of an application in an HTML document, use the front panel of the VI you embed to [enable the Web Server](#) in the application.

1. Open the front panel you want clients to view.
2. Select **Tools»Web Publishing Tool** to display the Web Publishing Tool dialog box.
3. Select a VI name in the **VI Name** field or select **Browse** from the **VI Name** pull-down menu and navigate to a VI.

Note The Web Publishing Tool creates an HTML file in the [application instance](#) from which you open the VI. If you close the VI and then reopen the VI in a different application instance, clients cannot view the VI.

4. Configure how the front panel appears in the Web browser.
 1. In the **Viewing Mode** section, select if you want to embed, take a snapshot, or monitor a front panel.
 2. Place a checkmark in the **Show border** checkbox if you want a border around the embedded panel image.
 3. Place a checkmark in the **Request control when connection is established** checkbox if you want the embedded front panel to request control immediately.
5. Click the **Next** button to display HTML output options. Enter text in the **Document title**, **Header**, and **Footer** text boxes.

6. If you want to preview the document in the default browser, click the **Start Web Server** button and then click the **Preview in Browser** button. If the **Start Web Server** button is dimmed, the Web Server is already running.
7. Click the **Next** button to display options for saving the file. Verify or edit the path to the directory where you want to save the HTML file and the HTML filename. If you want to view the document from a remote computer, save the HTML document in the [Web Server root directory](#), usually `labview\www`.
8. Click the **Save to Disk** button to save the title, text, and VI front panel image in an HTML document. The **Document URL** dialog box displays the URL for the Web page if you saved the document in the Web Server root directory.
 1. Click the **Connect** button to view the HTML document, or click the **OK** button to return to the **Web Publishing Tool** dialog box.
 2. Click the **Done** button to exit the Web Publishing Tool.
9. If you selected **Embedded** in the **Viewing Mode** section, make sure the VI whose front panel you want clients to view is in memory on the server computer.

Complete the following steps if you started the Web Server in the Web Publishing Tool and need to turn off the Web Server.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Web Server from the **Category** list of the **Options** dialog box.
3. Remove the checkmark from the **Enable Remote Panel Server** checkbox in the **Remote Panel Server** section.
4. Click the **OK** button to close the **Options** dialog box and save the changes.

If you want clients who do not have LabVIEW installed to view and control a front panel remotely, they must [install the LabVIEW Run-Time Engine](#), then [view the front panel remotely](#).

You can open the HTML file created by the Web Publishing Tool in any HTML editor and customize the HTML. Find the section of the code that embeds the front panel image to [specify attributes for the image](#).

Embedding a Front Panel Image in an Existing HTML Document

National Instruments recommends you use the Web Publishing Tool to [create an HTML document and embed static or animated images of the front panel](#) or to embed a front panel in an HTML document so a client computer can [view and control the front panel remotely](#).

However, if you already have an HTML document, complete the following steps to embed a static image of a front panel in the existing HTML document. Store the HTML document in the [Web Server root directory](#). Refer to [Web Server Syntax Examples](#) for examples of embedding VIs in HTML documents.

1. Copy the following HTML code into the existing HTML document:

```
<IMG SRC="http://web.server.addr/.snap?VI_Name[&type=png]
[&depth=24][&quality=80][&compression=-1][&refresh=60]
[&full=off]">
```

2. Replace web.server.addr with the address of the LabVIEW Web Server.
3. Replace VI_Name with the name of the VI.

A question mark separates the URL from the VI name. Parameters in brackets [] are optional.

4. (Optional) You can add the following parameters to specify attributes for the image.

Parameter	Description
VI_Name	Name of the VI front panel to view. VI_Name is the top-level VI in stand-alone applications. Encode the VI name according to standard URL naming conventions. For example, replace non-alphanumeric characters with %xx, where xx is the hexadecimal value of the character. You also can use a plus + sign to replace spaces.
type	Image type, either PNG or JPEG. If no type is specified, the default is png. If the browser does not support PNG, the default is jpeg.
depth	Depth of the color in the image. depth can be 1, 4, 8, or 24 bits. If no depth is specified, the default is the color depth of the monitor.
quality	Image quality and memory size of the JPEG front panel image. This parameter is applicable only when type is jpeg. quality can be between 0 and 100. If no quality is specified, the default is 80.
compression	Compression level used for compressing PNG images. This parameter is applicable only when type is png. compression can be between -1 and 9, where 0 indicates no compression. 1 indicates less compression, but faster speed, and 9 indicates high compression, but slower speed. Compression levels 3 through 6 perform as well as level 9 and perform fewer calculations than level 9. If no compression is specified, the default is -1, which uses best combination of speed and compression.
refresh	Maximum age in seconds of a cached image. If a cached image is older than refresh seconds, a new image is generated.
full	Specifies whether to return the image of all controls or just the part visible in the window. Set full to on to indicate all controls and off to indicate the window content. If no full is specified, the default is off.

To view the front panel image, type the IP address or computer name of the server followed by the name of the HTML file in the URL of a Web browser. For example, `http://ipaddress/filename.html`.



NOTE

To display VI front panels on the Web, the VIs must be in memory on the computer.

Embedding a VI in an Existing HTML Document

National Instruments recommends you use the Web Publishing Tool to [create an HTML document and embed VIs](#) so a client computer can view and control the front panel of the VI in a Web browser.

However, if you already have an HTML document, use the following syntax to embed a VI in the HTML document so a client computer can view and control the front panel of the VI in a browser. Store the HTML document in the [Web Server root directory](#). Refer to [Web Server Syntax Examples](#) for examples of embedding VIs in HTML documents.

The client and server computers must be running the same version of the remote panel protocol in LabVIEW. If you are using a browser to view and control a remote front panel, you must [use a version of the LabVIEW Run-Time Engine](#) compatible with the version of LabVIEW on the server computer.



NOTE
Different versions of the LabVIEW browser plug-in can run concurrently, but they communicate with different remote front panel servers. Each browser plug-in can display VIs developed in the same version of LabVIEW. The HTML OBJECT/EMBED tag determines which plug-in the browser loads. For the LabVIEW ActiveX control, the CLASSID you specify in the OBJECT tag determines which plug-in to load. The CLASSID for each version of the plug-in is different.

```
<OBJECT ID="LabVIEWControl" CLASSID="CLSID:A40B0AD4-
B50E-4E58-8A1D-8544233807B0" WIDTH= x HEIGHT= x CODEBASE="ftp://ftp.ni.com/
support/labview/runtime/windows/ 15.0 /LVRTE 15.0 min.exe"> <PARAM
name="LVFPPVNAME" value=" VI_NAME.vi "> <EMBED SRC=".LV_FrontPanelProtocol.
rpvi150 " LVFPPVNAME=" VI_NAME.vi " TYPE="application/x- labviewrpvi150 " WIDTH= x
HEIGHT= x PLUGINSOURCE="http://digital.ni.com/express.nsf/bycode/exck2m"></EMBED>
</OBJECT>
```

Refer to the following table for more information about the syntax used in different versions of LabVIEW. Changes between versions are highlighted in bold in the previous example.

LabVIEW Version	CODE BASE	EMBED SRC	EMBED TYPE
2015	15.0	rpvi150	labviewrpvi150
2014	14.0	rpvi140	labviewrpvi140
2013	13.0	rpvi130	labviewrpvi130

Note If clients access the VIs using the LabVIEW ActiveX control you need only to include the <OBJECT> tag information. However, to best support all potential clients with various configurations, you should include both the <OBJECT> and <EMBED> tags. This makes sure all clients can access the VIs remotely using a browser.

The following list describes the parameters you must configure.

Parameter	Description
CODEBASE	The location of the LabVIEW Run-Time Engine installer, including the name of the .exe file, for clients who access VIs using the LabVIEW ActiveX Control. The address provided in the previous example includes the location of the current version of the installer.
VI_NAME	The name of the VI, such as Example.vi. VI_NAME does not include path or directory names even if the VI is located in a subdirectory or inside a.llb file.
PLUGINSPAGE	The URL where clients can download the LabVIEW Run-Time Engine installer, for clients who access VIs using the Netscape plug-in. The address provided in the previous example includes the location of the current version of the installer.
(Optional) WIDTH	Width of the front panel in pixels.
(Optional) HEIGHT	Height of the front panel in pixels.

Printing in LabVIEW

Enhancing VI Printouts

Use the following techniques to improve the appearance of VI or front panel printouts:

- Use transparency and the Coloring tool to hide portions of front panel controls that you do not want visible in printouts. For example, you can use transparency to simplify the appearance of the edges of controls.
- Use objects on the Decorations palette to highlight sections of the front panel. For example, you can surround a section of controls with a box to set it off from the remainder of the user interface.
- Use graphics from other applications as pane backgrounds, items in ring controls, and parts of other front panel objects to enhance the printout.



RELATED INFORMATION

[Creating Transparent Objects](#) on page 1040

[Using Decorations on the Front Panel](#) on page 1382

[Configuring the Front Panel](#) on page 593

Generating Images of Individual Front Panel Objects

Use the Get Image method to generate an image of a single front panel object. Use the Append Control Image to Report VI to append an image of a front panel object to a report.

Complete the following steps to use the Get Image method to generate an image of a single front panel object.

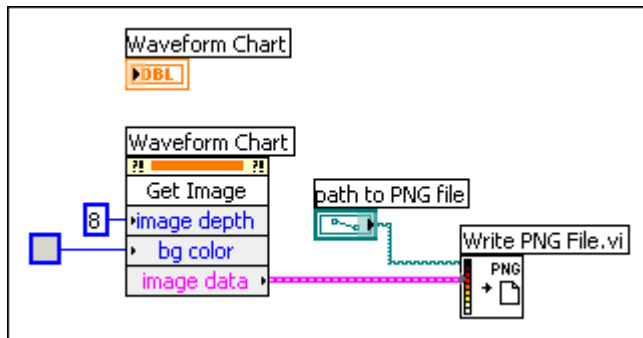
1. On the block diagram, right-click the terminal of the control or indicator for which you want to generate an image and select **Create»Invoke Node»Get Image** from the shortcut menu to create an Invoke Node.
2. Right-click the **Image Depth** input and select **Create»Constant** from the shortcut menu to create a numeric constant.
3. Enter 1, 4, 8, or 24 in the numeric constant and press the <Enter> key to set the color depth of the image.

Value	Definition
1-bit	1-bit, black and white image
4-bit	4-bit, 16-color image
8-bit	8-bit, 256-color image
24-bit	24-bit, true-color image

4. (Optional) Specify a background color for the image by wiring a color box constant or a numeric color value to the **BG Color** input.
5. Add one of the following VIs to the block diagram:
 - Add the Write JPEG File VI to save the image in the JPEG format.
 - Add the Write PNG File VI to save the image in the PNG format.
 - Add the Write BMP File VI to save the image in the BMP format.
6. Wire the **Image Data** output of the Invoke Node to the **image data** input of the graphic format VI.
7. Right-click the **path to format file** input of the graphic format VI and select **Create»Control** from the shortcut menu, where **format** is the graphic format of the image.
8. In the **path to format file** front panel control, enter a path that includes the directory where you want to save the image and a filename for the image.

Run the VI to save the image to the directory you entered.

The following block diagram saves an image of a waveform chart to the PNG format.



You also can *create a subVI that contains the Invoke Node* and wire a control reference to the subVI.

Printing the Front Panel of a VI after the VI Runs

Select **Operate»Print at Completion** to print the front panel of a VI when it finishes running.

You also can complete the following steps to print the front panel of a VI after the VI runs.

1. Select **File»VI Properties**.
2. Select **Print Options** from the **Category** pull-down menu.
3. Place a checkmark in the **Automatically Print Panel Every Time VI Completes Execution** checkbox.
4. Click the **OK** button.

Selecting these options *to print a VI* is similar to selecting **File»Print Window** when the front panel is the active window.

If you use the VI as a subVI, LabVIEW prints when that subVI finishes running and before the subVI returns to the caller.



TIP

To print a front panel that does not include the name, date, or page number, select **File»VI Properties** and select **Print Options Properties** from the pull-down menu and remove the checkmark from the **Print header** checkbox.

Printing VIs Programmatically

Use one of the following ways to print VIs programmatically rather than interactively with the dialog boxes that appear when you select **File»Print Window** and **File»Print** :

- Set a VI to automatically print its front panel every time it finishes running.
- Create a subVI to print the VI.

- Use the Report Generation VIs to print reports or to save HTML reports that contain VI documentation or data the VI returns.
- Use the Print methods in the VI class to programmatically print a VI window or to print VI documentation or save it to HTML, RTF, or text files at any time.

**NOTE**

If you print VI documentation from a stand-alone application, you can print only the front panels.

When you print VIs programmatically, LabVIEW uses the printing options for the current VI. For example, if you last printed the VI by selecting **File»Print**, LabVIEW uses any options you set in the Print dialog box when you print the VI programmatically. If you set Printing properties in the Application class or the VI class, LabVIEW changes the printing options for the current VI and uses the options you set programmatically.

**RELATED INFORMATION**

[Printing the Front Panel of a VI after the VI Runs](#) on page 1455

[Using a SubVI to Print Data from a Higher Level VI](#) on page 1457

Troubleshooting Printer Errors

Use the following techniques to find solutions to errors you might encounter when printing from LabVIEW.

- Use a different method of printing. On the Printing page of the **Options** dialog box, you can select standard, PostScript, or bitmap methods of printing, depending on the operating system you use.
- Check printer driver information. Printing problems can occur because of errors in the printer driver. Contact the printer manufacturer for the most current version of the printer driver.
- Use a different computer and/or a different printer.
- Verify that the printer accepts the page margins you set. Incorrect page margins can cause problems, such as a print job disappearing from the print queue before producing output.
- Change video settings and check the video driver. Screen resolution and color depth settings can affect print jobs. Contact the manufacturer of the graphics card for the most current version of the video driver.
- (macOS) Allocate more memory to LabVIEW.
- Generate [HTML](#) or [RTF](#) documentation and use a Web browser or word processing application to print the document.
- Use the Report Generation VIs to create, manipulate, and print reports programmatically.

If text on labels and front panel controls clips when you print VIs, the font for the printer does not match the size of the font on the monitor. Use one of the following techniques to prevent clipped control and label text in VI printouts:

- Use the PostScript printer driver for your printer if the printer supports PostScript language.
- Enlarge the labels and front panel controls so that when the text expands because of a font mismatch the text still fits in the control or label boundary.
- Find a font that is the same size on both the monitor and the printer. Many printer drivers have font substitution algorithms to assist this process.
- (Windows) Use bitmap printing to make the printout exactly match what you see on the monitor.

Refer to the KnowledgeBase for more information about troubleshooting printing errors, including how to submit a bug report to National Instruments.

Using a SubVI to Print Data from a Higher Level VI

You might want to *programmatically print a VI* only if the user clicks a button or if some condition occurs, such as a test failure. You also might want more control over the format for the printout, or you might want to print only a subset of the controls. In these cases, you can use a subVI that is *set to print at completion*.

Complete the following steps to use a subVI to print data from a higher level VI.

1. Create a subVI.
2. *Format the front panel* the way you want LabVIEW to print it.
3. Select **Operate»Print at Completion** in the subVI.
4. When you want to print, call the subVI and wire the data you want to print to the subVI.

Reports in LabVIEW

Generating Reports

Use the Report Generation VIs to print reports or to save HTML reports that contain VI documentation or data the VI returns. Use the Print VI Panel or Documentation VI to create a basic report that contains VI documentation. Use the Create Easy Text Report VI to create a basic report that contains data the VI returns. Use the other Report Generation VIs to create more complex reports.

Use the Report Express VI to generate a preformatted report that contains VI documentation, data the VI returns, and report properties, such as the author, company, and number of pages.

Formatting Data for Report Generation

You can use the Report Generation VIs to display and format information in LabVIEW. The Append Table to Report VI formats data as a 2D numeric or string array. You can use the String functions to convert data into text to append to a report. Use the Append Image to Report VI to include a picture of a graph of the data in the report.

Using Tokens in Reports

Tokens are strings that generate information automatically, which is useful in the headers and footers of reports. You can use tokens to number pages and place a time stamp on a report. You can use tokens with any parameter of a Report Generation VI or the Report Express VI that accepts a string. You can use tokens only with standard and HTML reports.

For example, if you are using the Create Easy Text Report VI, entering the token `<page>` in one of the footer strings adds a page number to the footer of every page of the report.

The following list describes the tokens available in LabVIEW.

Token	Description
<code><tab></code>	(Windows) For HTML reports, LabVIEW interprets this token as <code>
</code> .
<code><page></code>	Current page number. LabVIEW ignores this token for HTML reports.
<code><pages></code>	Total number of pages. LabVIEW ignores this token for HTML reports.
<code><pagenofm></code>	Current page number with the total number of pages in the report. LabVIEW ignores this token for HTML reports. Example: 7 of 30
<code><shortdate></code>	Current date in the form <code>xx / xx / xx</code> . Example: 10/5/98
<code><longdate></code>	Current date in the form <code>Day, Month and Date, Year</code> . The month, date, and year order defaults to the date settings of your operating system. Example: Monday, October 05, 1998
<code><time></code>	Current time in the form <code>Hour : Minute : Second AM/PM</code> . Defaults to the clock settings of your computer. Example: 1:58:22 PM
<code><newline></code>	Inserts a line break in the report.

Adding New Exported VIs Entries

Complete the following steps to add a new entry to the **Exported VIs** list.

1. Select **Tools»Options** to display the Options dialog box.

2. Select VI Server from the **Category** list.
3. Click the **Add** button in the **Exported VIs** section to add a new VI to create a new entry.
4. Enter the name of the VI or directory path in the **Exported VI** text box. You also can *use wildcards* in the VI name or directory path.
5. Click the **Allow Access** or **Deny Access** buttons to allow or deny access to the selected VIs, respectively. A green checkmark appears to the left of the item when you allow access and a red X appears when you deny access. If an entry does not have a green checkmark or a red X by its name, the syntax for the entry is incorrect. Edit the entry or click the **Remove** button to delete it.
6. Click the **OK** button to close the **Options** dialog box and save the changes.

Editing Existing Exported VIs Entries

Complete the following steps to edit the name of a VI or directory path in the **Exported VIs** list.

1. Select **Tools»Options** to display the Options dialog box.
2. Select VI Server from the **Category** list.
3. Select the VI name or directory path in the **Exported VIs list** you want to modify. The name or path appears in the text box.
4. Edit the name or path in the text box or click the **Allow access** or **Deny access** buttons. You also can *use wildcards* in the VI name or directory path.
5. Click the **OK** button to close the **Options** dialog box and save the changes.

Removing VIs from the Exported VIs List

Complete the following steps to remove a VI name or directory path from the **Exported VIs** list.

1. Select **Tools»Options** to display the Options dialog box.
2. Select VI Server from the **Category** list.
3. In the **Exported VIs** section, select the VI name or directory path in the **Exported VIs list** you want to remove.
4. Click the **Remove** button.
5. Click the **OK** button to close the **Options** dialog box and save the changes.

Adding New Machine Access List Entries

Complete the following steps to add a new entry to the **Machine access list**.

1. Select **Tools»Options** to display the Options dialog box.

2. Select VI Server from the **Category** list of the **Options** dialog box.
3. Click the **Add** button in the **Machine access** section to create a new entry.
4. Type a valid browser address in the **Machine name/address** text box. Enter a domain name, such as `mydomain.com`, or an IP address, such as `130.164.123.123`. You also can *use wildcards* in the browser address.

**NOTE**

If the VI Server runs on a computer that does not have access to a DNS server, do not use domain name entries in the **Machine access list**. Requests to resolve the domain name or IP address fail, slowing down the computer.

1. Click the **Allow Access** or **Deny Access** buttons to allow or deny access to the selected item, respectively. A green checkmark appears to the left of the item when you allow access, and a red X appears when you deny access. If an entry does not have a green checkmark or a red X by its name, the syntax for the entry is incorrect. Edit the entry or click the **Remove** button to delete it.
2. Click the **OK** button to close the **Options** dialog box and save the changes.

Editing Existing Machine Access List Entries

Complete the following steps to edit a browser address in the **Machine access list**.

1. Select **Tools»Options** to display the Options dialog box.
2. Select VI Server from the **Category** list.
3. In the **Machine access** section, select the address in the **Machine access list** you want to modify. The address appears in the text box.
4. Edit the address in the text box or click the **Allow Access** or **Deny Access** buttons. You also can *use wildcards* in the browser address.
5. Click the **OK** button to close the **Options** dialog box and save the changes.

Removing Machine Access List Entries

Complete the following steps to remove a browser address from the **Machine access list**.

1. Select **Tools»Options** to display the Options dialog box.
2. Select VI Server from the **Category** list.
3. In the **Machine access** section, select the address in the **Machine access list** you want to remove.
4. Click the **Remove** button.
5. Click the **OK** button to close the **Options** dialog box and save the changes.

Setting Machine Access for the VI Server

You can enable server support for TCP/IP by placing a checkmark in the **TCP/IP** checkbox in the **Protocols** section of the VI Server page of the Options dialog box. Use the **Machine access** section to configure which IP addresses can access VIs on a computer. Create a **Machine access list** that allows and denies access to individual IP addresses. You also can use the Server:TCP/IP Access List property to list programmatically the TCP/IP addresses of machines that may access the VI server.

When a remote LabVIEW application attempts to open a connection to the VI Server, the VI Server compares the IP address to the entries in the **Machine access list** to determine whether it should grant access. If an entry in the **Machine access list** matches the IP address, the VI Server permits or denies access based on how you [set up the entry](#). If no entry matches the IP address, the VI Server denies access. When you [add](#) new **Machine access list** entries, [edit](#) existing entries, or [remove](#) entries, use the correct [syntax, order, and wildcards](#) in the entries.

Removing Browser Access List Entries

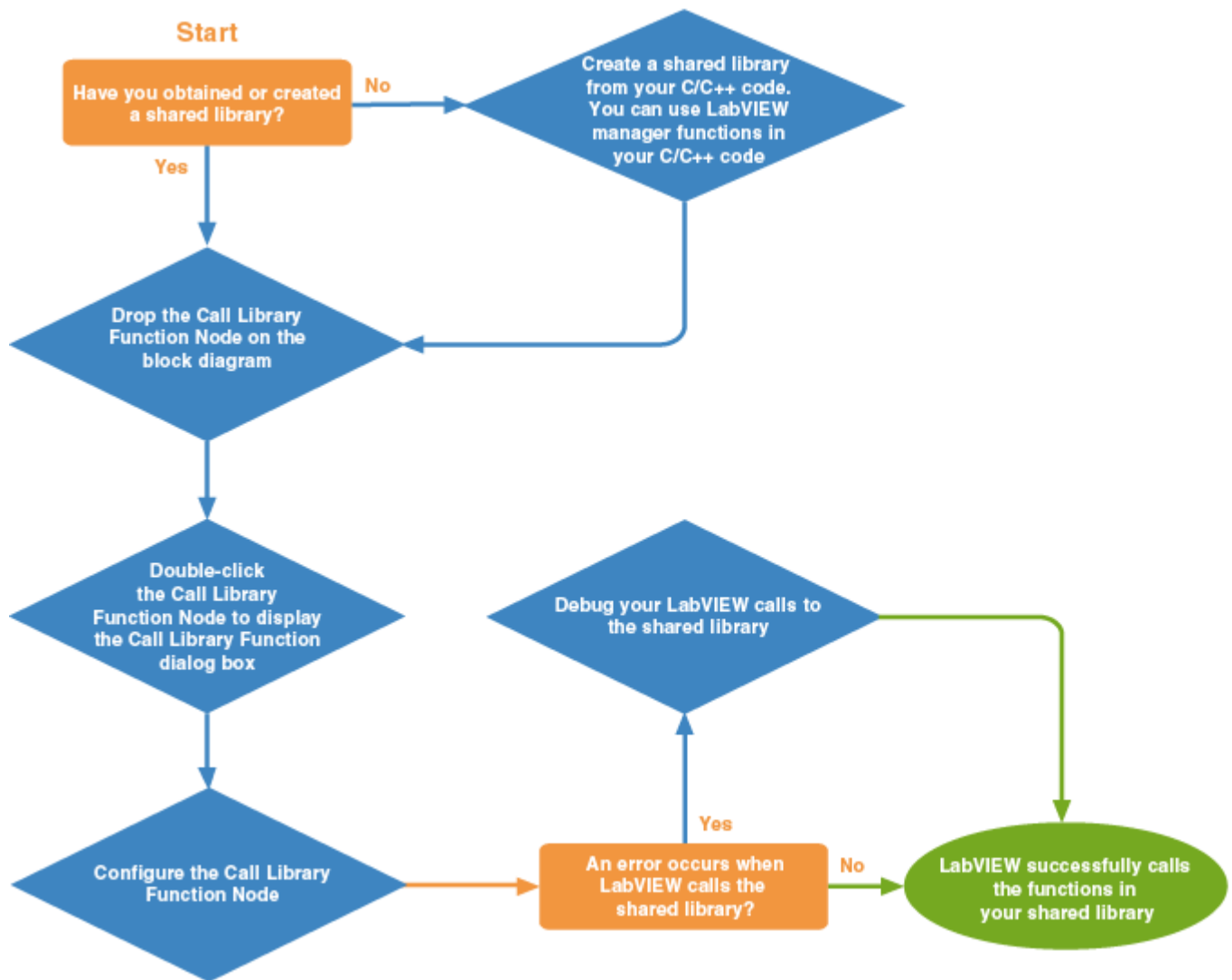
Complete the following steps to remove a browser address from the **Browser Access List**.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Web Server from the **Category** list.
3. In the **Browser Access List** in the **Browser Access** section, select the address you want to remove.
4. Click the **Remove** button.
5. Click the **OK** button to close the **Options** dialog box and save the changes.

Calling Shared Libraries in LabVIEW

Use the Call Library Function Node to call most standard shared libraries in LabVIEW. Use the Automation Open function with the Property Node and the Invoke Node to call a shared library that contains [ActiveX](#) objects.

The following illustration shows the typical workflow for calling shared libraries in LabVIEW. Click a blue diamond box for detailed information about each step.



(Windows) To view related topics, click the **Locate** button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the **Contents** tab so you can navigate the related topics.

Related Concepts

[Calling Shared Libraries](#)

[Configuring the Call Library Function Node](#)

[Using Arrays and Strings in the Call Library Function Node](#)

[Debugging Shared Libraries and Calls to Shared Libraries](#)

[Importing Shared Libraries](#)

[Using LabVIEW Manager Functions in Shared Libraries](#)

Memory Manager

Related How-To

Using the Flatten To String Function



RELATED INFORMATION

[ActiveX Container Overview](#) on page 1490

[Configuring the Call Library Function Node](#) on page 1620

[Debugging Shared Libraries and Calls to Shared Libraries](#) on page 1646

[Calling Shared Libraries](#) on page 1619

[Configuring the Call Library Function Node](#) on page 1620

[Using Arrays and Strings in the Call Library Function Node](#) on page 1651

[Debugging Shared Libraries and Calls to Shared Libraries](#) on page 1646

[Importing Shared Libraries](#) on page 1655

[Using LabVIEW Manager Functions in Shared Libraries](#) on page 1660

[Memory Manager](#) on page 1672

[Using the Flatten To String Function](#) on page 1474

[ActiveX Container Overview](#) on page 1490

[Configuring the Call Library Function Node](#) on page 1620

[Debugging Shared Libraries and Calls to Shared Libraries](#) on page 1646

[Calling Shared Libraries](#) on page 1619

[Configuring the Call Library Function Node](#) on page 1620

[Using Arrays and Strings in the Call Library Function Node](#) on page 1651

[Debugging Shared Libraries and Calls to Shared Libraries](#) on page 1646

[Importing Shared Libraries](#) on page 1655

[Using LabVIEW Manager Functions in Shared Libraries](#) on page 1660

[Memory Manager](#) on page 1672

[Using the Flatten To String Function](#) on page 1474

Building a Function Prototype

To build a shared library from a text-based programming language, you can build a function prototype in LabVIEW and then fill in all the details of the code. When you allow LabVIEW to generate this function prototype, you help ensure that the basic syntax of the code in the shared library is valid. The prototype source file you create is a .c file and contains C declarations for the parameters you want to pass.

Complete the following steps to build a prototype source file, `myshared.c`.

1. Open a new, blank VI and save the VI as `Array Average`. This function will calculate the average of an array of values.
2. Add a Call Library Function Node to the block diagram.
3. Right-click the Call Library Function Node and select **Configure** from the shortcut menu to open the Call Library Function dialog box.
4. Leave the **Library name or path** control empty.



NOTE

Use the **Library name or path** control to specify the shared library the Call Library Function Node *calls*.

1. Enter the following general specifications:
 1. Type `avg_num` in the **Function name** field.
 2. Select the **C** control from the **Calling convention** field.
2. Define the return value using the following specifications:
 1. Navigate to the **Parameters** tab of the **Call Library Function** dialog box.
 2. Change the default name in the **Name** control from `return_type` to the more descriptive name `error`.
 3. Select **Numeric** from the **Type** pull-down menu.
 4. Select **Signed 32-bit Integer** from the **Data type** pull-down menu.
3. Define the `a` parameter using the following specifications:
 1. Click the **Add a parameter** button to the right of the parameter list.
 2. Replace the default name `arg1` in the **Name** field with the name, `a`.
 3. Select **Array** from the **Type** pull-down menu.
 4. Select **4-byte Single** from the **Data type** pull-down menu.
 5. Select **Array Data Pointer** from the **Array format** pull-down menu.
4. Define the `size` parameter using the following specifications:
 1. Click the **Add a parameter** button to the right of the parameter list.
 2. Replace the default name `arg2` in the **Name** field with the name, `size`.
 3. Select **Numeric** from the **Type** pull-down menu.
 4. Select **Signed 32-bit Integer** from the **Data type** pull-down menu.
 5. Select **Value** from the **Pass** pull-down menu.
5. Define the `avg` parameter using the following specifications:
 1. Click the **Add a parameter** button to the right of the parameter list.
 2. Replace the default name `arg3` in the **Parameter** field with the name, `avg`.
 3. Select **Numeric** from the **Type** pull-down menu.
 4. Select **4-byte Single** from the **Data type** pull-down menu.

5. Select **Pointer to Value** from the **Pass** pull-down menu.
6. Check that the **Function prototype** field displays the return value and three parameters in the correct order, as follows: `int32_t avg_num(float *a, int32_t size, float *avg);`



Note The syntax you see in the **Function Prototype** field is technically correct. However, the `.c` file that the Call Library Function Node *generates* is more precise because the first parameter appears as `float a[]`.

1. Click the **OK** button to save your settings and close the dialog box.
2. Notice that the Call Library Function Node icon updates to reflect your settings.
3. Right-click the Call Library Function Node and select **Create C File** from the shortcut menu.
4. Save the file as `myshared.c`.



Note In this example, you use a `.c` source file. When you work with C++ libraries, change the extension of the source file to `.cpp`.

Preventing C++ Name Decoration

Keep the C++ compiler from introducing platform dependence in exported function names through a process called name mangling by using the C++ compiler function export directive, `extern "C"{}` , in your header file, as shown in the following example code:

```
extern "C" { int32_t MyDLLFunction(int32_t nInput, uint32_t
nOutput, void *arg1); }

int32_t MyDLLFunction(int32_t nInput, uint32_t nOutput, void
*arg1)

{

/* Insert Code Here */

}
```



Note If you disable C++ decoration of a function, the compiler cannot create polymorphic versions of the function.

After you build the function prototype, *complete the.c file*.

Completing the.c File

After you *build a function prototype* to build a shared library from a text-based programming language, complete the .c file.

The Call Library Function Node generates the following source code skeleton in myshared.c:

```
/* Call Library source File */

#include "extcode.h"

int32_t avg_num(float a[], int32_t size, float *avg);

int32_t avg_num(float a[], int32_t size, float *avg)
{
    /* Insert code here */
}
```

Replace the `/* Insert code here */` spacer with the following function code, making sure to place the code within the pair of curly braces:

```
int i;

float sum = 0;

if(a != NULL)
{
    for(i=0; i < size; i++)
        sum = sum + a[i];
}

else

return (1);

*avg = sum / size;

return (0);
```

Required Libraries

This simple example requires the `extcode.h` header file for a few simple data types. `extcode.h` provides access to a set of LabVIEW manager functions that perform simple and complex operations, ranging from low-level byte manipulation to routines for sorting data and managing memory. When you build more complex shared libraries, you must include header files for all related libraries. For example, a Windows shared library project might need to include `windows.h`.

When you want to use the LabVIEW manager functions inside a shared library, you must include the `labviewv.lib` library file in the compiled project. This file appears in the `labview\cintools` directory.

You need the LabVIEW manager functions if you intend to do any of the following tasks:

- Allocate, free, or resize arrays, strings, or other data structures that are passed into or out of the library from LabVIEW.
- Work with LabVIEW path data types.
- Work with file refnums inside the library.
- Use any of the Support Manager functions.

Refer to the LabVIEW Manager functions for more information about the manager functions.

After you complete the `.c` file, [build the library project](#) in an external IDE.

Importing Functions from a Shared Library File

Use the **Import Shared Library** wizard to create or update a LabVIEW project library of wrapper VIs for functions in a Windows `.dll` file, a macOS `.framework` file, or a Linux `.so` file. You must provide the name of a shared library file and a header `.h` file for the wizard to parse. The **Import Shared Library** wizard supports only header files that are compatible with C/C++ syntax and use the C or Standard `_stdcall` calling convention.

Complete the following steps to create or update wrapper VIs for functions in a shared library file.

1. Select **Tools»Import»Shared Library** to launch the **Import Shared Library** wizard. The **Specify Create or Update Mode** page appears.
2. If you want to create wrapper VIs for a shared library you did not import previously, select **Create VIs for a shared library**. If you want to update wrapper VIs for a

shared library that you did import previously, select **Update VIs for a shared library** and select the library from the list.



NOTE

When you select the **Update VIs for a shared library** option, the wizard overwrites the previously-generated project library file and the existing VIs within that project library file. If you choose not to re-import generated VIs within the project library file, the VIs remain unchanged in the directory. The wizard retains the most recent settings for each individual function in a particular shared library. For example, if you have a shared library that contains three functions, you might update only the second function. The next time you run the wizard on that shared library file, it retains the original settings for functions one and three and the new settings for function two.

1. Click the **Next** button to display the **Select Shared Library and Header File** page of the wizard.
2. Enter the path to the shared library file you want to import in the **Shared Library File** text box, or click the **Browse** button to navigate to and select the file.
3. If you want to create VIs for a shared library that is not available on the local computer, such as if the shared library exists on an RT target, place a checkmark in the **Shared library file is not on the local machine** checkbox and enter the filename of the shared library file in the **Shared Library File** text box. You do not have to enter the path to the shared library file.



Note If you specify only a filename and not a path, the shared library file must be in the destination directory for the wrapper VIs or it must be available through the system search path.

1. Enter the path to the header file that you want to use with the shared library in the **Header (.h) File** text box, or click the **Browse** button to navigate to and select the file. The **Import Shared Library** wizard supports only header files that are compatible with C/C++ syntax and use the C or standard (WINAPI) calling convention.
2. Click the **Next** button to display the **Configure Include Paths and Preprocessor Definitions** page, and enter any include paths or preprocessor definitions you want the wizard to use when it parses the header file.
3. Click the **Next** button to display the **Select Functions to Convert** page. A dialog box appears to notify you that LabVIEW is parsing the header file.
4. The **Select Functions to Convert** page displays all the functions found in the shared library. If the wizard cannot import a function, the function appears in the list with a black **x** glyph beside the item name. Select the function to display possible reasons for the error.



Note If you selected the **Shared library file is not on the local machine** option, this page lists only the functions declared in the header file. If you did not select the **Shared library file is not on the local machine** option, this page also lists functions referenced in the shared library file that are not declared in the header file.

1. (Optional) If the wizard cannot import one or more functions, click the **Back** button to add, remove, or update existing include paths or preprocessor definitions. If the errors are extensive, consider cancelling out of the wizard and editing the header file to be sure the file is well-formed before using the **Import Shared Library** wizard to generate wrapper VIs.
2. On the **Select Functions to Convert** page, select the functions you want to convert by placing a checkmark in the checkbox beside each function. Click the **Check All** button if you want to create wrapper VIs for all the functions in the list.
3. Click the **Next** button to display the **Configure Project Library Settings** page.
4. Enter a name for the project library file you want to create in the **Project Library Name (.lvlib)** text box. LabVIEW automatically includes all generated VIs within this project library file and uses a `.lvlib` extension for the file.
5. Enter the path to the directory in which you want to save the files the wizard generates in the **Project Library Path** text box. You also can click the **Browse** button to navigate to and select the directory.
6. (Optional) If you want to save a copy of the original shared library file in the destination directory, place a checkmark in the **Copy the shared library file to the destination directory** checkbox.



Note The wizard copies only the shared library file you specify. It does not copy any dependent shared library files. To ensure that the generated VIs are not broken, manually copy all dependent shared library files to the destination folder or add them to the system search path.

1. Click the **Next** button to display the **Select Error Handling Mode** page.
2. Select the error handling mode you want to use for the majority of the wrapper VIs from the **Error Handling Mode** pull-down menu. The **Example Block Diagram** field shows the basic block diagram the wizard generates for the wrapper VIs. If you want to select multiple error handling modes, you can re-run the wizard, select the **Update VIs for a shared library** option, and regenerate only the VIs for which you want to use a different error handling mode.
3. Click the **Next** button to display the **Configure VIs and Controls** page.
4. Review the configuration for the VIs and controls you want to create. Use the options on the **Settings** and **Name and Description** tabs to edit specific VI and control information for the majority of VIs you want to generate. An icon represents the data type for each control. An exclamation point  glyph appears beside any data types that LabVIEW cannot process. For data types that LabVIEW does not support, create a custom control and add the control to the generated library after you complete the wizard.
5. Click the **Next** button to display the **Generation Summary** page.
6. Review the **Generation Summary**. If you want to make any changes, click the **Back** button to navigate back through the wizard.
7. On the **Generation Summary** page, click the **Next** button to create wrapper VIs for the functions you selected in the specified shared library file. The **Generation Progress** page shows the progress of the wizard as it generates the wrapper VIs. When the generation completes, the **Finish** page appears.

8. Place a checkmark in the **Open the generated library** checkbox if you want to open the generated library file when the wizard closes.
9. Place a checkmark in the **View the report** checkbox if you want to view a generation report when the wizard closes. If you select this option, the report opens in the default Web browser.
10. Click the **Finish** button to close the wizard.
11. Review the project library file and the generated wrapper VIs. If you selected an error handling mode that calls a separate error handling VI, you must open the error VI and modify it.

Refer to *Example: Importing Functions from a Shared Library File* to create and update a library of VIs based on the functions included in an example shared library file, `ImportSharedLibraryExample.dll`.



RELATED INFORMATION

[Import Shared Library Wizard - LabVIEW Programming Reference Manual](#)

[Calling External APIs](#) on page 1633

[Creating Type Definitions and Strict Type Definitions](#) on page 1382

[Example: Importing Functions from a Shared Library File](#) on page 1470

Example: Importing Functions from a Shared Library File

Complete the following steps to create and update a library of VIs based on the functions in a regular expression DLL using the **Import Shared Library** wizard. Refer to *Importing Functions from a Shared Library File* to create or update wrapper VIs for functions in any shared library file.

1. Select **Tools»Import»Shared Library** to launch the **Import Shared Library** wizard. The **Specify Create or Update Mode** page appears.
2. Select **Create VIs for a shared library**, and click the **Next** button to display the **Select Shared Library and Header File** page of the wizard.
3. Click the **Browse** button next to the **Shared Library File** text box to navigate to and select `labview\examples\dll\ImportSharedLibraryExample\ImportSharedLibraryExample.dll`. After you select this file, LabVIEW designates `labview\examples\dll\ImportSharedLibraryExample\ImportSharedLibraryExample.h` as the default header file.



NOTE

`ImportSharedLibraryExample.dll` is not supported on macOS.

4. If a checkmark appears in the **Shared library file is not on the local machine** checkbox, remove it.
5. Click the **Next** button to display the **Configure Include Paths and Preprocessor Definitions** page. You do not need to add any additional include paths or preprocessor definitions for this DLL.

6. Click the **Next** button to display the **Select Functions to Convert** page. A dialog box appears to notify you that LabVIEW is parsing the specified header file.
7. Select the functions for which you want to create wrapper VIs by placing a checkmark in the checkbox next to each function. The two functions that appear contain checkmarks by default. LabVIEW creates VIs for each of the functions you select.
8. Click the **Next** button to display the **Configure Project Library Settings** page.
9. By default, `ImportSharedLibraryExample` appears in the **Project Library Name (.lvlib)** field. LabVIEW automatically includes all generated VIs within this project library file and gives it a `.lvlib` file extension. Also by default, `labview\user.lib\ImportSharedLibraryExample` appears in the **Project Library Path** field, and the **Copy the shared library file to the destination directory** checkbox contains a checkmark.
10. Click the **Next** button to display the **Select Error Handling Mode** page.
11. Select **Function Returns Error Code/Status** from the **Error Handling Mode** pull-down menu. If an error occurs, use the error status description to form the error cluster information in LabVIEW.
12. Click the **Next** button to display the **Configure VIs and Controls** page.
13. Review the configuration for the VIs and controls you want to create. Select a VI or a control on the left and then use the fields on the **Settings** and **Name and Description** tabs to review and edit specific information. Make the following changes to the default configuration settings:
 - `DWTest_Change_Info()`
 1. Change the **Default string to initialize the pointer** text box for name to **Old**.
 - `DWTest_Copy()`
 1. Change **Input/Output** for `outputEnum1` to **Output**.
 2. Change **Input/Output** for `outputEnum2` to **Output**.
 3. Change **Input/Output** for `outputRunTimeSizeArray` to **Output**.
 4. Change **Input/Output** for `outputBool` to **Output**.
 5. Change **Input/Output** for `outputString` to **Output**.

Tip You can change VI and control names to be more descriptive or to improve readability. Similarly, control names in LabVIEW can include spaces to improve readability.

14. Click the **Next** button to display the **Generation Summary** page, and review the **Generation Summary** to determine if you are ready to generate VIs. If you want to make any changes, click the **Back** button.
15. Click the **Next** button to create wrapper VIs for the functions you selected in the specified shared library. The **Generation Progress** page shows the progress of the wizard as it generates the wrapper VIs. When the generation completes, the **Finish** page appears.
16. Place checkmarks in the **Open the generated library** and **View the report** checkboxes, and click the **Finish** button to close the wizard, open the project library file, and view the report in the default browser.

17. Review the report. The report indicates that no errors or warnings occurred.
18. Review the project library file and the generated VIs within it. After you initially generate the VIs, you can make changes to the VIs individually within the block diagram or front panel of each VI. Then you can re-run the wizard to update the VIs within the library.
19. Close the `ImportSharedLibraryExample.lvlib` project library file.

Update the Generated Project Library VIs with Custom Controls

The generated library is complete and usable as is. However, you might want to make changes to several VIs within the library. To do this, you can re-launch the wizard and select the **Update VIs for a shared library** option on the **Specify Create or Update Mode** page. With this option, you can update and overwrite specific VIs that you generated previously. Complete the following steps to update a subset of the VIs within `ImportSharedLibraryExample.lvlib` with custom controls:

1. Select **Tools»Import»Shared Library** to launch the **Import Shared Library** wizard. The **Specify Create or Update Mode** page appears.
2. Select **Update VIs for a shared library** and select `ImportSharedLibraryExample.lvlib` from the list of projects.
3. Click the **Next** button to display the **Select Shared Library and Header File** page of the wizard. The **Shared Library File** text box contains the path `labview\examples\dll\ImportSharedLibraryExample\ImportSharedLibraryExample.dll`, and the **Header (.h) File** text box contains `labview\examples\dll\ImportSharedLibraryExample\ImportSharedLibraryExample.h` as the default header file.

Note `ImportSharedLibraryExample.dll` is not supported on macOS.

4. Click the **Next** button to display the **Configure Include Paths and Preprocessor Definitions** page. You do not need to add any additional include paths or preprocessor definitions for this DLL.
5. Click the **Next** button to display the **Select Functions to Convert** page. A dialog box appears to notify you that LabVIEW is parsing the specified header file.
6. Select the functions you want to convert by placing a checkmark in the checkbox next to each function. The two functions that appear contain checkmarks by default. Click the **Uncheck All** button to remove the checkmarks.
7. Place a checkmark in the checkbox next to `DWTest_Copy()` to select it as the function to update.
8. Click the **Next** button to display the **Configure Project Library Settings** page. You do not need to change any of the settings on this page.
9. Click the **Next** button to display the **Select Error Handling Mode** page. LabVIEW prompts you to confirm whether you want to overwrite the specified files. Click the **OK** button.
10. On the **Select Error Handling Mode** page, select **Function Returns Error Code/Status** from the **Error Handling Mode** pull-down menu.
11. Click the **Next** button to display the **Configure VIs and Controls** page.

12. Make the following changes to the default configuration settings:
 - DWTest_Copy()
 1. Change the **Control Type** of outputStruct to **Custom Control**.
 2. Click the **Browse** button next to the **Path to Custom Control** field to navigate to labview\examples\dll\ImportSharedLibraryExample\CustomDefinedControl.ctl as the custom control.
13. Click the **Next** button to display the **Generation Summary** page.
14. Click the **Next** button to update the VIs.
15. On the **Finish** page, select the **Open the generated library** option and click the **Finish** button to close the wizard and open the project library file.
16. Close the ImportSharedLibraryExample.lvlib project library file.



RELATED INFORMATION

[Import Shared Library Wizard - LabVIEW Programming Reference Manual](#)

[Importing Functions from a Shared Library File](#) on page 1467

[Using Error Clusters](#) on page 1437

Troubleshooting the Import Shared Library Wizard

The following are some common issues you might face while [using the Import Shared Library wizard](#). If the suggested solutions do not help you, you may need to [debug the shared library](#).

The Wizard Cannot Locate any Functions

The **Functions** list on the Select Functions to Convert page of the Import Shared Library wizard lists all the functions in the shared library file you specify. If no functions appear after the wizard parses the header file, check that the path to the shared library file is correct and that the library contains functions.

The Wizard Locates Functions that It Cannot Import

The **Import Shared Library** wizard lists all the functions in the shared library file you specify, but it can import only those functions that the wizard can parse correctly from the header file. The warnings that appear when you select a function in the **Functions** list on the **Select Functions to Convert** page can help you determine possible problems. Check that the header file contains all the necessary function declarations. Each function declaration must include the [calling convention](#), the data type of the return value, and defined parameters. To correct the header file, you can edit the file directly or use the Configure Include Paths and Preprocessor Definitions page of the wizard to add symbol definitions.

The following example code from a header file only compiles `functionA` if you define `WIN32` :

```
#ifdef WIN32 int functionA(int para); #endif
```

If you want LabVIEW to import `functionA` but the header file does not define `WIN32`, you can either edit the header file directly to define `WIN32`, or you can define `WIN32` in the **Import Shared Library** wizard. To define `WIN32` in the wizard, enter `WIN32` in the **Preprocessor Definitions** text box on the **Configure Include Paths and Preprocessor Definitions** page.

The following example code shows another function in a header file. In this situation, you did not define `NIAPI`, so LabVIEW does not recognize the symbol:

```
NIAPI int functionA(intpara);
```

If you want LabVIEW to import the function, you can add `NIAPI=_stdcall` to the **Preprocessor Definitions** text box to define `NIAPI`.

The Wizard Cannot Locate Declared Functions

The **Functions** list only shows functions that appear in the shared library file. If the header file contains declarations for functions that are not in the shared library file, the wizard does not list those functions. To see all the functions declared in the header file regardless of whether they exist in the shared library file, place a checkmark in the **Shared library is not on the local machine** checkbox on the Select Shared Library and Header File page of the wizard. The wizard then lists all the functions declared in the header file when you return to the **Select Functions to Convert** page. Be sure to remove the checkmark from this checkbox before you generate the wrapper VIs.

The Wizard Cannot Locate Nested Header Files

If the header file you specify contains nested header files, you must provide the paths to the nested files in the **Include Paths** text box on the **Configure Include Paths and Preprocessor Definitions** page of the wizard. If you do not specify include paths for nested header files, the wizard cannot parse the header file correctly.

Using the Flatten To String Function

The Flatten To String function takes LabVIEW data and converts it into a string. This string, when used in conjunction with the various communication functions or I/O functions, can be stored in a file or database or be sent to other computers.

The string created by the Flatten To String function is a LabVIEW string. If the **prepend array or string size?** input of the Flatten To String function is `TRUE`, the LabVIEW string has a 4-byte number, which is a 32-bit, signed integer data type, at the beginning of the string that specifies the length of the string. Specifying the length of the string enables a LabVIEW string to include NULL characters, such as the ASCII character zero (0). If a LabVIEW string

is passed to external code and used as a C string, NULL characters embedded in the string might cause problems because C strings are interpreted as terminating at the first NULL character.

To pass the flattened form of LabVIEW data to C code, convert the flattened LabVIEW data from a binary string format to an alphanumeric string format. For example, suppose your string consists of the following five characters:

* character 35 (#)

* character 65 (A)

* character 0 (NULL)

* character 50 (2)

* character 107 (k)

Complete the following steps to convert the preceding five characters from a binary string format to an alphanumeric string format.

1. Convert the decimal values of the five characters into hexadecimal values.

35d = 0x23

65d = 0x41

0d = 0x00

50d = 0x32

107d = 0x6B

2. Write down the actual alphanumeric characters for the hexadecimal values and include only a single NULL value at the end.

* character 50 (2)

* character 51 (3)

* character 52 (4)

* character 49 (1)

* character 48 (0)

* character 48 (0)

* character 51 (3)

* character 50 (2)

- * character 54 (6)
- * character 66 (B)
- * character 0 (NULL)

Converting from a binary string format to an alphanumeric string format doubles the size of the string. However, converting to an alphanumeric format preserves the information in the string when you use the string in an environment where you have to replace LabVIEW strings with C strings.

Creating Dotted Properties and Methods

Dotted properties and methods allow properties that return references to concatenate with other properties or methods. Dotted properties and methods are a shorthand version of cascading multiple Property Nodes that return references together where the reference returned from the property is wired to the reference input on another property or method. The intermediate references returned in dotted properties are automatically closed by the dotted property or dotted method. If the final property returns a reference, that reference is not closed automatically.

You can use the Class Browser window to create dotted properties and methods. Dotted properties return a reference and have a blue glyph next to the property name in the **Properties and Methods** list.

Complete the following steps to create a dotted property or method.

1. Double-click on the dotted property to update the **Properties and Methods** list to the appropriate class. The dotted property appears in the **Selected property or method** box as an HTML link. If you select another property or method, it concatenates to the dotted property and shows up in the box.



NOTE

A.NET or ActiveX property that returns a reference to a class outside the current type library cannot be made into a dotted property. When you view the property in the **Class Browser** window, a blue glyph does not appear next to the property name in the **Properties and Methods** list.

1. (Optional) You can click an HTML link in the box to navigate back to where you selected the dotted property and update the **Class Browser** window to the selected item.

Creating Properties and Methods

Properties are attributes of a VI, object, or application. Methods perform actions on a VI, object, or application. Use a Property Node to get or set a property of a specific VI, object, or application. Use an Invoke Node to perform an action, or method, on a specific VI, object, or application.

You can create multiple properties and methods for the same VI, object, or application.

Complete the following steps to create a property or method using a Property Node or Invoke Node, or use one of the [alternative ways](#) to create a property or method.

1. Determine for which object you want to create a property or method. Obtain a reference to that object in one of the following ways:
 - VI—If you want to create a property or method for the current VI, you do not have to obtain a reference. If you want to create a property or method for another VI, add the Open VI Reference function to the block diagram. Open VI Reference
 - Front panel object—Right click the block diagram terminal of the object and select **Create»Reference** from the shortcut menu to create a reference to the object.
 - Application—If you want to create a property or method for the current application instance, you do not have to obtain a reference. If you want to create a property or method for another application instance, add the Open Application Reference function to the block diagram. Open Application Reference
2. Determine whether you want to create a property or a method.
 - To create a property, add a Property Node to the block diagram. Property Node
 - To create a method, add an Invoke Node to the block diagram. Invoke Node
3. Wire the reference to the object to the **reference** input of the node. The node automatically changes to show the class of the object that you wired to it. Different classes of objects have different properties and methods available to them.



NOTE

If you want to create a property or method for the current application or VI, you do not have to wire data to the **reference** input of the node because LabVIEW uses the current application or VI by default. However, you must manually select either the Application or VI class by right-clicking the node and selecting **Select Class** from the shortcut menu.

4. To select the specific property or method to use, click the white area of the node and select the property or method from the hierarchical menu that appears.
5. (Optional) Property Nodes allow you to get or set multiple properties with the same node. To add more properties to the same Property Node, right-click the white area of the node and select **Add Element** from the shortcut menu or use the Positioning tool to [resize the node](#).
6. (Optional) To change a property from read to write, right-click the property and select **Change to Read** or **Change to Write** from the shortcut menu.
7. If the Property Node returns a reference or an array of references to an object, use the Close Reference function to close the reference.



Note If you add a property to a While Loop or For Loop and the property returns a reference to an object, you must close that reference for each iteration of the loop. You can move the property outside the loop and close the reference after all iterations of the loop execute.

Alternative Ways to Create Properties and Methods

- You can use the Class Browser window to browse for and *create properties and methods*.
- You can right-click a front panel object or its block diagram terminal, select **Create»Property Node** or **Create»Invoke Node** from the shortcut menu, and select a property or method from the list to create a property or method for the object. You do not have to wire a reference to the resulting Property or Invoke Node because the node is implicitly linked to the object. If the object has a label, the property has the same label. You can change the label after you create the node. If you select a method, the associated parameters appear in the node.
- (Windows) To create a .NET property or method, wire a *.NET container* or a *.NET reference* to the reference terminal of the node.
- To create an ActiveX property or method, wire an *ActiveX container* or *automation refnum* to the **reference** terminal of the node.

Creating Properties and Methods with the Class Browser Window

You can use the Class Browser window to select an object library and create a new property or method.

Complete the following steps to create a new property or method using the **Class Browser** window.

1. Select **View»Class Browser** to display the **Class Browser** window.
2. From the **Object library** pull-down menu, select a library.



NOTE

(Windows) To select a .NET Framework or ActiveX library, you must select **.NET»Select Assemblies** or **ActiveX»Select Type Libraries** to display the Select Assemblies or Select Type Libraries dialog box, respectively. Place a checkmark in the checkbox for the assembly or library you want and click the **OK** button to return to the **Class Browser** window.

1. Select a class from the **Class** pull-down menu. Use the following buttons to navigate the classes.
 -  Click the **Select View** button to toggle between an alphabetical view and a hierarchical view of the items in the **Class** pull-down menu and the **Properties and Methods** list.
 -  Click the **Search** button to launch the Class Browser Search dialog box.
2. From the **Properties and Methods** list in the **Class Browser** window, select a property or method. Dottable properties have a blue glyph next to the property name in this list. To *create a dotted property*, double-click on a dottable property.

The property or method you select appears in the **Selected property or method** box.



Note *LabVIEW classes* do not support Invoke Nodes. LabVIEW class methods are member VIs. If you use the **Create** button or if you drag a method directly to the block diagram, a subVI appears. Methods in LabVIEW classes appear as subVIs rather than Invoke Nodes. You cannot define LabVIEW class properties in this version of LabVIEW.

1. Click the **Create** button or the **Create Write** button to attach a node with the selected property or method to your mouse cursor and add the node to the block diagram. The **Create** button creates a property for reading or a method. This button dims when you select a write-only property. To create a property for writing, click the **Create Write** button. The **Create Write** button dims when you select a method or read-only property. You also can drag a property or method from the **Properties and Methods** list directly to the block diagram.
2. Repeat steps 2 through 5 for any other properties and methods you want to create and add to the block diagram.

Registering and Handling .NET and ActiveX Events

When handling a .NET or ActiveX event, you must register for the event (see steps 1 through 5) and then create a callback VI to handle that event (see steps 6 and 7).



NOTE

National Instruments recommends that you unregister for events when you no longer need to handle them using the Unregister For Events function. If you do not unregister for events, LabVIEW continues to generate and queue the events as long as the VI runs, even if no Event structure is waiting to handle them, which consumes memory and can hang the VI if you enable front panel locking for the events.

Complete the following steps to register and handle .NET and ActiveX events in LabVIEW.

1. Create a new control in a *.NET container* or an *ActiveX container*. You also can use the Constructor Node to *create a .NET object* or use the Automation Refnum to *call an ActiveX object* from LabVIEW.



NOTE

LabVIEW only supports registering and handling .NET Core events using the Data Constructor Node.

2. Add the Register Event Callback function to the block diagram.
3. Wire the reference or object to the **Event** input of the Register Event Callback function.
4. Click the down arrow next to the **Event** input on the Register Event Callback function and select an event from the shortcut menu.

5. (Optional) Wire the data that you want to pass to the callback VI to the **User Parameter** input. You can wire any LabVIEW data type, including a cluster, to this parameter.
6. Right-click the **VI Ref** input and select **Create Callback VI** from the shortcut menu. LabVIEW creates a VI based on the event you select and the data you wire to the **User Parameter**. LabVIEW wires the callback VI to the **VI Ref** input.



Note You can use an existing VI as a callback VI as long as the connector pane of the VI you intend to use matches the connector pane of the event data. The callback VI must be *reentrant*, and the reference to the callback VI must be strictly typed.

1. Open the callback VI and edit the block diagram to handle the event. The callback VI contains the following elements:
 - **Event Common Data** contains the following elements:
 - **Event Source** is a numeric control that specifies the source of the event, such as LabVIEW, .NET, or ActiveX. A value of 1 indicates an ActiveX event. A value of 2 indicates a .NET event.
 - **Event Type** specifies which event occurred. This is an enumerated type for user interface events and a 32-bit unsigned integer type for .NET, ActiveX, and other event sources. For .NET and ActiveX events, the event type represents the method code, or ID, for the event registered.
 - **Time Stamp** is the time stamp in milliseconds that specifies when the event was generated.
 - **Control Ref** is a reference to the .NET or ActiveX object or the Automation Refnum on which the event occurred.
 - **Event Data** is a cluster of the parameters specific to the event the callback VI handles. LabVIEW determines the appropriate **Event Data** when you select an event from the Register Event Callback function. If an event does not have any data associated with it, LabVIEW does not create this control in the callback VI.
 - **Event Data Out** is a cluster of the modifiable parameters specific to the event the callback VI handles. This element is available only if the event has output parameters.
 - (Optional) **User Parameter** is data that you want to pass to the callback VI when the .NET or ActiveX object generates the event.

Note LabVIEW does not wire the **Event Data** input to the **Event Data Out** output automatically. Therefore, if you want to pass any part of the **Event Data** input through the callback VI, you must wire the **Event Data** input to the **Event Data Out** output manually.

Creating .NET Controls in a .NET Container



NOTE

The following content applies to .NET Framework only. LabVIEW has been gradually adding support for .NET Core since LabVIEW 2024 Q3. Refer to *LabVIEW New Features and Changes* for specific .NET Core support added in different releases.

Complete the following steps to create a .NET control in a .NET container.



NOTE

Creating and communicating with .NET objects in LabVIEW requires the .NET CLR 4.0 that installs with LabVIEW. You must use a .NET 2.0 configuration file if you want to load .NET 2.0 mixed-mode assemblies. Refer to the requirements for using .NET with LabVIEW for more information about .NET restrictions in LabVIEW. National Instruments strongly recommends that you use .NET objects only in LabVIEW projects.

1. Add a .NET container to the front panel window.
2. Right-click the front panel window .NET container and select **Insert .NET Control** from the shortcut menu to display the Select .NET Control dialog box.
3. Select an assembly from the **Assembly** pull-down menu. You also can click the **Browse** button to access private assemblies. .NET assemblies are .dll file types. After you select a private assembly, the assembly appears in the **Assembly** pull-down menu the next time you launch this dialog box.



NOTE

If you move a VI that uses a private assembly to a different folder or machine, you either must move the associated private .NET assembly files to a subdirectory that LabVIEW can search, or you must save the files in the Global Assembly Cache (GAC). If you build a VI that uses a private assembly into a shared library or stand-alone application, LabVIEW copies the associated private .NET assembly files to the data subdirectory in the same directory as the library or application.

1. Select a .NET control from the **Controls** list and click the **OK** button. The .NET control appears in the front panel window .NET container.
2. On the block diagram, wire the .NET control to a Property Node, Invoke Node, or Register Event Callback node to set properties, invoke methods, or handle events for the control.

Do not use the Close Reference function to close the reference to a .NET control. By leaving the reference open, you ensure that the control displays correctly until you close the VI. LabVIEW automatically closes the reference when you close the VI. This recommendation differs from how you manage a reference to a block diagram .NET object.

**TIP**

If you encounter problems loading a particular assembly or accessing objects in an assembly, debug the assembly call or use the .NET Assemblies in Memory dialog box to verify that the correct version of the assembly is in memory.

**NOTE**

.NET controls automatically use the user interface execution system.

You also can add .NET controls to the Controls palette for later use.

**RELATED INFORMATION**

[Requirements for Using .NET Framework Assemblies in LabVIEW](#) on page 791

[Loading .NET Framework 2.0, 3.0, and 3.5 Assemblies in LabVIEW](#) on page 1484

[Requirements for Using .NET Framework Assemblies in LabVIEW](#) on page 791

[Managing a Project in LabVIEW](#) on page 365

[Using .NET with LabVIEW](#) on page 789

[Creating .NET Objects on the Block Diagram](#) on page 1482

[Debugging Calls to .NET Framework Assemblies](#) on page 1488

[Multitasking in LabVIEW](#) on page 752

[Creating .NET Subpalettes](#) on page 839

Creating .NET Objects on the Block Diagram

You can use LabVIEW to access objects associated with third party .NET servers. Complete the following steps to create and configure a .NET object on the block diagram.

Step 1: Configure the Constructor Node

Add the Constructor Node to the block diagram to launch either the **Select .NET Core Constructor** or the **Select .NET Framework Constructor** dialog box. Then select an assembly, objects, and constructors for a .NET object.

Refer to the following list for important details about configuring each component of the **Select .NET Framework Constructor** dialog box:

- **Assembly** —This component lists all public assemblies in the Global Assembly Cache. To access private assemblies, click the **Browse** button. When you search private assemblies, note that .NET assemblies are .dll file types. After you select a

private assembly, the assembly appears in the **Assembly** pull-down menu the next time you launch this dialog box.



NOTE

If you move a VI that uses a private assembly to a different folder or machine, you either must move the associated private .NET assembly files to a subdirectory that LabVIEW can search, or you must save the files in the Global Assembly Cache (GAC). If you build a VI that uses a private assembly into a shared library or stand-alone application, LabVIEW copies the associated private .NET assembly files to the data subdirectory in the same directory as the library or application.

- **Objects** —The assembly you choose in the previous step determines the available objects classes. After you select a class of objects, the constructors for that class appear in the **Constructors** section of the **Select .NET Framework Constructor** dialog box.
- **Constructors** —After you select a constructor, click the **OK** button. LabVIEW then displays the name of the class you selected on the Constructor Node.

Step 2: Configure Properties, Methods, or Events for the .NET Object

1. The .NET Constructor Node creates a reference to an instance of a .NET object. Once you have that reference, you can act on the .NET object. To act on the object, wire the **new reference** output of the Constructor Node to a Property Node or Invoke Node and set properties or set methods for the object. With .NET Framework, you can Register Event Callback and handle events for the object.
2. Use the Close Reference function to close the reference to the .NET object and relinquish the memory resource LabVIEW allocated to that object.

Tips and Troubleshooting Information for .NET Framework

- .NET CLR 4.0 installs with LabVIEW. If you want to load .NET 2.0 mixed-mode assemblies, you must use a .NET 2.0 configuration file.
- If you encounter problems loading a particular assembly or accessing objects in an assembly, debug the assembly call or use the .NET Assemblies in Memory dialog box to verify that the correct version of the assembly is in memory.

Related Information

[Loading .NET 2.0, 3.0, and 3.5 Assemblies in LabVIEW](#)

[Requirements for Using .NET Assemblies in LabVIEW](#)

[Using .NET with LabVIEW](#)

[Creating Properties and Methods](#)

[Registering Handling .NET and ActiveX Events](#)

Using Static .NET Properties and Methods

LabVIEW supports static .NET properties and methods. You can execute static properties and methods on classes without wiring a .NET object reference to the Property Node or Invoke Node.



NOTE

Creating and communicating with .NET Framework objects in LabVIEW requires the [.NET CLR 4.0](#) that installs with LabVIEW. You must use a [.NET 2.0 configuration file](#) if you want to load .NET 2.0 mixed-mode assemblies. Refer to the [requirements for using .NET with LabVIEW](#) for more information about .NET restrictions in LabVIEW. National Instruments strongly recommends that you use .NET objects only in [LabVIEW projects](#).

Complete the following steps to use static properties and methods with a .NET object.

1. Add one of the following nodes to the block diagram:
 - Property Node
 - Invoke Node
2. Right-click the node and select either **Select Class»NET Core»Browse** or **Select Class»NET Framework»Browse** from the shortcut menu to display the Select Object From Assembly dialog box.
3. Select a class from the **Assembly** pull-down menu.
4. Select an object from the **Objects** listbox.
5. Click the **OK** button.
6. Right-click the Property Node or Invoke Node and select a static property or method from the shortcut menu. LabVIEW indicates static properties and methods with a [S] at the beginning of the property or method name.



NOTE

If you resize the Property Node to include more properties, and some of those properties are not static properties, you must wire a .NET server reference to the Property Node by using a Constructor Node to [create a .NET object](#).

Loading .NET Framework 2.0, 3.0, and 3.5 Assemblies in LabVIEW

The .NET Framework 4.0 allows you to load pure managed assemblies built in any version of the .NET Framework and mixed-mode assemblies built in .NET 4.0. Although LabVIEW

loads the .NET Common Language Runtime (CLR) 4.0 by default, you can force LabVIEW to use the CLR 4.0 to load .NET mixed-mode assemblies that target the CLR 2.0.



NOTE

The .NET Frameworks 2.0, 3.0, and 3.5 each use the CLR 2.0. However, you cannot configure LabVIEW to use the CLR 2.0. Assemblies that target older versions of the CLR run in the CLR 4.0 and might behave unexpectedly.

To enable the following objects to load .NET mixed-mode assemblies that target the CLR 2.0, you must use the configuration file described below:

- LabVIEW 2013
- An executable built in LabVIEW 2013
- An executable built outside of LabVIEW that calls into an interop assembly or DLL built in LabVIEW

Complete the following steps to configure LabVIEW to load .NET mixed-mode assemblies that target the CLR 2.0:

1. Open a text editor.
2. Copy the following text into the text editor:

```
<?xml version="1.0" encoding="utf-8" ?> <configuration> <startup useLegacyV2RuntimeActivationPolicy="true"> <supportedRuntime version="v4.0.30319"> </startup> </configuration>
```
3. Save this new configuration file in the same directory as LabVIEW.exe. Name the file LabVIEW.exe.config.
4. Relaunch LabVIEW.

Now LabVIEW can load .NET mixed-mode assemblies that target the CLR 2.0.



NOTE

If you build a stand-alone application from a LabVIEW VI or project that uses .NET 2.0 mixed-mode assemblies, the user of your application must install the .NET Framework 2.0 on his system and place a configuration file named `application name.exe.config` in the same directory as the application. The text of this configuration file must be the same as the text of the configuration file in step 2 of the previous procedure. To prevent the user from having to create his own configuration file, consider *building an installer* that includes the configuration file.

Loading Changes to a .NET Framework Assembly in Memory

**NOTE**

The following content applies to .NET Framework only. LabVIEW has been gradually adding support for .NET Core since LabVIEW 2024 Q3. Refer to *LabVIEW New Features and Changes* for specific .NET Core support added in different releases.

When you make changes to an assembly while that assembly is in memory, you must force LabVIEW to reload the assembly in order to access the changes. Complete the following steps to force LabVIEW to load changes to an assembly.

**NOTE**

An assembly remains in memory even if you delete all nodes that reference the assembly from the VI.

1. Identify the name, version number, and path on disk of the assembly that you changed.
2. Use the .NET Assemblies in Memory dialog box to identify all LabVIEW projects that have a version of the modified assembly in memory. For each project that uses the assembly, verify that the version number and path of the assembly in memory match the version number and path of the assembly that you changed. If this information does not match, LabVIEW is using a different copy of the assembly than the one you updated. You can [debug the assembly call](#) to make LabVIEW load the correct assembly.
3. Close and reopen all projects that reference the modified assembly.

**NOTE**

If a VI in the main application instance references the modified assembly, you must relaunch LabVIEW in order to access the changes. National Instruments strongly recommends that you use .NET objects only in [LabVIEW projects](#).

Specifying Which Version of a .NET Assembly to Use

**NOTE**

The following content applies to .NET Framework only. LabVIEW has been gradually adding support for .NET Core since LabVIEW 2024 Q3. Refer to *LabVIEW New Features and Changes* for specific .NET Core support added in different releases.

If you have multiple versions of the same assembly on your machine, you must specify which version of the assembly to use in the following situations:

- You originally configure a .NET Constructor Node to load one version of the assembly but later want the node to access a different version of the assembly. LabVIEW permanently associates the original version of the assembly with your application. Therefore, you cannot access properties, methods, and behaviors of the alternate version until you create the configuration file.
- You distribute your application to another machine that may have multiple versions of the assembly.** Specifying which version of the assembly to use ensures that the distributed application detects and uses the correct version. You must distribute the configuration file with the application.

To guarantee that a LabVIEW project or built application refers to a specific version of a .NET assembly, close the project and create a *.NET configuration file* similar to the following example, replacing the italicized code according to the table later in this topic:

```
<configuration> <runtime> <assemblyBinding
  xmlns="urn:schemas-microsoft-com:asm.v1"> <
probing
  privatePath="Assembly\ver2"/ >
<dependentAssembly>
  <assemblyIdentity name=" AssemblyName "
publicKeyToken="
  xee7d93a67000dr7 " culture="Neutral" />
<bindingRedirect
  oldVersion=" oldVersionNumber " newVersion="
  newVersionNumber " /> </dependentAssembly>
</assemblyBinding> </runtime> </configuration>
```

Code to Replace	Guidelines
probing privatePath="Assembly\ver2"	Specifies the directory within the project from which LabVIEW must load the assembly. If you can guarantee that the assembly is in the Global Assembly Cache (GAC), omit this line from the configuration file. Otherwise, this line forces LabVIEW to search in this directory before <i>anywhere else</i> .
AssemblyName	Represents the name of the assembly with multiple versions.
xee7d93a67000dr7	Uniquely identifies the assembly. Generate this key using the Strong Name tool, available on the Microsoft Developer Network (MSDN).
oldVersionNumber	Refers to the version of the assembly that LabVIEW is currently using in the form n.n.n.n. To prevent LabVIEW from using multiple different versions of the assembly, specify a range of versions here using the form n.n.n.n - n.n.n.n.
newVersionNumber	Refers to the version of the assembly that you want LabVIEW to use instead of oldVersionNumber.

For examples of using .NET configuration files with LabVIEW, refer to the KnowledgeBase at ni.com.

Debugging Calls to .NET Framework Assemblies



NOTE

The following content applies to .NET Framework only.

.NET objects can exhibit unexpected behavior when LabVIEW has a different version of the assembly in memory than the version you want to use. To diagnose many of these .NET assembly conflicts, use the .NET Assemblies in Memory dialog box. Because this dialog box displays identifying information for the assemblies in memory, it allows you to verify if LabVIEW loaded the correct version of an assembly.

Complete the following steps to debug a call to a .NET Framework assembly.

1. Determine the name, version number, and location on disk of the assembly you want LabVIEW to load. Ensure that the assembly is located within the [search path that LabVIEW uses to locate assemblies](#).
2. Select **View» .NET Assemblies in Memory** to display the .NET Assemblies in Memory dialog box.
3. In the dialog box, locate the name of the assembly that you are trying to load.
4. Identify discrepancies between the assembly information in the dialog box and the identifying information of the desired assembly from step 1. If there are no discrepancies but LabVIEW does not reflect your most recent changes to the assembly, you might need to [force LabVIEW to load the updated assembly](#).
5. Choose one of the following procedures for replacing the assembly in memory with the desired assembly.
 - Move the loaded assembly out of the [assembly search path](#). This action forces LabVIEW to search for a different version of the same assembly.
 - Create a project configuration file to [force LabVIEW to load the desired version of the assembly](#).
6. Close and reopen the projects that call the assembly. This action forces LabVIEW to unload the incorrect version of the assembly from memory and then to load the correct version.



NOTE

If the VI that loaded the incorrect assembly is not part of a project, you must restart LabVIEW to clear the incorrect assembly from memory. NI strongly recommends that you use .NET objects only in [LabVIEW projects](#).

ActiveX

Accessing an ActiveX-Enabled Application

To access an ActiveX-enabled application in LabVIEW, complete the following steps:

1. Add an automation refnum control to the front panel of a VI.
2. Select an ActiveX class for the automation refnum control to specify the ActiveX object you want to access.
3. Use the Automation Open function on the block diagram to open the reference to the ActiveX object. When you open the reference, you enable LabVIEW to communicate with the referenced ActiveX object.
4. Pass the reference to a Property Node or an Invoke Node to access properties or invoke methods on the object. When you click one of these nodes, LabVIEW displays all of the properties or methods available for the object. The object implements the Component Object Model (COM) interface, which determines the available properties and methods.



NOTE

If LabVIEW does not display the properties or methods that you expect for a particular ActiveX object, the object might implement multiple COM interfaces. LabVIEW can display the properties or methods from only one COM interface implemented by an object at a time. You can force LabVIEW to display the properties and methods from a different COM interface that the object implements by using the Variant To Data function to cast the object to the other COM interface.

5. Use the Close Reference function to remove the object from memory.

You can use the previous steps to access common ActiveX enabled applications, such as Microsoft Office applications. For example, Microsoft Excel is an ActiveX-enabled application. You can open a reference to Excel and call properties and methods that open Excel so it appears on the user screen, create a workbook, create a spreadsheet, and write a table from LabVIEW to the Excel spreadsheet.

Refer to the Excel - Write Table VI in the `labview\examples\Connectivity\Excel` directory for an example of using LabVIEW as an Excel client.



NOTE

Applications that include ActiveX custom interfaces appear with a  icon. Click the icon to select an object for the custom interface.

Accessing ActiveX Properties

You can access the properties for an ActiveX object in the following ways:

- ActiveX Control Property Browser —Available only with ActiveX objects in a container.
- ActiveX property pages —Available only with ActiveX objects in a container, and not available for all ActiveX objects.
- Property Node —Use to set ActiveX properties programmatically.



RELATED INFORMATION

[Using the ActiveX Control Property Browser](#) on page 1495

[Using ActiveX Property Pages](#) on page 1495

[Creating Properties and Methods](#) on page 1476

Activating LabVIEW as an ActiveX Server

When you activate LabVIEW as an ActiveX server, LabVIEW exports two classes: an application class and a virtual instrument class. The application class opens a reference to the virtual instrument class.

Complete the following steps to activate LabVIEW as an ActiveX server.

1. Select **Tools»Options** to display the Options dialog box.
2. Select VI Server from the **Category** list.
3. Place a checkmark in the **ActiveX** checkbox in the **Protocols** section of the **VI Server** page.
4. Set the options in the **Accessible Server Resources** section of the page depending on your security needs.
5. Configure VIs in the **Exported VIs** section to allow or deny access to VIs from external clients depending on your security needs.
6. Click the **OK** button.

If you want to call a stand-alone application from an ActiveX automation client such as Visual Basic, you must enable the ActiveX server for the application when you build the application. To do so, place a checkmark in the **Enable ActiveX server** checkbox on the Advanced page of the Application Properties dialog box.

ActiveX Container Overview

Use the ActiveX container to embed ActiveX objects on a VI front panel. This allows Windows-based applications to appear and interact with LabVIEW controls. You can place

two types of ActiveX objects in the ActiveX container. You can create a new ActiveX control or document. You also can insert an existing ActiveX control or document.



RELATED INFORMATION

[Creating New Controls in an ActiveX Container](#) on page 1492

[Creating New Documents in an ActiveX Container](#) on page 1493

[Inserting Existing Controls or Documents in an ActiveX Container](#) on page 1494

Calling an ActiveX Object from LabVIEW

Use an automation refnum if you do not want to embed an ActiveX object or control in the LabVIEW front panel window.

Complete the following steps to call an ActiveX object from LabVIEW.

1. Add an [automation refnum](#) control to the front panel window.
2. Associate an ActiveX object with the automation refnum control.
 1. Right-click the control and select **Select ActiveX Class»Browse** from the shortcut menu to display the Select Object From Type Library dialog box.
 2. Select a type library from the top pull-down menu of the **Select Object From Type Library** dialog box.
 3. Select an object from the **Objects** box and click the **OK** button.
3. Open a reference to the automation refnum.
 1. Add the Automation Open function to the block diagram.
 2. Wire the automation refnum to the **Automation Refnum** input of the Automation Open function.
4. Access a property or invoke a method on the automation refnum.
 1. Add one of the following nodes to the block diagram.
 - Property Node
 - Invoke Node
 2. Wire the **Automation Refnum** output of the Automation Open function to the **reference** input of the node.
 3. Click the node to display a list of the properties or methods available for the referenced object.



NOTE

If LabVIEW does not display the properties or methods that you expect for the referenced ActiveX object, the object might implement multiple Component Object Model (COM) interfaces. LabVIEW can display the properties or methods from only one COM interface implemented by the object at a time. You can force LabVIEW to display the properties and methods from a different COM interface that the object implements by using the Variant to Data function to cast the automation refnum to the other COM interface.

- 4. Select a property or method.



Note Not all ActiveX objects expose methods.

- 1. Close the reference to the ActiveX object.
 - 1. Add the Close Reference function to the block diagram.
 - 2. Wire the **reference out** output of the Invoke Node to the **reference** input of the Close Reference function.
- 2. Run the VI. The Automation Open function opens a reference to the ActiveX object you requested in the automation refnum control in the front panel window and performs the method you requested in the Invoke Node.

COM/Automation and LabVIEW Data Types

The following table lists the COM/Automation data type and the corresponding LabVIEW data type.

COM/Automation Data Type	LabVIEW Data Type
Integer	Integer
Float	Float
BSTR	String
Date	String
Currency	String
Scode (error)	Integer
IDispatch	Automation refnum
IUnknown	Automation refnum
ActiveX variant	LabVIEW variant
SafeArray of one data type	LabVIEW array of corresponding data type
SafeArray of variant	LabVIEW cluster

Creating New Controls in an ActiveX Container

Complete the following steps to create a new ActiveX control in an ActiveX container.

- 1. Add an *ActiveX container* to the front panel window.

2. Right-click the ActiveX container in the front panel window and select **Insert ActiveX Object** from the shortcut menu to display the Select ActiveX Object dialog box.
3. Select **Create Control** from the pull-down menu in the **Select ActiveX Object** dialog box.
4. Select an ActiveX control from the list of those registered with the computer and click the **OK** button. The ActiveX control appears in the ActiveX front panel container.



NOTE

If you place a checkmark in the **Validate Servers** checkbox, LabVIEW verifies the server and lists only the servers in the registry that actually exist on your computer. If you do not place a checkmark in the **Validate Servers** checkbox, LabVIEW does not verify the server and lists all servers in the registry.

1. *Access the ActiveX properties associated with the control or document to set the properties for the control or document or **invoke a method on the object**.*

Creating New Documents in an ActiveX Container

Complete the following steps to create a new ActiveX document in an ActiveX container.

1. Add an *ActiveX container* to the front panel window.
2. Right-click the ActiveX container and select **Insert ActiveX Object** from the shortcut menu to display the Select ActiveX Object dialog box.
3. Select **Create Document** from the pull-down menu in the **Select ActiveX Object** dialog box.
4. Select the document type from the list of those registered with the computer and click the **OK** button. The ActiveX document appears in the ActiveX container.

Editing ActiveX Documents

Complete the following steps to edit an ActiveX document in the ActiveX container.

1. Right-click the ActiveX container and select **Edit Object** from the shortcut menu. The document's application opens the document in a new window.
2. Edit the document in the new window.

If the ActiveX document supports an ActiveX Automation interface, you can use the document with the ActiveX Property Node and Invoke Node on the block diagram.

Inserting an ActiveX Object on the Front Panel

To insert an ActiveX object on the front panel, complete the following steps:

1. Right-click the ActiveX container and select the ActiveX control you want to insert from the **Insert ActiveX Object** shortcut menu.
2. (Optional) *Set properties* for the object by using the ActiveX Control Property Browser, property pages, or the Property Node.



NOTE

If you want to set properties programmatically, use the Property Node.

3. (Optional) Use the Invoke Node to invoke the methods associated with the object.

You can use ActiveX objects to access common ActiveX enabled applications, such as Microsoft. For example, you can display a web page on the front panel by using an ActiveX container to access the Microsoft Web Browser control, selecting the Navigate class of methods, selecting the URL method, and specifying the URL.

If you use the ActiveX container, you do not have to open or close the reference to the container. You can wire the reference directly to the Invoke Node or Property Node because the ActiveX container embeds the calling application in LabVIEW. However, if the ActiveX container includes properties or methods that return other automation refnums, you must close these additional references.

Inserting Existing Controls or Documents in an ActiveX Container

Complete the following steps to insert an existing ActiveX control or document in an ActiveX Container.

1. Add an *ActiveX Container* to the front panel.
2. Right-click the ActiveX Container.
3. Select **Insert ActiveX Object** from the shortcut menu to display the Select ActiveX Object dialog box.
4. Select **Create Object From File** from the top pull-down menu in the **Select ActiveX Object** dialog box.
5. Click the **Browse** button and navigate to the control or document you want to insert. You can select an object anywhere on the computer. Place a checkmark in the **Link To File** checkbox to update the control or document when you update the ActiveX object. If you do not place a checkmark in the **Link To File** checkbox, LabVIEW inserts a static version of the control or document.
6. Click the **OK** button. The ActiveX control or document appears in the ActiveX Container.

Selecting an ActiveX Class

Complete the following steps to select an ActiveX server class.

1. Add the Automation Refnum control to the front panel window.
2. Right-click the Automation Refnum control and select **Select ActiveX Class»Browse** from the shortcut menu.
3. Select an ActiveX server class from the Select Object From Type Library dialog box.

After you select an ActiveX server class, you can invoke a method on that object or set a property for that object.



RELATED INFORMATION

[Types of Refnum Controls](#) on page 1069

[Creating Properties and Methods](#) on page 1476

Using ActiveX Property Pages

Many ActiveX objects include property pages, which organize the properties associated with the object on separate tabs.

Like the [ActiveX Control Property Browser](#), ActiveX property pages are an easy way to set the properties of an ActiveX object interactively, but you cannot use them to set properties programmatically, and you can use property pages only with ActiveX objects in a container. Property pages are not available for all ActiveX objects. Property pages also are not available in run mode or while a VI runs.

Complete the following steps to use ActiveX property pages to view and set ActiveX properties.

1. [Create a new control](#), [create a new document](#), or [insert an existing control or document](#) in an ActiveX container.
2. Right-click the object in the container and navigate to the name of the object in the shortcut menu. Select **Properties** to launch the **Control Properties** dialog box for that object.
3. Select a tab and edit the properties you want to set.
4. Click the **OK** button to close the property page dialog box. The changes take effect immediately.

Using the ActiveX Control Property Browser

Use the ActiveX Control Property Browser to view and set all the properties associated with an ActiveX control or document in an ActiveX container.

The ActiveX Control Property Browser is an easy way to set the properties of an ActiveX object interactively, but you cannot use the browser to set properties programmatically, and you can use the ActiveX Control Property Browser only with ActiveX objects in a container. The ActiveX Control Property Browser is not available in run mode or while a VI runs.

Complete the following steps to use the ActiveX Control Property Browser to view and set ActiveX properties.

1. *Create a new control, create a new document, or insert an existing control or document* in an ActiveX container.
2. Right-click the object in the container and select **Property Browser** from the shortcut menu.
3. Set the properties. Find the property you want to set in the left column of the Property Browser, and modify its value in the right column of the Property Browser. Notice that the help at the bottom of the Property Browser updates to describe the property you currently have selected. The changes take effect immediately.
4. Close the Property Browser.

Editing Existing Property and Invoke Nodes

You can use the Select Property or Select Method dialog box to edit an existing Property Node or Invoke Node.

Complete the following steps to edit a Property Node or Invoke Node.

1. Right-click a property on a Property Node or a method on an Invoke Node and select **Select Property»Browse** or **Select Method»Browse** from the shortcut menu. The **Select Property** or **Select Method** dialog box appears.
2. From the **Properties** or **Methods and Dottable Properties** list of the dialog box, select a property or method to replace the one currently shown in the Property Node or Invoke Node. The property or method you select appears in the **Selected property** or **Selected method** box.
3. Click the **OK** button to update the node on the block diagram.

Linking Property Nodes and Invoke Nodes to Terminals

If you *copy or clone* a Property Node or Invoke Node, LabVIEW can disconnect the node from its front panel object.

Complete the following steps to link a Property Node or Invoke Node to a front panel object.

1. Right-click the node.

2. Select **Link to** and the name of an object from the shortcut menu. LabVIEW replaces the node with an *implicitly linked node*.
3. Right-click the node again and select **Disconnect From Control** from the shortcut menu to remove the implicitly linked node.

You also can use the **Link to** shortcut menu to associate an implicitly linked node with a different front panel object. The **Link to** shortcut menu includes an option for each front panel object in the VI.

Connectivity in LabVIEW

Developing and Distributing an Application

Requires: Application Builder or Professional Development System

You can convert a LabVIEW project into a distributable application for use on other computers and from other programming languages. Complete the steps in this document to create any of the following *types of distributable applications*:

- Stand-alone applications
- Installers
- .NET Framework assemblies
- Packages
- Packed project libraries
- Shared libraries
- Source distributions
- Zip files

Preparing to Build the Application

1. **Open the *LabVIEW project* from which you want to build the application.** You must build an application from a project, rather than from an individual VI.
2. **Save the entire project to ensure that all VIs are saved in the current version of LabVIEW.**
3. **Verify the settings in the VI Properties dialog box for each VI.** If you plan to distribute your application, ensure that the settings in the **VI Properties** dialog box are accurate for the built version of your VIs. For example, to polish the appearance of your built application, verify the settings on the following pages of the **VI Properties** dialog box:
 - Window Appearance
 - Window Size

- Window Run-Time Position

**NOTE**

If your application contains VIs with *separate compiled code*, you can configure settings for those VIs on the **Source Files Settings** page of the **Properties** dialog box for your application.

4. **Verify that the paths used in the development environment still behave correctly on destination computers.** If the project *loads VIs dynamically*, use relative paths, instead of absolute paths, to specify the location of the VI. Because file hierarchies can vary on different computers, *relative paths* ensure that paths function correctly in both the development environment and the destination computers on which the application runs.

**NOTE**

If your application *uses the 8.x file layout*, you may need to use a different relative path on the destination computer from the relative path in the development environment.

You can specify the destination for the generated files on the **Destinations** page of the properties dialog box for the build specification you are creating.

5. **Verify that the Current VI's Path function returns the path you expect.** In a stand-alone application or shared library, the Current VI's Path function returns the path to the VI in the application file and treats the application file as an LLB. For example, if you build `foo.vi` into an application, the function returns a path of `C:\...\Application.exe\foo.vi`, where `C:\...\Application.exe` represents the path to the application and its filename.
6. **Ensure that VI Server properties and methods work as expected in the LabVIEW Run-Time Engine.** The LabVIEW Run-Time Engine does not support some VI Server properties and methods. Therefore, avoid using these properties and methods in the VIs that you include in an application or shared library. You can run the Built Application Compatibility test from the VI Analyzer Toolkit to ensure that your VI Server properties are compatible with the LabVIEW Run-Time Engine.

Configuring Specifications for a Built Application

**NOTE**

If you save a custom run-time menu for a front panel control as a separate run-time menu file, or `.rtm` file, and want to include this control in an application build, you must add the `.rtm` file to the **Always Included** listbox on **Source Files** page of the **Properties** dialog box for your application.

1. **Create the build specification.** Expand **My Computer** in the **Project Explorer** window. Right-click **Build Specifications** and select **New»your application type** from the shortcut menu to display the **Properties** dialog box for your application. If you previously hid **Build Specifications** in the **Project Explorer** window, you must display the item again to access it.

2. **From the Properties dialog box for your application, configure the required pages of the build specification.**



NOTE

If you want to include any type of application within an installer, make sure to specify that all the files in the application are relative to the primary destination of the application. Otherwise, when you include the build output of the application within an installer, the installer will reorganize the original file organization of the application by moving all files that are not relative into the primary destination. You can specify the primary destination for any type of application on the **Destination** page of the **Properties** dialog box for the application.

Choose the type of application you want to build from the following list:

Stand-Alone Application

- **Required Configurations**
 - Source Files
- **Recommended Configurations**
 - Information
 - Destinations
 - Source File Settings
- **Additional Information**
 - To ensure that the LabVIEW Run-Time Engine can load all VIs, use the Mark Project Items to Separate Compiled Code dialog box to include compiled code in all VIs.

Installer

- **Required Configurations**
 - Source Files
- **Recommended Configurations**
 - Product Information
 - Destinations
 - Source Files Settings
 - Version Information
- **Additional Information**
 - To ensure that the LabVIEW Run-Time Engine can load all VIs, use the Mark Project Items to Separate Compiled Code dialog box to include compiled code in all VIs. Refer to the [caveats and recommendations for building installers](#) for more information.

.NET Interop Assembly

If you have the .NET Framework 4.0 installed on your development computer, you can use the Application Builder to build a .NET interop assembly.

- **Required Configurations**
 - Source Files
- **Recommended Configurations**
 - Information
 - Destinations
 - Source File Settings
- **Additional Information**
 - You also can access LabVIEW classes through a .NET interop assembly. LabVIEW can generate a .NET class for LabVIEW classes that you specify, and you access these .NET classes through a .NET assembly.
 - To ensure that the LabVIEW Run-Time Engine can load all VIs, use the Mark Project Items to Separate Compiled Code dialog box to include compiled code in all VIs. Refer to the [characteristics of building .NET interop assemblies](#) for more information.

Package

- **Recommended Configurations**
 - Source Files
 - Information
 - Destinations
 - Package
 - Package Installer
 - Feed
- **Additional Information**
 - You cannot distribute stand-alone VIs.
 - Refer to the [Creating Packages for Distribution](#) topic for more information about creating packages.

Packed Project Library

- **Required Configurations**
 - Source Files
- **Recommended Configurations**
 - Information
 - Destinations
 - Source File Settings

- **Additional Information** Refer to the [caveats and recommendations for packed project libraries](#) for more information.

Shared Library

- **Required Configurations**
 - Source Files
- **Recommended Configurations**
 - Information
 - Destinations
 - Source Files Settings
- **Additional Information** Refer to the [Characteristics of LabVIEW-built shared libraries](#) for more information.

Source Distribution

- **Required Configurations**
 - Source Files
- **Recommended Configurations**
 - Information
 - Destinations
 - Source File Settings

Zip File

- **Required Configurations**
 - Source Files
- **Recommended Configurations**
 - Zip Information
 - Zip File Structure
- **Additional Information**
 - If you select the **Zip entire project** option from the Source Files page of the Zip File Properties dialog box, LabVIEW does not automatically include outputs from other build specifications in the **Included Items** tree. You must use the arrow buttons to add a source distribution, shared library, or application to the zip file.
 - Outputs of zip or installer build specifications do not appear in the **Project Files** tree on the **Source Files** page. To include another zip file or installer in the zip file, add the zip file or installer you want to include to the LabVIEW project under the target from which you are building the zip file.

- You can use the Zip VIs or the Build VI to build zip files programmatically. To use the Zip VIs, you must know the relative path for every file you want to include in the zip file.
3. **Explicitly include dynamically loaded VIs in the build specification.**
If a VI loads other VIs *dynamically* using the *VI Server* or calls a dynamically loaded VI through a Call By Reference or Start Asynchronous Call node, you must add those VIs to the **Always Included** listbox on the **Source Files** page of the **Properties** dialog box for your application.

You also can distribute dynamically loaded VIs for an application by including them within a source distribution.
 4. **Save the updated settings for your build specification.** Click the **OK** button to update the build specification in the project and close the dialog box. The build specification name appears in the project under **Build Specifications**. To save changes you make to a build specification, you must save the project that contains the build specification.

Building the Application

Right-click the build specification name for the application you want to build, and select **Build** from the shortcut menu. You also can build an application programmatically using the Build VI or the *ExecuteBuildSpec* command line operation.



NOTE

You can preview your application to ensure that it appears correctly. From the Preview page of the **Properties** dialog box for your application, click the **Generate Preview** button to review the generated files for your application.

Distributing the Built Application

1. **Ensure that the computer on which you want to run the application has access to the LabVIEW Run-Time Engine.** The *LabVIEW Run-Time Engine* must be installed on any computer on which users run an application or shared library. You can distribute the LabVIEW Run-Time Engine with the application or shared library. (Windows) You also can include the LabVIEW Run-Time Engine in an installer.
2. **Distribute legal information for your end user.** If you use installers to distribute your applications, enter information for a custom license agreement on the Dialog Information page of the Installer Properties dialog box. Review <National Instruments>\Legal Information.txt for information about including legal information in installers built with NI products.
3. **Refer to the following table for tips to assist you with distributing LabVIEW-built applications.**

Action	Benefit
To verify that the built application performs correctly, enable debugging by placing a checkmark in the Enable Debugging checkbox on the Advanced page of the Properties dialog box for your application. You also can connect to a built application by selecting Operate»Debug Application or Shared Library .	When you test a built application, you ensure that behavior did not change between the development environment and the distribution environment.
After you test your application, <i>disable debugging</i> for your VIs.	Disabling debugging for VIs can reduce file size and increase run-time performance.
Create an <i>About</i> dialog box for a stand-alone application.	Most professional applications include an About dialog box that provides general information about the application, such as version, copyright, and support information.
If your end user has spoken language requirements that differ from the language of the original application, adjust the default language settings for the application on the Run-Time Languages page of the Properties dialog box for your application.	When you distribute an application, dialog boxes and menus adopt the spoken language of the operating system on which the application runs, but the text and controls you implement remain in the language of the operating system on which you originally built the application. By default, stand-alone application and shared library build specifications provide support for Chinese (Simplified), English, French, German, Japanese, and Korean.
If your application uses the same VI Server port as another application, <i>create a custom configuration file</i> .	If you try to run an application that requires the VI Server at the same time as the conflicting application, the VI Server does not run, and LabVIEW does not provide a warning.
Distribute a custom configuration file along with your built application.	If you need to ensure that the application always runs under certain LabVIEW environment settings, you can preserve and redistribute those settings by distributing a <i>custom configuration file</i> , or preference file, with your application.

Building a Universal Framework on macOS

A universal framework, or shared library, contains both 64-bit and 32-bit code so that both 64-bit and 32-bit applications can call the framework. You need both 64-bit and 32-bit versions of LabVIEW to build a universal framework. Complete the following steps to build a universal framework in LabVIEW:

1. Using 64-bit LabVIEW, build the framework that you want to make universal. LabVIEW builds the framework with 64-bit VI code.

- Using 32-bit LabVIEW, build the same framework you built in the previous step. LabVIEW builds the framework with 32-bit VI code.
- Navigate to the 32-bit framework on disk, and copy the .11b file from the 32-bit framework into the 64-bit framework you created in step 1. After you copy the .11b file into the 64-bit framework, you no longer need the 32-bit framework you created in step 2.

**NOTE**

You must copy the .11b file from the 32-bit framework into the 64-bit framework. The framework does not work universally if you copy the .11b file from the 64-bit framework into the 32-bit framework.

You can now call the framework you built in step 1 from both 32-bit and 64-bit applications.

Related Information

[Building and Distributing Applications](#)

[How to Build a Framework \(Shared Library\)](#)

[Debugging Applications and Shared Libraries](#)

Creating Packages for Distribution

You can create packages and deploy them to clients through NI Package Manager or SystemLink.

Before creating packages, create a source distribution, packed project library, shared library,.NET assembly, or executable to include in your packages.

**NOTE**

You cannot distribute stand-alone VIs.

- In the Project Explorer window, right-click **Build Specifications** and select **New»Package**.
- In the Package Properties dialog box, under **Information**, enter the name you want to use for your package in **Package name**. The package name must be all lowercase and contain no spaces. NI Package Manager and other packages use this name to reference the package you create.
- Under **Destinations**, set up any destination folders where you plan to install the package in the client.
- Under **Source Files**, specify which files or build specification outputs you want to install.
- (Windows) Under **Shortcuts**, you can add a shortcut to the Startup menu if you want your application to run as soon as the system starts up. You can also add a

shortcut to the Desktop or Program Menu on Windows. By default, LabVIEW adds a shortcut in the Windows Program menu for each application included in the package.

6. Under **Package**, specify the display name and synopsis for your package. The display name is what users will see in the Package Manager or NI SystemLink when they install your package. The synopsis is the first line of the description that users see before they click on a package to see its full description.
7. Under **Dependencies**, specify dependencies on any currently installed packages.
 1. Configure the relationship of the dependency. You can define a dependency as **required**, **recommended**, or **suggested** and specify version ranges.
 2. To prompt the user to install your package when they install a related package, place a checkmark in the **Include enhanced relationship** checkbox.
8. Under **Version Information**, specify the version of your package and what version you want to display to the user. If you remove the display version, the user sees the full version in the Package Manager.
9. Under **Advanced**, set your package to run executables or scripts while it installs or uninstalls. For example, you can specify whether the process for installing or uninstalling the package waits for a certain action to complete.
10. (Optional) Under **Package Installer**, place a checkmark in the **Create a Package Installer** checkbox to build your distribution into a package installer file (.exe).
11. (Optional) Under **Feed**, choose whether to add your package to a local feed or publish the package to a feed on the SystemLink server.
12. Click **Build**. A (Windows).nipkg file or (NI Linux Real-Time).ipk file appears in the destination you specified.

You can distribute the package to clients through Package Manager or SystemLink. (Windows) Your clients can use Package Manager or SystemLink to subscribe to a feed to find and install your packages. (NI Linux Real-Time) Your clients can install packages through SystemLink or from the command line on the NI Linux Real-Time target. Package Manager does not support .ipk files.

Configuring Run-Time Languages for Applications and Shared Libraries

You can specify the language or languages to use with a stand-alone application or shared library. By default, LabVIEW uses the same run-time language as Windows. If LabVIEW cannot use the Windows run-time language or the platform is non-Windows, LabVIEW

defaults to English. Change the default behavior by configuring the run-time language options.



NOTE

- Language settings are only available in the 32-bit version of LabVIEW.
- LabVIEW localizes text for items such as default run-time menus, default menu items for custom run-time menus, descriptions of error codes, and LabVIEW dialog boxes. However, you must *localize the text* for items such as captions and control descriptions.

You can configure the run-time language options for a stand-alone application using the following methods. The methods appear in order of precedence.

- Run the application with the `-lang language` command-line argument, where language is the English name of the language you want to use.
- Add the following line to the *configuration file* for the application: `AppLanguage=language`, where language is the English name of the language you want to use. Modifying the configuration file for the application does not modify the project settings.
- (Windows) If you do not specify the language you want to use in the command line or the configuration file of the application, the application uses the operating system language. Navigate to **Control Panel»Regional and Language Options** to view the language settings for the operating system. If the application does not support the operating system language, the application uses the default language for the application.
- Use the **Default language** option to configure the default language for the application. In the **Project Explorer** window, right-click an application under **Build Specifications**, select **Properties** from the shortcut menu, and navigate to the **Run-Time Languages** page of the Application Properties dialog box. If you previously hid **Build Specifications** in the **Project Explorer** window, you must *display the item* again to access it.
- Use the **Supported Languages** option to configure the supported languages for an application in the build specification. In the **Project Explorer** window, right-click an application under **Build Specifications**, select **Properties** from the shortcut menu, and navigate to the **Run-Time Languages** page of the Application Properties dialog box to configure language options for the application. The **Supported languages** option on the **Run-Time Languages** page lists all supported languages.

If you do not specify a language using the previous options, the application uses the first available supported language in alphabetical order. The **Supported languages** option on the **Run-Time Languages** page of the Application Properties dialog box lists all supported languages. If the application cannot use any of the supported languages because the required language files have been deleted, corrupted, or are not readable, LabVIEW cannot load the application.



NOTE

The *LabVIEW Run-Time Engine* is multilingual, so you do not have to include multiple versions of it with an application, shared library, or installer in order to use localized text.

You can use the Application:Language property to determine the default language for an application programmatically.

Use the Run-Time Languages page of the Shared Library Properties dialog box to configure language options for a shared library.

Copying LabVIEW Work Environment Options

You can copy the *options you set* in the LabVIEW work environment to the stand-alone application you build.

If you use a *LabVIEW configuration file* with the stand-alone application, you do not need to manually copy LabVIEW work environment options. The Application Builder automatically copies the options that you set in the LabVIEW work environment. If you use a *custom configuration file* with the stand-alone application and the file does not contain the LabVIEW work environment options you want, you can manually copy the entries that specify the LabVIEW work environment options from the LabVIEW configuration file.

Complete the following steps to manually copy LabVIEW work environment options.

1. Open the LabVIEW configuration file. Refer to the following table for the name and location of the LabVIEW configuration file. In the following table, x is the LabVIEW version currently in use.

Platform	LabVIEW Configuration File
	LabVIEW x \labview.ini
	~/Library/Preferences/LabVIEW.app x 64-bit Preferences
	/home/<username>/natinst/.config/LabVIEW-x/labview.conf

2. In the LabVIEW configuration file, copy the entries that specify the LabVIEW work environment options you want. The entries that specify the LabVIEW work environment options are in the following format: `preference_name = value`.
3. Paste the entries to the stand-alone application configuration file. Refer to the following table for the name of the stand-alone application configuration file. In the following table, myapp is the name of the stand-alone application and x is the LabVIEW version currently in use.

Platform	Stand-Alone Application Configuration File
	myapp .ini
	myapp .app x Preferences
	myapp .conf

4. Save the stand-alone application configuration file.

Related Information

How LabVIEW Stores Options

Creating an About Dialog Box for a Stand-Alone Application

Most applications have an **About** dialog box that displays general information about the application and the user or company that designed it. You can create a VI that LabVIEW runs when a user selects **Help»About** to display information about the stand-alone application you create. After the VI runs, LabVIEW closes the dialog box.

You can have only one About VI per application. If you do not supply an About VI, LabVIEW displays a default dialog box similar to the dialog box that appears in the LabVIEW development system when you select **Help»About**.

Complete the following steps to create an About VI and include it in a stand-alone application.

1. Build a VI with a front panel window that contains the information you want to display in the **About** dialog box, such as the version number, company name, and copyright information. The About VI you create can share subVIs with other VIs you include in the application. However, you cannot use the About VI as a subVI because the About VI cannot run while other VIs in the application run.



NOTE

The front panel window must include a National Instruments copyright notice. Refer to the National Instruments Software License Agreement located on the LabVIEW Platform media for more information about the requirements for any **About** dialog box you create for a LabVIEW application.

1. Build the block diagram of the About VI. For example, if you want the **About** dialog box to display until the user clicks a button, use a While Loop with an Event structure on the block diagram.
2. Save the VI. The name of the VI must start with About.
3. Add the VI to the project from which you are building the application.
4. When you configure the application build settings, add the About VI to the **Always Included** list on the Source Files page of the Application Properties dialog box.
5. From the Source File Settings page, select the About VI in the **Project Files** tree. In the **Use default save settings** section, make sure a checkmark does not appear in the **Remove front panel** checkbox and that the destination is the application.
6. Configure the remaining application build settings and build the application.

Enabling the Web Server in Stand-Alone Applications

To **embed the front panel** of a stand-alone application in an HTML document, **enable and configure** the Web Server in a application as you would in a VI. Include the front panel of the VI and the HTML file in the installation.



NOTE

When you build an application, on the Source File Settings page of the Application Properties dialog box, remove the checkmark from the **Remove front panel** checkbox for the VI you want to publish so that you can embed the front panel in an HTML document.

After you generate the application, the output folder contains both an `.ini` configuration file and a `niwebserver.conf` configuration file.

In the `.ini` file, add the line `WebServer.Enabled=True` if it does not already exist. You also can copy the `WebServer*` preferences from the LabVIEW INI file for the LabVIEW development system to the `.ini` file for the application. You may need to modify these preferences to be appropriate for the application.



Note When running the application, make sure no other application, including the LabVIEW development environment, uses the same port.

Use the `niwebserver.conf` file to configure other Web Server functionality in the application, such as the **root directory of the Web Server**.

Retrieving Application and Shared Library Versions Programmatically (Windows)

You can use the `fileVersionInfo.llb` located in the `labview\vi.lib` \Platform directory to retrieve the version information of a stand-alone application or shared library programmatically. The VI is not available on the **Functions** palette.



NOTE

The LabVIEW Professional Development System includes the Application Builder. If you use the LabVIEW Base Development System or Full Development System, you can purchase the Application Builder separately by visiting the National Instruments Web site.

Complete the following steps to retrieve the version information of an application or shared library programmatically.

1. In the `FileVersionInfo` VI, the top-level VI of the LLB, specify in the **Application path** in path control the location on disk of the application or shared library whose version information you want to retrieve.

2. Run the VI.

LabVIEW returns the version information in the **File Version Info** and **FileVersionInfo Out** indicators.

Using a Configuration File with a Stand-Alone Application

You can use a LabVIEW configuration file or another, custom, configuration file with a LabVIEW stand-alone application.

LabVIEW Configuration Files

If you select to use the LabVIEW configuration file with a stand-alone application, LabVIEW creates a configuration file that includes a subset of the configuration tokens that appear in the LabVIEW configuration file in the `labview` directory. You can configure additional build specification settings using the Application Properties dialog box. For example, if you select **Enable debugging** from the Advanced page of the **Application Properties** dialog box, LabVIEW adds debugging tokens to the configuration file of the stand-alone application.

Complete the following steps to use the LabVIEW configuration file in a stand-alone application.

1. In the **Project Explorer** window, right-click **Build Specifications** and select **New»Application** from the shortcut menu. If you previously created a build specification for the application, double-click the build in the **Project Explorer** window to edit the build.
2. On the **Advanced** page of the **Application Properties** dialog box, ensure that the **Use custom configuration file** checkbox is not enabled.
3. (Optional) Make other selections on the **Advanced** page. Some of these changes add configuration tokens to the configuration file of the stand-alone application.
4. Click the **OK** button to apply the changes and close the dialog box.

Custom Configuration Files

If you use a custom configuration file with a stand-alone application, LabVIEW uses the configuration file you specify as a template. LabVIEW adds additional tokens to the configuration file you specify depending on the options you select in the **Application Properties** dialog box. For example, if you select **Enable debugging** from the **Advanced** page of the **Application Properties** dialog box, LabVIEW adds debugging tokens to the custom configuration file you included.

Complete the following steps to use a custom configuration file in a stand-alone application.

1. In the **Project Explorer** window, right-click **Build Specifications** and select **New»Application** from the shortcut menu. If you previously created a build specification for the application, double-click the build in the **Project Explorer** window to edit the build.
2. On the **Advanced** page of the **Application Properties** dialog box, place a checkmark in the **Use custom configuration file** checkbox. The Select Project File dialog box appears.
3. Select the configuration file you want to include from the **Files in Project** list. If the file you want to include is not in the project, complete the following steps.
 1. Click the **Add** button. A file dialog box appears.
 2. Browse to the file you want to include and click the **Add** button. The dialog boxes close and the file you selected appears in the **Configuration file in project** text box on the **Advanced** page of the **Application Properties** dialog box.
4. (Optional) Make other selections on the **Advanced** page. Some of these changes add configuration tokens to the custom configuration file.
5. Click the **OK** button to apply the changes and close the dialog box.

Characteristics of LabVIEW-Built .NET Framework Assemblies



NOTE

The following content applies to .NET Framework only. Prior to LabVIEW 2025 Q1, .NET Framework Assemblies were called .NET Interop Assemblies.

Calling a .NET Framework Assembly from Another Program

After you build a .NET Framework assembly, you can call the assembly from other programs that support .NET. However, ensure that the computer on which the other program runs meets the following requirements and recommendations:

- The LabVIEW Run-Time Engine must be installed on any computer on which users run the .NET Framework assembly. You can distribute the LabVIEW Run-Time Engine with the .NET Framework assembly. (Windows) You also can include the LabVIEW Run-Time Engine in an installer.
- National Instruments recommends that the target computer for the .NET Framework assembly have the same or later version of the .NET Framework installed as the version that LabVIEW used to build the application.
- To call the .NET Framework assembly outside of the LabVIEW development environment, you must reference `NationalInstruments.LabVIEW.Interop.dll` in the Microsoft Visual Studio project. The LabVIEW Run-Time Engine automatically installs this DLL to the `National Instruments\Shared\LabVIEW Run-Time` directory. Optionally, you can manually install `NationalInstruments.LabVIEW.Interop.dll` to the Global Assembly

Cache (GAC). If you want to install this DLL to the GAC, you must sign the .NET assembly with a strong name key file.

- When you create a debuggable .NET Framework assembly, LabVIEW places a configuration (ini) file in the same directory as the assembly. If you want to debug the assembly in other programs, you must distribute the .ini file with the assembly.

LabVIEW Data Types in .NET Framework Assemblies

When you build a .NET Framework assembly, LabVIEW directly translates simple data types such as numerics, booleans, strings, and arrays of simple data types to the corresponding .NET data type. However, LabVIEW-specific data types, such as clusters, waveforms, complex numbers, refnums, and LabVIEW classes, require LabVIEW to define new .NET data types in the generated assembly.

One way to determine how LabVIEW converts its own data types to .NET data types is to build the .NET Framework assembly and then view the manifest of the assembly in a .NET programming environment such as Microsoft Visual Studio. You can also use the MSIL Disassembler tool available on the Microsoft Developer Network (MSDN) to view the generated assembly. However, the following sections explain some basic details about how LabVIEW converts clusters, enumeration type controls, and LabVIEW classes to .NET data types.

Clusters and Enumerations

LabVIEW converts clusters and enumerations to .NET structures whose elements correspond to the elements of the cluster or enumeration. LabVIEW uses the following rules to name the .NET structure that corresponds to a cluster or enumeration:

- Clusters and enumerations defined as type definitions or strict type definitions—LabVIEW uses the label of the type definition or strict type definition as the name of the .NET structure
- All other clusters and enumerations—LabVIEW names the .NET structure LVCluster_# or LVEnum_#

LabVIEW Classes

You can export the member VIs of a LabVIEW class. In LabVIEW, these VIs require the owning LabVIEW class as an input. Therefore, when you export these VIs to a .NET Framework assembly, LabVIEW must create an equivalent .NET class definition in the assembly. The .NET class definition contains a static method for each member VI. Each of these static methods requires an instance of the newly defined .NET class as a parameter. LabVIEW does not export the private data of the owning LabVIEW class.

LabVIEW also exports the inheritance structure of LabVIEW classes whose method VIs you export. The generated .NET classes have the same inheritance structure.

LabVIEW Errors

When LabVIEW generates a .NET method for a VI, LabVIEW does not export the **error in** and **error out** clusters as parameters of the new method. Instead, the new method throws a .NET exception if an error occurs while the method is executing. This exception contains the same information as the error cluster.

Characteristics of LabVIEW-Built Shared Libraries

None

LVDLLStatus Function

All DLLs built from LabVIEW, in addition to the functions you export, contain an exported function called `LVDLLStatus`, with the following prototype:

```
MgErr LVDLLStatus(CStr errStr, int32 errStrLen, void *module)
```

The calling program uses this function to verify that the LabVIEW DLL loaded correctly. If an error occurs while loading the DLL, the function returns the error. Pass a string buffer to the `errStr` parameter to receive additional information about the error. Set the `errStrLen` parameter to the number of bytes in the string buffer passed as `errStr`. You can use the `module` parameter to retrieve the handle to the LabVIEW Run-Time Engine being used by the DLL. Typically, you can leave this parameter as `NULL`.

The `LVDLLStatus` function blocks the execution if LabVIEW calls this function while the LabVIEW Run-Time Engine is still loading.



NOTE

If you call a LabVIEW-built DLL from within LabVIEW, LabVIEW automatically calls the `LVDLLStatus` function for you.

SetExecuteVIsInPrivateExecutionSystem Function

All DLLs built from LabVIEW, in addition to the functions you export, contain an exported function called `SetExecuteVIsInPrivateExecutionSystem`, with the following prototype:

```
Void SetExecuteVIsInPrivateExecutionSystem(Bool32 value)
```

Use this function to specify whether LabVIEW executes VIs in a multithreaded execution system when you call LabVIEW-built shared libraries to applications in other languages. If you pass 1 to the `value` parameter, LabVIEW executes VIs in a multithreaded execution system when you call LabVIEW-built shared libraries to applications in other languages. If you call this function, the `value` parameter of this function overrides the value of the **Execute VIs in private execution system** checkbox in the Advanced page of the Shared Library Properties Dialog Box.

(macOS) To build part of a shared library in LabVIEW, Application Builder must use gcc, the standard command-line compiler for Unix and macOS operating systems. Use Xcode, the standard development environment for Apple computers, to install gcc. You can download Xcode for free from the Mac App Store.

Memory Allocation

To manage memory allocation for LabVIEW array types exposed as handles, the LabVIEW-built shared library provides `Allocate`, `Deallocate`, and `Resize` functions for each array type exposed as a handle. Open the generated header file to view the function prototypes specific to the generated shared library.

Caching Installer Distribution Components (Windows)

A distribution is a collection of files that form an installer. When you use the Application Builder to *build an installer* that contains additional installers or components, the Application Builder prompts you to locate the original distribution for the components. Caching installer distribution components copies them into a permanent location on the local system and eliminates the need to specify a location for the additional components each time you build the installer. The next time you build an installer that includes these components, the Application Builder automatically copies the components from your local drive instead of prompting you for a distribution media.

Complete the following steps to cache installer distribution components:

1. Right-click **Build Specifications** from the Project Explorer window and select **New»Installer** from the shortcut menu.
2. Select Additional Installers from the **Category** list.
3. Place a checkmark in the **To minimize media prompts while building your installers, copy the selected installers and all future installers to this computer** checkbox.
4. Click **OK**.



NOTE

The LabVIEW Professional Development System includes the Application Builder. If you use the LabVIEW Base Development System or Full Development System, you can purchase the Application Builder separately from ni.com.

Caveats and Recommendations for Building Installers

The following list describes some of the caveats and recommendations to consider when you build an installer.

- When you select files to include in the installer build from the **Source Files** page of the Installer Properties dialog box, you cannot select part of a build specification. You must include build specifications as a whole. The entire specification appears in the **Destination View** tree.

- If you want to include a project library in the installer build, make sure the project library does not include files on a network or other files with links that will break if moved. You cannot copy a project library from the **Project Files View** directory to the **Destination View** directory if any of the files are on different drives, or if the files do not share a common path with the LabVIEW project file `.lvproj`.
- Consider building a source distribution that contains the files you want to include in an installer build, to reduce the possibility of broken links for files on networks or different drives.
- When you [add a National Instruments product installer](#) to the installer build, the drivers and components for NI products that you include in the installer contain only those features that are installed on the computer on which you build the installer. For example, if you have only 10 of a possible 20 features of an NI product installed on the build computer and then select to include the full version of that product in the installer, the installer will include only the 10 features installed on the build computer, not all 20 features available for that product. It is possible for an installer you create to uninstall components on the computer on which it is installed without reinstalling current versions of those components. This behavior depends on the NI products installed on the build computer and those installed on the computer on which you run the installer. If the computer on which you run the installer has an NI product installed that is not installed on the build computer, that product may be removed during the installation. The installer displays information to you that explains what features will be removed as part of the installation before uninstalling any components and gives you the option to stop the installation.
- You can include [custom error codes](#) in the installer. The `[LabVIEW x Run-Time]` folder in the **Destination View** directory corresponds to the `Shared\LabVIEW Run-Time\x.x` directory, where `xx` is the LabVIEW version currently in use. If you place a checkmark in the **Install custom error code files** checkbox on the Advanced page, the installer build includes all error code files from `labview\project\errors` and `labview\user.lib\errors` and installs them in the `Shared\LabVIEW Run-Time\x.x\errors` directory.



NOTE

You must manually create an `errors` folder in the `labview\user.lib` directory to organize your error code files.

- If you receive an out of memory error when building an application for an installer build, consider one of the following solutions:
 - If the application includes large or complex VIs, lower the complexity threshold at which LabVIEW prioritizes editor responsiveness over execution speed. This change allows LabVIEW to devote more memory to the build process.
 - Reduce the number of VIs in your application by dividing portions of the application into separate source distributions. The VIs in the application use the Call By Reference node to call separate portions.
- When you test a stand-alone application, shared library, .NET assembly, or [Web service](#) for an installer build, if the load time takes longer than you expect, remove portions of the VIs that other VIs call dynamically once or occasionally. For example, if an application has a configuration dialog box that consists of many VIs and the user has to choose to load the dialog box, use the VI Call Configuration dialog box to [configure when the subVI loads](#) by selecting the **Reload for each call** option. When

you create applications, shared libraries, and .NET assemblies, modify the build so that it has additional destinations. To reduce load time, configure the VIs that you call dynamically to go into those additional destinations, rather than the main application or shared library.

- Use the [LabVIEW x] folder in the **Destination View** directory to add files to install in the directory of the latest version of LabVIEW that the user has installed.
 - If you include the build specification for any type of application within an installer, ensure that the build output of the build specification is relative to the primary destination of the build specification. The primary destination is specified on the **Destination** page of the **Properties** dialog box for the application. If the build output includes any files that are not relative to its primary destination, the installer will reorganize the original file hierarchy of the application. In particular, the installer will move any files that are not relative into the primary destination.
- You do not need additional license files when using LabVIEW distribution components with LabVIEW-built applications.
- When you add a build specification to a folder in the **Destination View** tree of the Source Files page of the Installer Properties dialog box, the name of the build appears in the **Destination View** tree, even though a folder with the name of the build does not exist.

Including Additional National Instruments Installers in an Installer Build

When you build an installer, you might need to include installers for devices or other National Instruments products in the installer build. For example, if you build an installer for a stand-alone application or shared library that uses the Storage/DataPlugin VIs, you must include the NI USI installer. By default, LabVIEW automatically adds installers for the National Instruments products that you installed on the computer you are using.



CAUTION

When you add a National Instruments product installer to the installer build, you are adding only the components of the installer that you installed on the computer you are using. When users run the installer you built, the installer might remove components of the product from the user computer that are not part of the installer. To minimize the risk of removing product components on user computers, ensure you have a complete and up-to-date installation of the product on the computer you are using, including all optional components, before you add the product installer to the build.

Manually Selecting Additional Installers

To prevent LabVIEW from selecting installers automatically, remove the checkmark from the **Automatically select recommended installers** checkbox on the Additional Installers page of the Installer Properties dialog box for your installer. You can then select the installers you want to include with your installer from the **National Instruments Installers to Include** list. This list might include but is not limited to the following installers:

- **NI LabVIEW Run-Time Engine** —Include this installer with applications and shared libraries. The LabVIEW Run-Time Engine includes 3D graph support and NI Reports, which provides [support for generating LabVIEW reports](#) in applications and shared libraries. You can specify whether to include the following components of the LabVIEW Run-Time Engine:
 - **Math Kernel Libraries** —Include this component for algorithms the Analysis VIs use for specific processors.
 - **NI ActiveX Container** —Include this component to distribute NI ActiveX objects, properties, methods, and events with an installer.
 - **NI Error Reporting** —Include this component to detect application crashes and internal warnings. Enabling this component also allows you to create reports on disk that contain memory dumps and log files that can help you troubleshoot an issue.
 - **NI LabVIEW Real-Time NBFifo** —Include this component when using LabVIEW Real-Time FIFOs on desktop computers.
 - **NI LabVIEW Run-Time Engine Non-English Support** —Include this component to make sure that the LabVIEW Run-Time Engine reports error messages in the language of the operating system on which you run an application type.
 - **NI LabVIEW Run-Time Engine Web Server** —Include this component when using the Web Server for stand-alone applications and shared libraries.
 - **NI System Web Server** —Include this component when using Web services to communicate between National Instruments software, such as LabVIEW and NI Measurement & Automation Explorer.
 - **NI Logos 5.4** —Include this component for the network protocol that uses network-published shared variables.
 - **NI TDM Streaming** —Include this component when using TDMS VIs to perform file I/O operations.
 - **NI VC2008MSMs** —Include this component to install Microsoft Visual C++ Runtime to load and run the LabVIEW Run-Time Engine on computers where the Microsoft Visual C++ Runtime has not been previously installed.
 - **NI VC2010MSMs** —This component installs Microsoft Visual C++ Runtime to load and run the LabVIEW Run-Time Engine with .NET Framework support. Include this component on computers where the LabVIEW Run-Time Engine has not been previously installed.
- **NI LabWindows/CVI; Run-Time Engine** —Include this installer with shared libraries (DLLs) built in LabWindows™/CVI™.
- **NI USI** —Include this installer if the application or shared library uses the Storage/DataPlugin VIs.
- **NI LabVIEW DSC Module Runtime** —Include this installer if the application or shared library uses LabVIEW DSC Module features, including security, alarming, and logging.
- **NI DataSocket** —Include this installer if the application or shared library uses [DataSocket](#). Includes the DataSocket Server, DataSocket Server Manager, and OPC Demo Server.
- **NI Measurement & Automation Explorer** —Include this installer to configure, test, and communicate with National Instruments hardware.

- **NI TDM Excel Add-In** —Include this installer to load .tdm and .tdms files from Microsoft Excel.
- **NI Distributed System Manager** —Include this installer to create and monitor variables, processes, I/O servers, and Web services, as well as to interact with the Shared Variable Engine and manage security and aliases.
- **NI Variable Engine** —Include this installer for applications in which you want to host, configure, or create shared variables and for applications that read and write live data using shared variables.
- **NI Web-based Configuration & Monitoring** —Include this installer if you want to use NI Web-based Configuration & Monitoring, which allows you to configure and control the Application Web Server, as well as remote devices, such as RT targets.

Installing LabVIEW-Built Installers Silently



NOTE

LabVIEW can build two types of installers: *Installer* and *Package*. You can choose to build either of them in **Build Specification**. For differences between the two types of installers, refer to Using Build Specifications.

To install a *Package*, refer to Automating an Installer in Package Manager User Manual.

To install an *Installer* silently, refer to the following content.

You can use the command line window to install LabVIEW-built installers silently. If you use the command line window to install a LabVIEW-built installer silently, consider the size of the installer. Because the installer does not display a progress bar during installation, it may be difficult to determine the progress of the installer. Therefore, a larger installer will take more time, and you will not be able to determine how much longer the installer will run.

Complete the following steps to install a LabVIEW-built installer silently.

1. **Create a LabVIEW-built installer** for the LabVIEW-built application.
2. Open the command line window.
3. Enter the following command to run the installer silently.

```
path \install.exe /q /acceptlicenses yes, where path is the path to install.exe.
```

If you have customized the installer name in the installer properties, make sure you replace the `install.exe` with your installer name.

**NOTE**

If your LabVIEW version does not support customizing the installer name, the installer name is `setup.exe`. You should use the following command: `path \setup.exe /q /acceptlicenses yes`.

**RELATED INFORMATION**

[Using Build Specifications](#) on page 390

[Automating an Installer](#)

Uninstalling LabVIEW-Built Installers

You can use the command line window to uninstall a LabVIEW-built installer and the files it installs. Complete the following steps to uninstall a LabVIEW-built installer.

1. Open the command line window and change to the directory containing the NIUninstaller, or the `uninst.exe` file. The `uninst.exe` file is in the `C:\Program Files\National Instruments\Shared\NIUninstaller` directory (Windows 64-bit) `C:\Program Files (x86)\National Instruments\Shared\NIUninstaller`.
2. Enter the following command to uninstall the installer. `uninst.exe /qb /x "product name"` where `product name` is the name of the product you want to uninstall.

**NOTE**

`product name` is case insensitive. Use the product name that appears in the Programs applet in the Control Panel to ensure the correct name. Also, enclose a product name that contains spaces in quotation marks.

You can also uninstall a LabVIEW-built installer using the Programs applet in the Control Panel. Uninstalling the installer using the Programs applet does not remove NI components that were installed with the installer. You must uninstall the NI components separately.

Using the TDM Excel Add-In (Windows)

Use the TDM Excel Add-In to load `.tdm` and `.tdms` files into Microsoft Excel. From a toolbar in Excel, choose which properties load into Excel at the file, group, and channel levels, including custom properties.

Refer to *Editing TDM(S) files with the TDM Excel Add-In* on ni.com for more information about the TDM Excel Add-In.

**RELATED INFORMATION**

[Editing TDM\(S\) files with the TDM Excel Add-In](#)

Working with the Application Manifest

Microsoft highly recommends that the *requested execution level* of an application be `asInvoker`. Therefore, `asInvoker` is the default requested execution level for all LabVIEW-built applications. However, there might be occasions where you cannot adhere to this recommendation. Complete the following steps to edit the manifest for a stand-alone application, .NET interop assembly, or shared library.

Editing the Application Manifest

1. You must verify that you have installed `mt .exe` on the computer before you can edit the manifest. Use `mt .exe` to interact with the manifest using command-line arguments. If `mt .exe` is not on the computer, download the Software Development Kit from the Microsoft website.
2. To edit the manifest, you must first retrieve it. Complete the following steps to retrieve the manifest.

1. Open the command line window.



NOTE

On most Windows operating systems, you can access the command line window by selecting **Start»Run**, entering `cmd` in the **Open** text box and clicking the **OK** button.

2. Enter the following command to retrieve the manifest: `mt .exe -inputresource: directory path \ Application Name .exe -out: Application Name .manifest` in the command line window where `directory path` is the path to the built application and `Application Name` is the name of the application.

Note If `mt .exe` is not in the same directory as the application whose manifest you want to retrieve, you must specify the path to `mt .exe` in the previous command.

3. Press the <Enter> key to run the command. The application manifest with the file extension `.manifest` appears in the same directory as the application.
3. Complete the following steps to edit the manifest.
 1. Right-click the manifest in the destination folder and open the file in a text editor or an XML editor.
 2. Locate the code `<requestedExecutionLevel level="asInvoker" uiAccess="false">`. Change the value `asInvoker` to `requireAdministrator` or `highestAvailable` depending on the execution level you want the built application to have.
 3. Save the manifest file.

Validating the Application Manifest

The tags within the *application manifest* are case sensitive. It is good practice to validate the changes you make to the manifest after you edit it. However, validating the manifest does not find spelling errors. Complete the following steps to validate the manifest.

1. In the command line window, enter the following command to validate the manifest: `mt.exe -manifest Application Name.manifest -validate_manifest` in the command line window where Application Name is the name of the application.
2. Press the <Enter> key to run the command.

Embedding the Application Manifest

After you edit and validate the application manifest, you can embed the changes when you configure the settings for a stand-alone application, .NET Framework assembly, or shared library.

1. On the **Windows Security** page of the **Properties** dialog box, place a checkmark in the **Embed Manifest File** checkbox.
2. Specify the manifest file using the Select Project File dialog box.

You also can embed the manifest without using the **Windows Security** page. Complete the following steps to embed the manifest using the command line window.

1. In the command line window, enter the command `mt.exe -manifest Application Name.manifest -outputresource: directory path \Application Name.exe; #1`, where directory path is the path to the built application and Application Name is the name of the application.
2. Press the <Enter> key to run the command.

Running Operations Using the Command Line Interface for LabVIEW

You can run *a predefined set of operations* in LabVIEW by executing commands using the command line interface (CLI) for LabVIEW. For example, use the CLI for LabVIEW to automate the build process of LabVIEW applications. You can also run *custom operations* that you create in LabVIEW.



NOTE

The CLI for LabVIEW is available as a separate package. Refer to the NI website for more information about downloading the CLI for LabVIEW and compatible versions of LabVIEW.

Before running an operation using the CLI for LabVIEW, configure LabVIEW to communicate with the CLI for LabVIEW by selecting **Tools»Options»VI Server** and placing a checkmark in the **TCP/IP** checkbox.

To run an operation using the CLI for LabVIEW, execute a command with one or more of the following arguments. Required arguments are in bold.



Note All the arguments are case-sensitive.



Tip To learn more about getting started using the CLI for LabVIEW, execute the LabVIEWCLI -Help command. To learn more about a specific operation, execute the LabVIEWCLI -OperationName < operation name > -Help command. For example, execute the LabVIEWCLI -OperationName RunVI -Help command to learn more about the [RunVI](#) operation.

Argument	Description
LabVIEWCLI -OperationName	Runs the specified operation. If the operation has additional arguments, append the arguments after the operation name.
-AdditionalOperationDirectory	Specifies the directory that stores the necessary files for the operation. This argument is required only if you run a custom operation and the necessary files for the operation are not stored in one of the following default directories: (Windows) C:\Program Files (x86)\National Instruments\Shared\LabVIEW CLI\Operations (macOS) /Library/Application Support/National Instruments/LabVIEW CLI/Operations (Linux) /usr/local/natinst/nilvcli/Operations
-LabVIEWPath	Specifies the LabVIEW version to use to run the operation. This argument is required on macOS and Linux but optional on Windows. On Windows, the default value is the version of LabVIEW that is most recently used on the machine.
-PortNumber	Specifies the port of the VI server. This argument is required if the port number under Tools»Options»VI Server»TCP/IP is not set.
-LogFilePath	Specifies the path of the CLI for LabVIEW log file. The default is the path to the temporary location.

Argument	Description
-LogToConsole	Specifies whether to save the output to both the console and file. The value for this argument must be one of the following: • true • false The default value is true.
-Verbosity	Specifies the level of details to include in the log file. The value for this argument must be one of the following: • Minimal • Default • Detailed • Diagnostic The default value is Default.

Creating Custom Command Line Operations

To create a custom command line operation to **run** in the command line interface (CLI) for LabVIEW, complete the following tasks:

- **Part 1**: Define the operation name.
- **Part 2**: Define the operation behavior.
- **Part 3** (Optional): Define the help for the operation.

Part 1: Defining the Operation Name

In the CLI for LabVIEW, command line operations have a parent class and each operation is built from a child class of the parent class. To define the name of the operation, you must create a child class of the parent class and specify the name of the child class.

Complete the following steps to define the name of the operation:

1. Navigate to one of the following directories:
 - (Windows) C:\Program Files (x86)\National Instruments\Shared\LabVIEW CLI\Operations
 - (macOS) /Library/Application Support/National Instruments/LabVIEW CLI/Operations
 - (Linux) /usr/local/natinst/nilvcli/Operations

2. Open `Operations.lvproj` or `Toolkit-Operations.lvproj`, depending on whether the operation requires the LabVIEW development system only or additional LabVIEW modules and toolkits.

**NOTE**

To open `Toolkit-Operations.lvproj`, you must install the following toolkits and library:

- LabVIEW VI Analyzer Toolkit
- LabVIEW Unit Test Framework Toolkit
- UTF JUnit Report library—Install this library using the JKI VI Package Manager (VIPM) software.

3. Right-click **My Computer** and select **New»Class**.
4. In the **New Class** window, specify the name of the class, such as `AddTwoNumbers`. The class name is the name of the operation you are creating.
5. In the project tree, right-click the class that you created and select **Properties**.
6. On the **Inheritance** page, click **Change Parent Class**.
7. Select **CoreOperation.lvclass**, click **Inherit from Selected**, and click **OK**.

Part 2: Defining the Operation Behavior

To define the behavior of the operation, you must override the methods of the class you created in the previous section and add code in one of the overridden methods.

Complete the following steps to define the behavior of the operation:

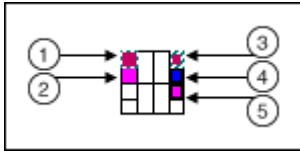
1. In the project tree, right-click the class that you created in the previous section and select **New»VI for Override**.
2. In the **New Override** window, select `GetHelp.vi` and `RunOperation.vi`, and click **OK**.
3. Save the project. When prompted, save the class and the overridden methods in a folder with the same name as the operation.

Note NI recommends that you put the folder in one of the following directories so that you do not need to specify the path to the folder using the `-AdditionalOperationDirectory` argument when running the operation.

- (Windows) `C:\Program Files (x86)\National Instruments\Shared\LabVIEW CLI\Operations`
- (macOS) `/Library/Application Support/National Instruments/LabVIEW CLI/Operations`
- (Linux) `/usr/local/natinst/nilvcli/Operations`

4. Open the block diagram of the `RunOperation.vi` method.
5. **Remove and rewire** the subVI.

6. Add code to define the behavior of the operation and save your changes. Do not modify the connector pane, as illustrated in the following figure.



Terminal	Data Type	Description
	LabVIEW object of the class that you created in the previous section.	
	Arguments that users enter in the CLI for LabVIEW when running this operation. Each string in the array represents an argument.	
	LabVIEW object of the class that you created in the previous section.	
	A code that the CLI for LabVIEW returns to the calling application, such as a Jenkins application. For example, you can configure the CLI for LabVIEW to return 0 if the operation runs successfully and -1 if an error occurs.	
	Output of the operation or error messages to display in the CLI for LabVIEW console or to save to the log file.	

Refer to `RunOperation.vi` in one of the following directories for an example of defining the behavior of the `AddTwoNumbers` operation, which adds two input numbers and returns the sum in the output.

- (Windows) `C:\Users\Public\Documents\National Instruments\LabVIEW CLI\Examples\AddTwoNumbers`
- (macOS) `/Library/Application Support/National Instruments\LabVIEW CLI\Examples\AddTwoNumbers`
- (Linux) `/usr/local/natinst/nilvcli/Examples/AddTwoNumbers`

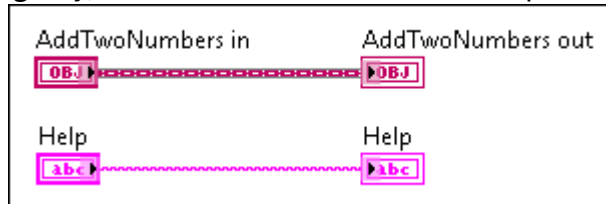
Part 3: Defining the Help for the Operation

To help users get started using the operation, you can document the operation by modifying the `GetHelp.vi` method that you created in the previous section. Users can then access the documentation by executing the `LabVIEWCLI -OperationName <operation name> -Help` command.

Complete the following steps to document the operation:

1. Open the `GetHelp.vi` method from the project tree.
2. On the front panel, create a string control named **Help** and enter the documentation in this control.

3. Right-click the **Help** control and select **Data Operation»Make Current Value Default** to set the current text as the default value of this control. Repeat this step each time you edit the text.
4. On the block diagram, *remove and rewire* the subVI.
5. Wire the objects in the following way, where AddTwoNumbers is the operation



name, and save your changes.

Changing the Default Ports for TCP-Based NI-PSP (NI Linux Real-Time, Windows)

Complete the following steps to create the LogosXT.ini file and use it to modify TCP ports [NI Publish-Subscribe Protocol \(NI-PSP\)](#) checks for availability.

1. Create a new text document in the appropriate directory for your operating system:
 Windows Directory
 1. Configure Windows to show all hidden folders.
 2. Navigate to the \National Instruments\Logos XT subfolder in the common application directory. The location of this directory depends on how you configured the installation of LabVIEW. The default location is C:\ProgramData\National Instruments\Logos XT. Create this subfolder if it does not exist.
 3. Create a new text document.
 4. Name the new text document LogosXT.ini and place it in the Logos XT subfolder.

NI Linux Real-Time Directory

1. Log in to the RT target with administrator privileges.
2. Navigate to the /etc/natinst/ directory.
3. Create a new text document.
4. Name the new text document LogosXT.ini



NOTE

The file system is case-sensitive.

2. Open LogosXT.ini and enter the following text. [LogosXT] LogosXT_PortBase = 59110
LogosXT_NumPortsToCheck = 100

Note LogosXT_PortBase specifies the port where the NI-PSP protocol begins to check availability. LogosXT_NumPortsToCheck specifies the maximum number of ports the protocol checks. By default, NI-PSP begins checking port availability at port 59110 and checks a maximum of 100 ports.

3. To modify the first port the NI-PSP protocol checks for availability, delete 59110 and enter a different port number for LogosXT_PortBase. To modify the maximum number of ports the NI-PSP protocol checks for availability, delete 100 and enter a different number for LogosXT_NumPortsToCheck.
4. Save and close LogosXT.ini. The changes take effect when you restart LabVIEW.

JSONP Support

To use JSONP in order to avoid the same-origin policy in most browsers, make any request as normal and add an extra \$callback parameter. For example, `GET http://my_server:3580/nivariable/VariableObjects('ni.var.psp://localhost')/type?$format=json&$callback=my_callback`

The JSON data is passed as a parameter to a function called `my_callback`, which you supplied with the `$callback` parameter. JSONP support is only valid when JSON format is requested. JSONP support is ignored if XML format is requested.

Waveform Representations

Waveforms in LabVIEW are clusters of four components: `t0`, `dt`, `Y`, and `attributes`. In JSON, waveforms are represented as the string value of the JSON object representation. In XML, LabVIEW flattens the values to an XML primitive VI. The "<" and ">" are escaped with "<" and ">," and stored as the string value of the `Value` property.



NOTE

JSON representation always includes a default error constant.

Using Buffering with DataSocket Functions

Complete the following steps to enable client-side buffering using DataSocket functions. You must complete these steps for each client connection you establish.

1. Open a data connection.
2. Specify the buffer size for the data client.
3. Read the buffered data a server writes.
4. Monitor the buffer size.
5. Close the data connection.
6. Wire all error in and error out terminals on the block diagram.

You also can enable buffering when sharing data using the DataSocket functions by adding the string `?maxbytes=1000&maxpackets=10`, to the end of the URL, where 1000 is

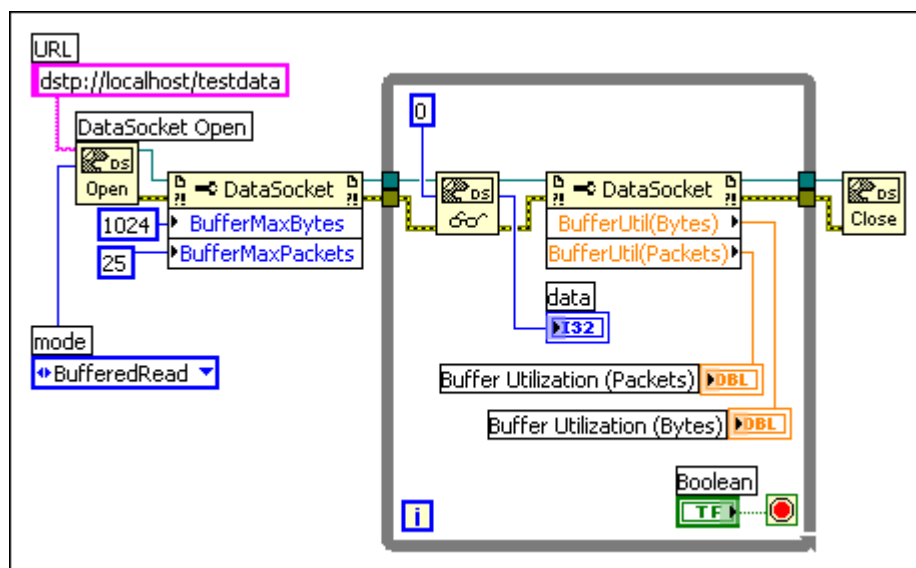
the maximum number of bytes the buffer contains and 10 is the maximum number of packets the buffer contains.



NOTE

Buffering applies only when you use the DataSocket Read function to read data a server writes. Buffering is not available when you use front panel DataSocket data binding to read data a server writes. Buffering is available for front panel data binding through the Shared Variable Engine if you bind controls to shared variables with buffering enabled on the Network page of the **Shared Variable Properties** dialog box.

The following example uses buffering with DataSocket functions.



RELATED INFORMATION

[Buffering Data](#) on page 646

[Programmatically Opening and Closing Data Connections](#) on page 1529

[Specifying the Buffer Size Using DataSocket Functions](#) on page 1530

[Reading the Buffered Data a Server Writes](#) on page 1530

[Monitoring the Buffer Size of a Data Connection](#) on page 1531

[Programmatically Opening and Closing Data Connections](#) on page 1529

[Setting up Error I/O in a VI](#) on page 1436

[Sharing Live Data Programmatically](#) on page 641

[Reading Live Data Programmatically](#) on page 1532

[Reading Front Panel Data Using the dstp, opc, ftp, http, or file Protocol](#) on page 1535

[Reading and Writing Shared Variables with Controls and Indicators](#) on page 633

[Understanding Shared Variable Technology](#) on page 610

Programmatically Opening and Closing Data Connections

Use the DataSocket Open and DataSocket Close functions to open and close data connections programmatically.

Complete the following steps to open and close a data connection programmatically.

1. Add the DataSocket Open function to the block diagram.
2. Wire a psp **URL** to the **URL** input or wire a **shared variable control** to the **URL** input.
3. Right-click the **mode** input and select **Create»Constant** from the shortcut menu.
4. Click the **mode** constant and select **Read** (0), **Write** (1), **ReadWrite** (2), **BufferedRead** (3), or **BufferedReadWrite** (4). You also can use the DataSocket:Mode property to set the data connection mode programmatically.
5. Add one of the following functions to the block diagram and create controls or constants for the appropriate inputs:
 - DataSocket Read function
 - DataSocket Write function
6. Wire the **connection id** output of the DataSocket Open function to the **connection in** input of the DataSocket Read or DataSocket Write function.
7. Add a DataSocket Close function to the block diagram.
8. Wire the **connection out** output of the DataSocket Read or DataSocket Write function to the **connection id** input of the DataSocket Close function to close the data connection. The connection also closes when the VI stops.
9. **Wire all error in and error out terminals** on the block diagram.

Writing Live Data Programmatically

Write live data programmatically to send live data updates from a VI that is not in a project, to control and view properties of the data connection, to send live data updates to other VIs without displaying the data on the front panel of the VI, or to **avoid data loss** that might occur if you write data from the front panel.

(Real-Time, Windows) National Instruments recommends that you **use a Shared Variable node** to write live data on a block diagram.

Complete the following steps to write live data programmatically.

1. Add the DataSocket Write function to the block diagram.
2. Specify a psp **URL** or wire a **shared variable control** to the **connection in** input of the DataSocket Write function.
3. Wire the **data** you want to write to the **data** input of the DataSocket Write function. If another application reads the live data, **convert the data** to variant data.

4. Run the VI or continually run the VI to write live data updates to the data connection.

Specifying the Buffer Size Using DataSocket Functions

Complete the following steps to specify the client-side buffer size using DataSocket functions.

1. Add the DataSocket Open function to the block diagram.
2. Wire a [URL](#) to the **URL** input or wire a [shared variable control](#) to the **URL** input.
3. Right-click the **mode** input and select **Create»Constant** from the shortcut menu.
4. Click the **mode** constant and select **BufferedRead**.
5. Add a Property Node to the block diagram.
6. Right-click the Property Node and select **Select Class»DataSocket»DataSocket** from the shortcut menu.
7. [Resize the Property Node](#) and add a terminal.
8. Click the first white area of the Property Node and select the Buffer Maximum Bytes property.
9. Click the second white area of the Property Node and select the Buffer Maximum Packets property.
10. Right-click the Property Node and select **Change All To Write** from the shortcut menu.
11. Create constants for each property and enter values for each property, such as 1024 for Buffer Maximum Bytes and 25 for Buffer Maximum Packets. LabVIEW stops filling the client-side buffer when the first buffer size limit is reached for either the Buffer Maximum Bytes property or the Buffer Maximum Packets property.
12. Wire the **connection id** output of the DataSocket Open function to the **reference** input of the Property Node.

After you specify the buffer size on the client, you can [read the buffered data](#).

Reading the Buffered Data a Server Writes

After you specify a buffer size using DataSocket functions, you can read the buffered data.

Complete the following steps to read the buffered data a server writes.

1. Add a While Loop to the block diagram.
2. Add the DataSocket Read function in the While Loop.
3. Wire a control or constant to the **type** input of the DataSocket Read function. The data type of the control or constant must match the data type of the data you expect to read. The default data type is the variant data type. If you are unsure of the data type of the data you expect to read or if another application generates the

data, use the variant data type. You can convert variant data to data that LabVIEW can manipulate.

4. Wire the **connection id** output of the Property Node you created when you specified the buffer size on the client to the **connection in** input of the DataSocket Read function.

After you read the buffered data, you can monitor the size of the buffer.



RELATED INFORMATION

[Specifying the Buffer Size Using DataSocket Functions](#) on page 1530

[Sharing Live Data Programmatically](#) on page 641

[Converting Variant Data to LabVIEW Data](#) on page 1172

[Specifying the Buffer Size Using DataSocket Functions](#) on page 1530

[Monitoring the Buffer Size of a Data Connection](#) on page 1531

Monitoring the Buffer Size of a Data Connection

After you read buffered data, you can monitor the buffer size and request diagnostic information about the buffers you specified.

Complete the following steps to monitor the buffer size.

1. Add a While Loop to the block diagram.
2. Add a DataSocket Open function to the left of the While Loop to open a data connection programmatically.
3. Specify the buffer size for the client connection.
4. Add a DataSocket Read function inside the While Loop.
5. Wire the **connection id** output of the DataSocket Open function to the **connection in** input of the DataSocket Read function.
6. Add a Property Node inside the While Loop to the right of the DataSocket Read function.
7. Wire the **connection out** output of the DataSocket Read function to the **reference** input of the Property Node.
8. Resize the Property Node and add a terminal.
9. Click the first white area of the Property Node and select the Buffer Utilization (Bytes) property.
10. Click the second white area of the Property Node and select the Buffer Utilization (Packets) property.
11. Create indicators for each property to monitor the use of the buffer constraints you specified.

LabVIEW stops filling the client-side buffer when the first buffer size limit is reached for either the Buffer Maximum Bytes property or the Buffer Maximum Packets property.



RELATED INFORMATION

[Reading the Buffered Data a Server Writes](#) on page 1530

[Programmatically Opening and Closing Data Connections](#) on page 1529

[Specifying the Buffer Size Using DataSocket Functions](#) on page 1530

[Resizing Nodes](#) on page 891

Reading Live Data Programmatically

Read live [data](#) from a data connection to manipulate the data before you add it to the front panel or pass the data to a VI or function on the block diagram.

(Real-Time, Windows) National Instruments recommends that you [use a Shared Variable node](#) to read live data on a block diagram.

Complete the following steps to read live data programmatically.

1. Add the DataSocket Read function to the block diagram.
2. Specify a [pspURL](#) in the **connection in** input of the DataSocket Read function or wire a [shared variable control](#) to the **connection in** input.
3. Wire a control or constant to the **type** input of the DataSocket Read function. The data type of the control or constant must match the data type of the data you expect to read. The default data type is the variant data type. If you are unsure of the data type of the data you expect to read or if another application is generating the data, use the variant data type. You can [convert variant data](#) to data that LabVIEW can manipulate.
4. Run the VI or continually run the VI for live data updates.

Controlling a Front Panel Control Remotely Using the dstp, opc, ftp, http, or file Protocol

Using [front panel data binding](#) to control a front panel control remotely through the [Shared Variable Engine](#) provides faster and more reliable data transmission than [dstp](#) as well as seamless connections to shared variables and [NI Publish-Subscribe Protocol \(NI-PSP\)](#) data items.

Complete the following steps to control a front panel control from another VI using the `opc`, `ftp`, or `http` protocols on Windows, and the `dstp` and `file` protocols on all LabVIEW-supported platforms.



NOTE

You must install the DataSocket Server to control front panel controls remotely using the `dstp` protocol.

1. If you are using a `dstp` connection, launch and configure the DataSocket Server. If you are using an `opc`, `ftp`, `http`, or `file` connection, skip to step 2.
 1. Open the DataSocket Server by selecting **Start»All Programs»National Instruments»DataSocket»DataSocket Server**.
 2. Launch the DataSocket Server Manager by selecting **Start»All Programs»National Instruments»DataSocket»DataSocket Server Manager**.
 3. Click the **New Item** button to create a predefined data item, and place a checkmark in the **Allow Multiple Writers** checkbox. Refer to the NI DataSocket Server Help for more information about configuring a connection for multiple writers.
2. Right-click a front panel object and select **Properties** from the shortcut menu to display the **Properties** dialog box for the object.
3. Click the right arrow button to navigate to the **Data Binding** page of the **Properties** dialog box.
4. Select **DataSocket** from the **Data Binding Selection** pull-down menu.
5. On Windows, enter an `opc`, `ftp`, or `http` [URL](#), or on all LabVIEW-supported platforms enter a `file` or `dstp` [URL](#) in the **Path** text box. You also can select **DSTP Server** or **File System** on all LabVIEW-supported platforms from the **Browse** pull-down menu to navigate to a URL for the predetermined data item you created in step 1c.
6. On the **Data Binding** page, select **Read/Write** from the **Access Type** pull-down menu.
7. Click the **OK** button.
8. Run the VI.
9. In a second VI (either on the local computer or on a remote computer), right-click a front panel object and select **Properties** from the shortcut menu to display the **Properties** dialog box for the object.
10. Click the right arrow button to navigate to the **Data Binding** page of the **Properties** dialog box.
11. Select **DataSocket** from the **Data Binding Selection** list.
12. Select **DSTP Server** or **File System** from the **Browse** pull-down menu to navigate to the URL you entered in step 5.
13. On the **Data Binding** page, select **Read/Write** from the **Access Type** pull-down menu.
14. Run or continually run both VIs. While the VI is running, if the connection to the DataSocket Server is successful, a small green indicator appears next to the front panel object on the VI. If the connection is not successful, a small red indicator

appears. Before LabVIEW attempts to connect to the server, a small transparent indicator appears. A status message also appears if you move the cursor over the indicator. You can hide this indicator by right-clicking the front panel object and selecting **Visible Items»DataSocket LED** from the shortcut menu. You also can use the DataSocket:LED Visible property to hide the indicator programmatically.

The front panel control in the first VI retrieves the value the second VI passes through the DataSocket connection. When a user changes the control value in the second VI, the DataSocket connection writes the new value to the front panel control in the first VI. If the front panel control has write-only access, the control writes the initial value when the VI runs or every time the VI resets if the VI runs continuously. If the front panel control has read-only or read/write access, the control reads the initial value when the VI runs.

Enabling the dstp, opc, ftp, http, or file Connection for a Control

Using *front panel data binding* to control a front panel control remotely through the *Shared Variable Engine* provides faster and more reliable data transmission than dstp as well as seamless connections to *shared variables* and *NI Publish-Subscribe Protocol (NI-PSP)* data items.

Enabling an opc, ftp, or http connection for a control on Windows, or a dstp or file connection for a control on all LabVIEW-supported platforms lets a control write and read live *data* through the DataSocket Server, and on Windows, through an OPC server, FTP server, or Web server. Do not enable a connection for a control if you want to establish or save the URL for the object but not write or read data.

Complete the following steps to enable a control for an opc, ftp, or http connection on Windows, or a dstp or file connection on all LabVIEW-supported platforms.

1. If you are using a dstp connection, launch and configure the DataSocket Server. If you are using an opc, ftp, http, or file connection, skip to step 2. Open the DataSocket Server by selecting **Start»All Programs»National Instruments»DataSocket»DataSocket Server**.
2. Right-click a front panel object and select **Properties** from the shortcut menu to display the **Properties** dialog box for the object.



NOTE

You cannot create front panel data connections for *type definitions*.

1. Click the right arrow button to navigate to the **Data Binding** page of the **Properties** dialog box.
2. Select **DataSocket** from the **Data Binding Selection** pull-down menu.
3. On Windows, enter an opc, ftp, or http *URL*, or on all LabVIEW-supported platforms enter a file or dstp *URL* in the **Path** text box. You also can select **DSTP Server** or **File System** on all LabVIEW-supported platforms from the **Browse** pull-down menu to navigate to a URL.

4. Select **Read/Write** from the **Access Type** pull-down menu.
5. Click the **OK** button.

Reading Front Panel Data Using the dstp, opc, ftp, http, or file Protocol

Using *front panel data binding* to control a front panel control remotely through the *Shared Variable Engine* provides faster and more reliable data transmission than dstp as well as seamless connections to *shared variables* and *NI Publish-Subscribe Protocol (NI-PSP)* data items.

Complete the following steps to read front panel *data* using the opc, ftp, or http protocols on Windows, and the dstp and file protocols on all LabVIEW-supported platforms.



NOTE

You must install the DataSocket Server to read dstp front panel data.

1. If you are using a dstp connection, launch and configure the DataSocket Server. If you are using an opc, ftp, http, or file connection, skip to step 2. Open the DataSocket Server by selecting **Start»All Programs»National Instruments»DataSocket»DataSocket Server**.
2. Right-click a front panel object and select **Properties** from the shortcut menu to display the **Properties** dialog box for the object.



Note You cannot create front panel data connections for *type definitions*.

1. Click the **Data Binding** tab of the **Properties** dialog box.
2. Select **DataSocket** from the **Data Binding Selection** pull-down menu.
3. On Windows, enter an opc, ftp, or http *URL*, or on all LabVIEW-supported platforms enter a file or dstp *URL* in the **Path** text box. You also can select **DSTP Server** or **File System** on all LabVIEW-supported platforms from the **Browse** pull-down menu to navigate to a URL.
4. Select **Read only** from the **Access Type** pull-down menu.
5. Click the **OK** button.
6. On the block diagram, *create a While Loop* to configure the VI to run continuously.
7. Run the VI. LabVIEW retrieves the data from the DataSocket Server and displays it in the front panel object of the VI. While the VI is running, if the connection to the DataSocket Server is successful, a small green indicator appears next to the front panel object on the VI. If the connection is not successful, a small red indicator appears. Before LabVIEW attempts to connect to the server, a small transparent indicator appears. A status message also appears if you move the cursor over the indicator. You can hide this indicator by right-clicking the front panel object and

selecting **Visible Items»DataSocket LED** from the shortcut menu when the VI is not running. You also can use the LED Visible property to hide the indicator programmatically.

Writing Front Panel Data Using the dstp, opc, ftp, http, or file Protocol

Using *front panel data binding* to control a front panel control remotely through the *Shared Variable Engine* provides faster and more reliable data transmission than dstp as well as seamless connections to *shared variables* and *NI Publish-Subscribe Protocol (NI-PSP)* data items.

Complete the following steps to write front panel *data* using the opc, ftp, or http protocols on Windows, and the dstp and file protocols on all LabVIEW-supported platforms.



NOTE

You must install the DataSocket Server to write front panel data using the dstp protocol. Writing data from the front panel might result in data loss. If you want to write all front panel data instead of only the most recent data, use the DataSocket Write function to *write live data programmatically*.

1. If you are using a dstp connection, launch and configure the DataSocket Server. If you are using an opc, ftp, http, or file connection, skip to step 2. Launch the DataSocket Server from the National Instruments directory.
2. Right-click a front panel object and select **Properties** from the shortcut menu to display the *Properties* dialog box for the object.
3. On the **Data Binding** page of the **Properties** dialog box, select **DataSocket** from the **Data Binding Selection** pull-down menu.
4. On Windows, enter an opc, ftp, or http *URL*, or on all LabVIEW-supported platforms enter a file or dstp *URL* in the **Path** text box. You also can select **DSTP Server** or **File System** on all LabVIEW-supported platforms from the **Browse** pull-down menu to navigate to a URL.
5. Select **Read/Write** from the **Access Type** pull-down menu.
6. Click the **OK** button.
7. On the block diagram, *create a While Loop* to configure the VI to run continuously.
8. Run the VI. LabVIEW writes the data from the front panel object to the server destination you entered in the URL. While the VI is running, if the connection to the DataSocket Server is successful, a small green indicator appears next to the front panel object on the VI. If the connection is not successful, a small red indicator appears. Before LabVIEW attempts to connect to the server, a small transparent indicator appears. A status message also appears if you move the cursor over the indicator. You can hide this indicator by right-clicking the front panel object and selecting **Visible Items»DataSocket LED** from the shortcut menu when the VI is not running. You also can use the DataSocket:LED Visible property to hide the indicator programmatically.

Web Service

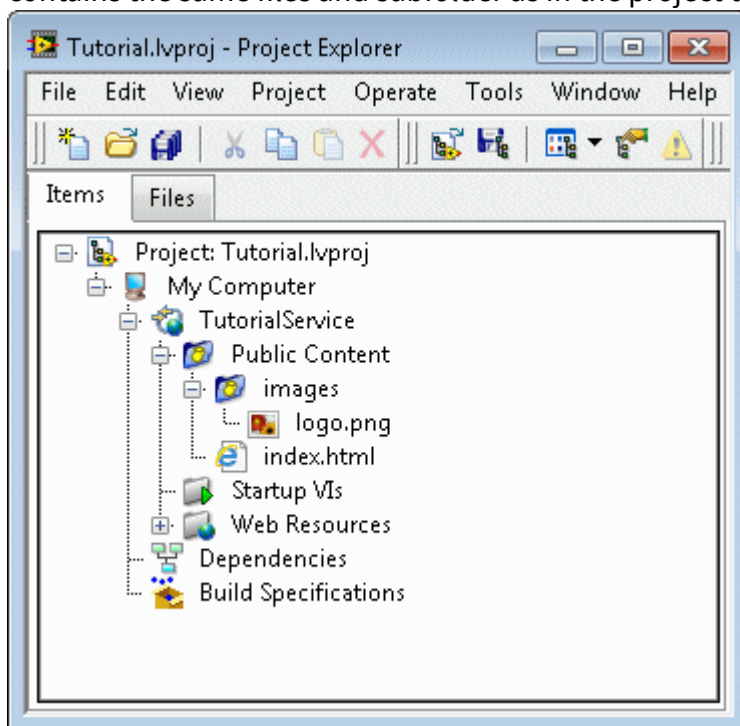
Integrating Static Content into a Web Service (Real-Time, Windows)

Parent Topic: *Developing Web Services*

When you build a Web service, you can integrate static content such as HTML files, JavaScript files, CSS files, images, and videos from within the LabVIEW project. For example, you can publish stand-alone web pages using Web services or provide a user interface that interacts with the HTTP method VIs in a Web service.

Complete the following steps to integrate static content into a Web service:

1. Organize static content on disk in two main folders: one that contains public content and one that contains private content you do not want to expose to users.
2. Open the LabVIEW project that contains the Web service files.
3. Right-click the Web service project item  in the project tree and select **Add Public Content Folder** or **Add Private Content Folder**.
4. In the browse dialog box that appears, select the folder on disk that contains files you want to include in the Web service.
5. LabVIEW creates an auto-populating folder named **Public Content** or **Private Content** under the Web service project item. Any files in the folder on disk appear under the folder that you added to the project tree. The following project shows a project that contains a **Public Content** folder. The corresponding folder on disk contains the same files and subfolder as in the project tree.



After you publish the Web service, you can access the static files from a client, such as a web browser, by navigating to the URL mapping for the file you want to access.

Finding the URL Mapping for a Public File

LabVIEW automatically assigns a URL mapping to each public static file. You cannot change this mapping. Complete one of the following steps to view the URL mapping for a public static file:

- For a specific static file or folder, right-click the file in the project tree and select **Show Public URL**.
- To view URL mappings for all public static files and folders, right-click the Web service project item and select **Properties** to display the **Web Service Properties** dialog box. On the **Site Map** page, the table contains a row with the URL mapping for each public static file and folder.

Integrating Default HTML Files

You can include an `index.html` file within each public content folder and subfolder in the project. When a client sends an HTTP request using a URL such as `http://localhost/TutorialService/`, the Web service returns the `index.html` file.

Complete the following steps to integrate a default `index.html` file for a specific public content folder:

1. On disk, add a file named `index.html` to a folder that appears under the **Public Content** folder in the project. In the previous example, you place the file in the folder on disk that you associated with the top-level **Public Content** folder. The file automatically appears in the project tree unless you stop the folder from auto-populating changes in the corresponding folder on disk.

Note The file extension must be `.html`.

2. Right-click the Web service project item and select **Properties**.
3. On the Service Settings page, place a checkmark in the **Serve default index.html for public folders** checkbox, and click **OK**.
4. In a Web browser, navigate to the URL that corresponds to the folder with the `index.html` file. The HTML file loads in the browser.

Integrating Private Content in Web Services

Private static files do not have URL mappings because web clients cannot access private content. To programmatically access private static content within Web service source files, use the Read Service Attribute VI to return the path to the folder that contains private content at run time. With the path to the private content folder, you can build a path to a specific file. Refer to the Read Service Attribute VI help for more information.

Related Information

Components of a Web Service: Static Files

Testing and Debugging a Web Service on the NI Web Server (Windows)

Parent Topic : *Developing Web Services*

Before you publish a Web service to a target, you might want to test that clients can invoke and communicate with HTTP method VIs and static content as expected. If you notice unexpected behavior or errors, you can correct them and then test the Web service again.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Workflow for Debugging a Web Service

Testing and debugging a Web service is typically an iterative process you complete in the LabVIEW project that contains the Web service files. Complete the following steps to test and debug a Web service on the NI Web Server:



Note In this topic, you test and debug a Web service on the NI Web Server. You can also test your Web service on the Application Web Server, refer to [Testing and Debugging a Web Service on the Application Web Server](#).

1. If the Web service is part of a stand-alone application, open and run any startup VIs for the application from the project.
2. Right-click the Web service project item  in the project tree and select **Start**. LabVIEW places the Web service on the NI Web Server, where it behaves as if published to a target.
3. If you are using the NI Web Server for the first time on your computer, complete the following steps to enable and configure the NI Web Server:
 1. Click **Configure NI Web Server** in the **Start Web Service** dialog box to launch NI Web Server Configuration.
 2. For testing and debugging, choose the **Simple local access** configuration preset and click **Next**.
 3. On the **Authentication** page, select **Log in using an admin user for server administration** and create your own password. Click **Next**.
 4. Click **Finish** to apply this configuration.
 5. Switch back to the LabVIEW project and repeat step 2.

4. Open HTTP method VIs from the project. Use standard LabVIEW debugging techniques, such as probes, breakpoints, and execution highlighting, to observe their behavior when the client invokes them.
5. Invoke an HTTP method VI from a client, such as a web browser, using the URL mapped to the VI. You can find and copy the appropriate URL directly from the project. Complete the following steps to access an HTTP Method VI:
 1. Right-click the HTTP method VI and select **Show Method URL** to display the HTTP Method URL dialog box.
 2. In the **Available Servers** pull-down menu, select the item that contains **NI Web Server**, and then click **Copy URL**.
 3. Close the dialog box.

Note The Web service does not respond to any requests from the client if the **HTTP Method URL** dialog box is open.

4. Paste the URL into a web browser and replace the {value} variables with numeric values, such as `http://127.0.0.1:80/TutorialService/Add?b=2&a=3`.

Note NI Web Server uses 80/443 as the default port in accordance with HTTP/HTTPS protocol. However, the port number might change if another application occupies the port. You can customize the port number in NI Web Server Configuration. Refer to *Configuring NI Web Server* in the NI Web Server Manual for more information on server settings like authentication, port, and remote connections.

5. Browse to the URL.
6. If you notice an error or finish testing the Web service, right-click the Web service project item in the project tree and select **Stop**. If startup VIs for a LabVIEW stand-alone application are running, you do not have to stop them.

Note You cannot edit HTTP method VIs until you stop the debugging session because LabVIEW reserves them for execution during debugging.

7. Edit the VIs to correct any issues you noticed.
8. (Optional) Repeat steps 2-7 to validate your changes to the Web service files.

Related Information

[Understanding the NI Web Server](#)

Setting Up the NI Web Server for Web Services (Windows)

Parent Topic: [Hosting Web Services](#)

To host your Web services on the NI Web Server, set up the web server before you run or publish the Web services.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Complete the following steps to enable and configure the [NI Web Server](#) for hosting Web services. If you use the Application Web Server to host Web services, refer to [Setting up the Application Web Server for Web Services](#) for detailed instructions.

1. Launch the NI Web Server Configuration:
 1. In the **Project Explorer** window, select **Tools»Options** to display the Options dialog box.
 2. On the **Web Server** page, click **Configure NI Web Server**.

Note LabVIEW launches NI Web Server Configuration when you first use the NI Web Server to host Web services.

2. The first time you launch NI Web Server Configuration, the **NI Web Server Guided Setup** appears.
3. Choose a configuration preset according to your needs or select **Help me choose a preset**. Click **Next**.
4. Follow the prompts to choose among configuration options. Click **Finish**.
5. The NI Web Server starts running after you configure it.
6. On the **Control** page, check the server status. You can restart, stop, enable, or disable the NI Web Server on this page.

Refer to [Configuring NI Web Server](#) for more information on server settings like authentication, port, and remote connections.

Configuring Web Services Security when Using the NI Web Server (Windows)

You can use HTTPS and privilege-based authorization to establish secure communication between Web clients and LabVIEW Web services.



NOTE

In this topic, you configure the security settings for the NI Web Server. If you use the Application Web Server to host Web services, refer to [Configuring Web Services Security when Using the Application Web Server](#) for detailed instructions.

Complete the following procedures to establish the different methods of Web services security:

- [Enabling HTTPS on the NI Web Server](#)
- [Assigning User Roles and Privileges](#)

Enabling HTTPS on the NI Web Server

You can enable Secure Sockets Layer (SSL) encryption on the NI Web Server for Web services. Use SSL encryption to create secure, encrypted connections when sending data between clients and the web server.

Complete the following steps to enable HTTPS connections:

1. Launch NI Web Server Configuration using the following steps:
 1. In the **Project Explorer** window, select **Tools»Options** to display the Options dialog box.
 2. On the **Web Server** page, click **Configure NI Web Server**.
2. On the **HTTPS** tab in NI Web Server Configuration, choose one of the **HTTPS settings**. Refer to Choosing an HTTPS Setting in the NI Web Server Manual for more information about each HTTPS setting.
3. Follow the instructions to generate a self-signed certificate or to install a certificate from a certification authority.
4. Configure the **Server HTTPS port** number.
5. Click **Apply and Restart**. The NI Web Server uses HTTPS to communicate with clients after it restarts.



Note LabVIEW automatically updates the HTTP Method URL according to the changes in the settings for HTTP connection and port of the NI Web Server.

Assigning User Roles and Privileges

You can define user roles and assign different privileges to each role so that you can control each client's access to each HTTP Method VI and NI Web Server applications.

Defining User Roles and Privileges for NI Web Server Application

Refer to Defining User Roles and Privileges in the NI Web Server Manual for information on defining user roles and assign privileges for NI Web Server applications in NI Web Server Configuration.

Assigning User Roles and Privileges to LabVIEW Web Services

Complete the following steps in LabVIEW to configure an HTTP Method VI to accept only requests from Web clients with valid privileges:

1. Right-click the Web service project item  in the project tree and select **Properties** to display the Web Service Properties dialog box.
2. On the NI Web Server Privileges page, customize and enter a name for the new privilege under the **Privilege** column to create a privilege.
3. Under the **Role** column, select a user role or enter a customized role that you create in NI Web Server Configuration to assign the privilege to the user role you specify. For example, you can assign writedata privilege to only admin user role.
4. On the HTTP Method VI Settings page, select a VI from the **Web Service VI** table and click the **NI Web Server** tab in **Web Service VI Properties**.
5. Select a privilege from the **Required Privilege** pull-down menu.
6. Click **OK** to apply the changes.

Accessing HTTP Method VIs with Privileges

A Web client must provide a valid username and password whenever the client submits an HTTP request for a VI protected by privileges. Refer to Choosing an Authentication Setting in the NI Web Server Manual and configure settings on the **Authentication** tab in NI Web Server Configuration to group users into different roles.

Publishing a Web Service through a Package (Windows)

Parent Topic : [Publishing Web Services](#)

You can publish a Web service to the NI Web Server through an [NI package](#) .nipkg. The Web service starts/stops running after you install/remove the package.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Complete the following steps to publish a Web service through an NI package:

1. Include the Web service in a package:
 1. Right-click **Build Specifications** in the Project Explorer window and select **New » Package** to launch the Package Properties dialog box.
 2. On the **Web Services** page, place a checkmark in the checkbox for the Web service you want to include.
 3. (Optional) On the **Package Installer** page, place a checkmark in the **Create a Package Installer** checkbox to build your distribution into a package installer file .exe.
 4. (Optional) On the **Feed** page, choose whether to add your package to a local feed or publish the package to a feed on a SystemLink server.
2. Build the package that contains the Web service:

1. Click **Build** and the Build Status dialog box appears.
2. When LabVIEW completes building the package, click **Explore** to open the destination directory where LabVIEW saves the package file(s).
3. Install the package on the target computer so that the Web service starts running on the target computer:
 1. Double-click the package .nipkg in the destination directory and install the package through NI Package Manager. When the installation is complete, the Web service is published to the NI Web Server on the local computer.

Note Publishing Web services to the NI Web Server requires administrator privilege.

2. Close the LabVIEW project. Published Web services run on the LabVIEW Runtime and do not depend on the LabVIEW Development System.

When Do I Create a Package Installer?

A package installer includes all package dependencies so that the target can install the package without network access or any NI software. If you want to publish the Web service to a computer that does not have an internet connection or any necessary software for installing the package and running the Web service, such as NI Package Manager and the LabVIEW Runtime, you need to create a package installer.

When Do I Add the Web Service to a Feed?

On the deployment target, you can subscribe to a feed to find and install the package and receive update notifications from NI Package Manager or NI SystemLink via network access. NI recommends using feeds when distributing multiple packages that have dependencies on one another.

Related Information

Installing Packages from a Feed

Deploying Applications to Clients on a Server

Publishing a Web Service through a Stand-Alone Application (Real-Time, Windows)

Parent Topic : [Publishing Web Services](#)

You can publish a Web service to the NI Web Server or Application Web Server through a stand-alone application. The Web service starts/stops running when the application starts/stops running.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Complete the following steps to publish a Web service through a stand-alone application:

1. Include the Web service in an application:
 1. (Windows) Right-click **Build Specifications** in the Project Explorer window and select **New » Application** to launch the Application Properties dialog box. (Real-Time) Right-click **Build Specifications** in **Project Explorer** and select **New » Real-Time Application** to launch the Real-Time Application Properties dialog box.
 2. On the **Web Services** page, place a checkmark in the checkbox for the Web service you want to include.
 3. Choose a web server for hosting the Web service. Refer to [Hosting Web Services](#) for more information on each web server.

Note (Real-Time) You can use only the Application Web Server to host Web services on an RT target because the NI Web Server does not support RT targets.

4. (Optional) If you choose **Application Web Server** in step c, configure which port you want to use to accept requests for the Web service. You must configure each application that runs simultaneously to use a unique port. Place a checkmark in the **Enable SSL** checkbox and configure the other items if you want to allow clients to use HTTPS protocol to call the Web service.
2. Build the application that contains the Web service:
 1. Click **Build** and the Build Status dialog box appears.
 2. When LabVIEW completes building the application, click **Explore** to open the destination directory where LabVIEW saves the application files.
3. Run the application on the target device so that the Web service starts running on the device:
 1. (Windows) Double-click the executable in the destination directory to run it. (Real-Time) Right-click the application you build under **Build Specifications** in **Project Explorer**, and click **Run as startup**. The application publishes the Web service to the web server on the target device.

Note Publishing Web services to the NI Web Server requires administrator privilege.

2. Close the LabVIEW project. Published Web services run on the LabVIEW Runtime and do not depend on the LabVIEW Development System.

Publishing a Web Service through an Installer (Windows)

Parent Topic : [Publishing Web Services](#)

You can publish a Web service to the Application Web Server through a Windows installer. The Web service starts/stops running after you finish the installation/removal process.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Complete the following steps to publish a Web service through a Windows installer:

1. Include the Web service in an installer:
 1. Right-click **Build Specifications** in the Project Explorer window and select **New » Installer** to launch the Installer Properties dialog box.
 2. On the **Web Services** page, place a checkmark in the checkbox for the Web service you want to include.
2. Build the installer that contains the Web service:
 1. Click **Build** and the Build Status dialog box appears.
 2. When LabVIEW completes building the installer, click **Explore** to open the destination directory where LabVIEW saves the installer files.
3. Run the installer on the target computer so that the Web service starts running on the target computer:
 1. Double-click the installer .exe in the destination directory to install the application, and the installed application publishes the Web service to the web server on the local computer.
 2. Close the LabVIEW project. Published Web services run on the LabVIEW Runtime and do not depend on the LabVIEW Development System.

Publishing and Unpublishing a Stand-Alone Web Service (Real-Time, Windows)

Parent Topic : [Publishing Web Services](#)

You can publish or unpublish a stand-alone Web service to the Application Web Server directly from a LabVIEW project.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

To publish a stand-alone Web service to either the computer where you develop the Web service or a connected RT target, right-click the Web service project item in the **Project Explorer** window and select **Application Web Server » Publish**. Published Web services run on the LabVIEW Runtime and do not depend on the LabVIEW Development System.

To unpublish a stand-alone Web service, right-click the Web service project item and select **Application Web Server » Unpublish**.

Testing and Debugging a Web Service on the Application Web Server (Real-Time, Windows)

Before you publish a Web service to a target, you might want to test that clients can invoke and communicate with HTTP method VIs and static content as expected. If you notice unexpected behavior or errors, you can correct them and then test the Web service again.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Workflow for Debugging a Web Service

Testing and debugging a Web service is typically an iterative process you complete in the LabVIEW project that contains the Web service files. Complete the following steps to test and debug a Web service on the Application Web Server:



Note In this topic, you test and debug a Web service on the Application Web Server. You can also test your Web service on the NI Web Server, refer to [Testing and Debugging a Web Service on the NI Web Server](#).

1. If the Web service is part of a stand-alone application, open and run any startup VIs for the application from the project.
2. (Windows) Right-click the Web service project item  under **My Computer** in the project tree and select **Application Web Server»Start (Debug Server)**. (Real-Time)
Right-click the Web service project item  the RT target in the project tree and select **Start (Debug Server)**. LabVIEW places the Web service on a debugging server, where it behaves as if published on a target.
3. Open HTTP method VIs from the project. Use standard LabVIEW debugging techniques, such as probes, breakpoints, and execution highlighting, to observe their behavior when the client invokes them.
4. Invoke an HTTP method VI from a client, such as a web browser, using the URL mapped to the VI. You can find and copy the appropriate URL directly from the project.
5. If you notice an error or finish testing the Web service, (Windows) right-click the Web service project item in the project tree and select **Application Web Server»Stop (Debug Server)**. (Real-Time) Right-click the Web service project item and select

Stop (Debug Server). If startup VIs for a LabVIEW stand-alone application are running, you do not have to stop them.

Note You cannot edit HTTP method VIs until you stop the debugging session because LabVIEW reserves HTTP method VIs for execution during debugging.

6. Edit the VIs to correct any issues you noticed.
7. (Optional) Repeat steps 2-6 to validate your changes to the Web service files.

Accessing HTTP Method VIs in Debugging Mode

Complete the following steps to determine the URL of an HTTP method VI during a debugging session:

1. Right-click an HTTP method VI and select **Show Method URL** to display the HTTP Method URL dialog box.
2. In the **Available Servers** pull-down menu, select the item that contains **Local Debugging**, and then click **Copy URL**. The Web service runs in the debugging environment, so you access it on a different server and port than if you had published it to a target.

Note The Web service does not respond to any requests from the client if the **HTTP Method URL** dialog box is open.

3. Paste the URL into a standard web browser and replace the {value} variables with numeric values, such as: `http://127.0.0.1:8001/TutorialService/Add?b=2&a=3`.

Related Information

[Developing Web Services](#)

[Debugging Server](#)

Setting Up the Application Web Server for Web Services

To host your Web services on the Application Web Server, you need to set up the web server before you run or publish the Web services.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#) and [tutorial](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

Complete the following steps to enable and configure the [Application Web Server](#) for hosting Web services that are not part of a LabVIEW stand-alone application.

**NOTE**

The web server that hosts Web services included with a stand-alone application is automatically enabled.

1. Open a web browser.
2. In the browser, enter a URL for the target on which you want to enable the Application Web Server. The URL must conform to the following syntax:
 - To set up the Application Web Server on the local host system, enter the following URL: `http://localhost:3582`.
 - To set up the Application Web Server on a remote target system, right-click the target in a LabVIEW project, select **Properties**, and browse to the Web Server page. Click the **Configure Web Application Server** button. If the button is disabled, ensure the Application Web Server is installed on the target.

**TIP**

You also can enter the following URL in a Web browser: `http://[remote device IP address]` where [remote device IP address] is the IP address of the remote device.

**NOTE**

Because NI Web-based Configuration & Monitoring requires Microsoft Silverlight, you might need to browse to the URL with Microsoft Internet Explorer, which provides better compatibility with Microsoft Silverlight.

3. Click **Login** and enter a username and password to gain access to NI Web-based Configuration & Monitoring for the system. If you have not established security settings for NI Web-based Configuration & Monitoring on the system, enter the username Admin and a blank password to gain access using the default credentials.
4. Click the **Web Server Configuration** button to display the **Web Server Configuration** page.
5. Click **Help** for information about enabling and configuring the Application Web Server.

The Application Web Server runs as an independent system service. You can enable and run the Application Web Server without running LabVIEW on the host system.

Related Information

[Hosting Web Services](#)

Configuring Web Services Security when Using the Application Web Server (Real-Time, Windows)

You can use [Transport Layer Security \(TLS\)](#), previously known as Secure Sockets Layer (SSL), encryption, user and group permissions, and API keys to establish secure communication between Web clients and LabVIEW Web services.

Complete the following procedures to establish the different methods of Web services security:

- [Assigning Permissions to Web Service Files](#)
- [Configuring API Key Security](#)



NOTE

In this topic, you configure the security settings for Application Web Server. If you use the NI Web Server to host Web services, refer to [Configuring Web Services Security when Using the NI Web Server](#) for detailed instructions.

Assigning Permissions to Web Service Files

You can require a specific permission for the following types of files:

- **HTTP method VIs** —Set permissions that apply to individual VIs.
- **Public static files** —Set permissions that apply to all public static files in the Web service.

Complete the following steps to configure a Web service to accept only HTTP requests from Web clients with valid permissions:

1. Right-click the Web service project item  in the project tree and select **Properties** to display the **Web Service Properties** dialog box.
2. Browse to the page that applies to the type of file you want to configure:
 - For HTTP method VIs, browse to the HTTP Method VI Settings page, select a VI from the **Web Service VI** table, and click the **Application Web Server** tab in **Web Service VI Properties**.
 - For public static files, browse to the Service Settings page and the **Static Documents** section.
3. In the **Required permissions** table, add the name of a permission you previously created in the **Security Configuration** page of NI Web-based Configuration & Monitoring.

Accessing Web Service Files with Permissions

A Web client must provide a valid username and password whenever the client submits an HTTP request for a file protected by permissions. The username and password must correspond to a user in the NI Web-based Configuration & Monitoring utility that has been granted the same permission added for the HTTP method VI or public static file.

Use the OpenHandle VI to provide a **username** and **password** for HTTP requests made by a Web client built in LabVIEW.

Configuring API Key Security

You also can use API keys to restrict which clients can send *HTTP requests* to HTTP method VIs. An API key is a string of seemingly random characters that consists of two parts, an access ID and a secret ID. The access ID works similarly to a username, and the secret ID works similarly to a password intended only for authorized clients.

You can configure a single API key that applies to all Web services running on any web server, including the Application Web Server, web servers for embedded applications, and the Web service debugging server. You first must configure an access ID and secret ID for the Web Server to generate API keys. Then you must enable API key security for each URL mapping that you want to protect.

Configuring API Key Security on the Web Server

To establish an access ID and secret ID on the Application Web Server, launch the NI Web-based Configuration & Monitoring utility and navigate to the **Web Server Configuration** page. Use the **Web Services API Key** tab to generate, apply, and reset API keys.

Securing URL Mappings with API Key Security

Complete the following steps to enable API key security for an HTTP method VI in a Web service:

1. Right-click the Web service project item  in the project tree and select **Properties** to display the **Web Service Properties** dialog box.
2. On the HTTP Method VI Settings page, select a VI from the **Web Service VI** table, and click the **Application Web Server** tab in the **Web Service VI Properties**.
3. Place a checkmark in the **Require API key** checkbox.

Accessing URL Mappings with API Key Security

When you establish an API key and enable that API key for an HTTP method VI, any HTTP request from a Web client that corresponds to the VI must contain the correct API key. You can use the SetAPIKey VI to configure a Web client built in LabVIEW to provide an API key when making HTTP requests.

Related Information

[Hosting Web Services](#)

Configuring the Root Directory of the Web Server

If you use the built-in LabVIEW [Web Server](#), the root directory is the location from which the Web Server loads files for document requests. The default root directory for the Web Server is `labview\www`.

Complete the following steps to configure the root directory of the Web Server.

1. Select **Tools»Options** to display the Options dialog box.
2. Select the Web Server page from the **Category** list of the **Options** dialog box.
3. Place a checkmark in the **Enable Remote Panel Server** checkbox in the **Remote Panel Server** section.
4. Enter the path of the root directory you want the Web Server to use in the **Root directory** text box. You also can click the browse button to navigate to and select the directory.
5. Click the **OK** button to save the current configuration and close the dialog box.



NOTE

When a user enters the URL of the Web Server without specifying a filename, an error page or directory listing might display. You can create an `index.htm` file in the Web Server root directory that automatically loads instead.

Tutorials

Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server (Windows)

Complete the following steps to create, publish, and exchange data between a LabVIEW Web service and a web client, in this case, a web browser.



NOTE

LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

This tutorial contains the following sections:

1. [Creating a Web Service](#)
2. [Testing and Debugging the Web Service](#)
3. [Configuring How Clients Access the Web Service](#)
4. [Publishing the Web Service to the NI Web Server](#)
5. [Accessing the Web Service with a Client](#)

Creating a Web Service

Develop a Web service in a LabVIEW project to group together and configure the VIs and other files that make up the Web service. Complete the following steps to create a Web service project and two HTTP method VIs.

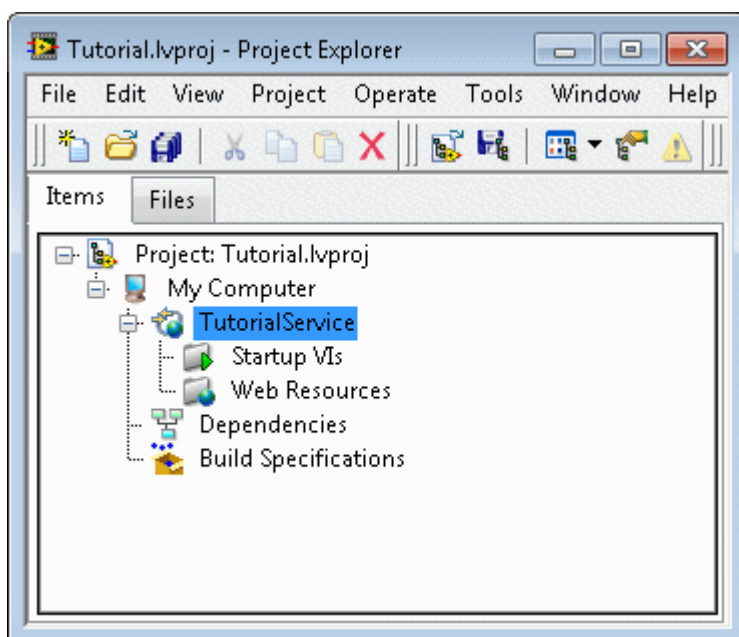
1. Create a LabVIEW project to organize the Web service files. Save the project as `Tutorial.lvproj`.
2. Right-click **My Computer** and select **New»Web Service**. LabVIEW adds a Web service project item and folders under the target.



NOTE

In this tutorial, you create a Web service and publish it on the host computer. You also can publish a Web service to a Real-Time (RT) target by creating a Web service project under the RT target in **Project Explorer**. If you want to publish your Web services to RT targets, you can use only the Application Web Server to host your Web services. Refer to [Tutorial: Creating and Publishing a LabVIEW Web Service to the Application Web Server](#) for detailed instructions.

3. Right-click the Web service project item , select **Rename**, and name the Web service `TutorialService`. The project appears as follows:



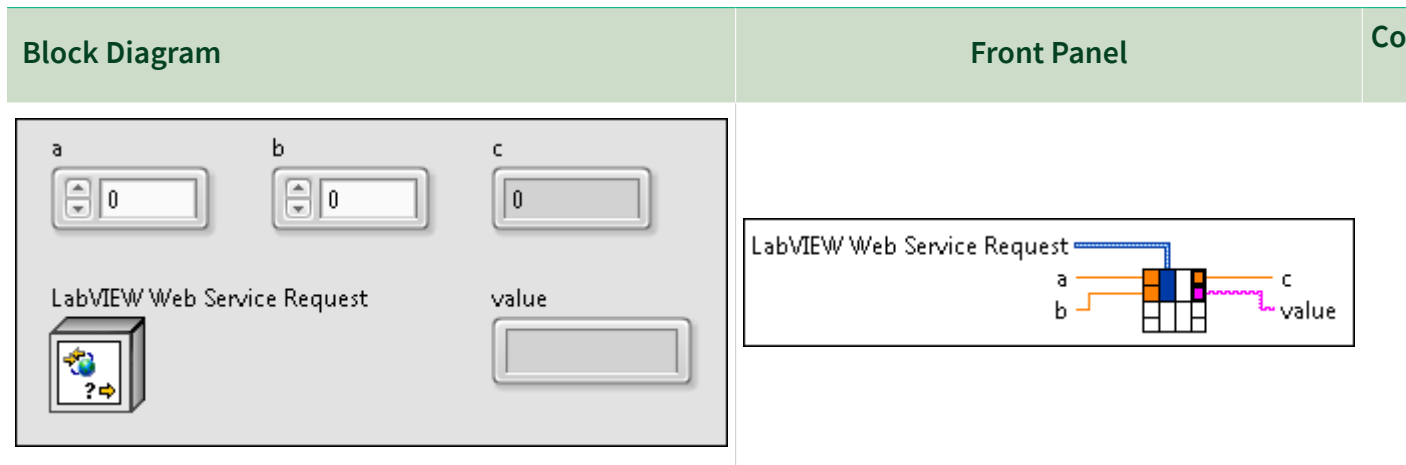
4. Right-click **Web Resources** and select **New VI** to create a new HTTP method VI, which is a VI that receives HTTP requests from clients and returns data to clients. LabVIEW opens a new VI from a template.



NOTE

By default, an HTTP method VI returns data to clients via the connector pane. In this tutorial, the HTTP method VIs return the default JSON-formatted text responses. When you create an HTTP method VI outside this tutorial, you also can stream data to a client in a custom format, such as a customized HTML page or image data. Configure the output format type on the HTTP Method VI Settings page of the **Web Service Properties** dialog box.

- In the new VI, build the block diagram, front panel, and connector pane as shown in the following figures.



When you publish this VI as part of a Web service, a web client can send an HTTP request to invoke it. When invoked, this VI performs the following actions:

- Receives the values of any controls assigned to the connector pane except **LabVIEW Web Service Request** from the HTTP request the client made. **LabVIEW Web Service Request** acts like a refnum by identifying the current HTTP request.
- Computes the sum of **a** and **b**.
- Returns the sum of those values to the client via the **c** indicator assigned to the connector pane.
- Executes the Read Request Variable VI and returns the query string the client included in the current HTTP request, which **LabVIEW Web Service Request** identifies.



TIP

In a more complex Web service, you might use other VIs on the Web Services palette to enable features such as setting HTTP headers and authentication.

- Notice that each control and indicator appears on the connector pane. With the default **Terminal** output type, Web services can receive data from clients or send data to clients only through inputs and outputs assigned to the connector pane of an HTTP method VI.

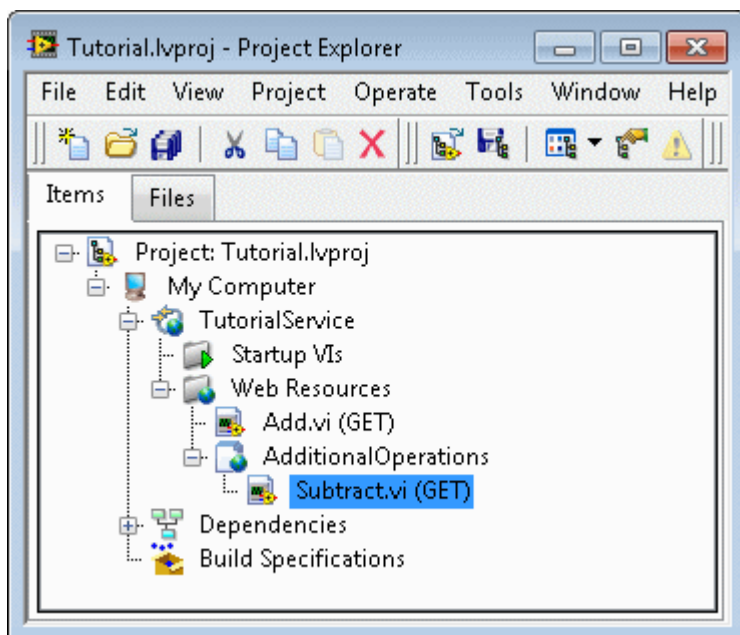


TIP

Assign recognizable names to each object. These names become part of the URL that web clients use to exchange data with the VI.

- Save the VI as `Add.vi`.
- Select **File»Save As** and select **Copy»Open additional copy** in the dialog box that prompts. Ensure the **Add copy to Tutorial.lvproj** checkbox contains a checkmark, and then click **Continue**.
- Save the new copy as `Subtract.vi`.
- Open the block diagram of `Subtract.vi` and replace the Add function with the Subtract function. Rewire the controls and indicators.

11. Return to the project, right-click **Web Resources**, and select **New Web Resource**. Rename the new project item **AdditionalOperations**.
12. Drag **Subtract.vi** under the **AdditionalOperations** project item. Save the project, which should appear as in the following figure.



Notice the following features in the project tree:

- LabVIEW appends an HTTP request type to the name of each HTTP method VI, which by default is GET. You can change the type of HTTP request that clients send to a specific HTTP method VI later in this tutorial.
- The Web service project item  also contains a **Startup VIs** folder. This tutorial does not use startup VIs, but you can find more information about them at the end of the tutorial.

Related Information

[Components of a Web Service](#)

[Assigning Terminals to Controls and Indicators](#)

[Developing HTTP Method VIs: Choosing the Format of Output Data](#)

[Configuring HTTP Headers, Streaming, and Buffering](#)

[Managing a Project in LabVIEW](#)

Testing and Debugging the Web Service

Before you publish the Web service to a target, you might want to test that the HTTP method VIs communicate with clients as expected. You can run the Web service on the NI Web Server directly from the LabVIEW project to allow a client to send requests to HTTP method VIs.

1. Right-click the Web service project item  and select **Start**.
2. If you are using the NI Web Server for the first time on your computer, complete the following steps to enable and configure the NI Web Server:
 1. Click **Configure NI Web Server** in the **Start Web Service** dialog box to launch NI Web Server Configuration.
 2. For this tutorial, choose the **Simple local access** configuration preset and click **Next**.
 3. On the **Authentication** tab, select **Log in using an admin user for server administration** and create your password. Click **Next**.
 4. Click **Finish** to apply this configuration.
3. Switch back to the LabVIEW project and repeat step 1. The Web service is now running on the NI Web Server.
4. Open `Add.vi`. Notice that the **Run** button indicates the VI is reserved for execution  because LabVIEW loads the Web service and makes it available on the NI Web Server for communication with clients.
5. Close `Add.vi`.
6. Complete the following steps to get the URL that clients use to invoke `Add.vi`:
 1. Right-click `Add.vi` and select **Show Method URL** to display the HTTP Method URL dialog box.
 2. In the **Available Servers** pull-down menu, select the item that contains **NI Web Server**, and then click **Copy URL**.
 3. Close the dialog box.



NOTE

The Web service does not respond to any requests from the client if the **HTTP Method URL** dialog box is open.

7. Paste the URL from step 6 into a standard web browser and replace the {value} variables with numeric values, such as: `http://127.0.0.1:80/TutorialService/Add?b=2&a=3`.



NOTE

NI Web Server uses 80/443 as the default port in accordance with HTTP/HTTPS protocol. However, the port number might change if another application occupies the port. You can customize the port number in NI Web Server Configuration. Refer to *Configuring NI Web Server* for more information on server settings like authentication, port, and remote connections.

8. Browse to the URL. `Add.vi` receives the HTTP request from the browser, computes the sum of the values for **a** and **b**, and returns a response with the name and output value of each indicator assigned to the connector pane of `Add.vi`. In this tutorial,

the response is a JSON string, which is the default **Terminal** output format. Recall that `Add.vi` has two indicators assigned to its connector pane: **c** and **value**, whose values the HTTP method VI returns in the response.

```
{ "c": 5, "value": "b=2&a=3" }
```



NOTE

If you notice unexpected behavior, you must stop the debugging session before you can edit the Web service files because you cannot edit the files while LabVIEW reserves them for execution. Right-click the Web service project item and select **Stop**.

9. Repeat steps 6 and 7 for `Subtract.vi`, or update the URL from step 7 to reflect the unique URL mapping for `Subtract.vi`: `http://127.0.0.1:80/TutorialService/AdditionalOperations/Subtract?b=2&a=3`.
10. Repeat step 8 for `Subtract.vi`.
11. Right-click the Web service project item and select **Stop**.

Related Information

[Testing and Debugging a Web Service](#)

[Setting up the NI Web Server for Web Services](#)

Configuring How Clients Access the Web Service

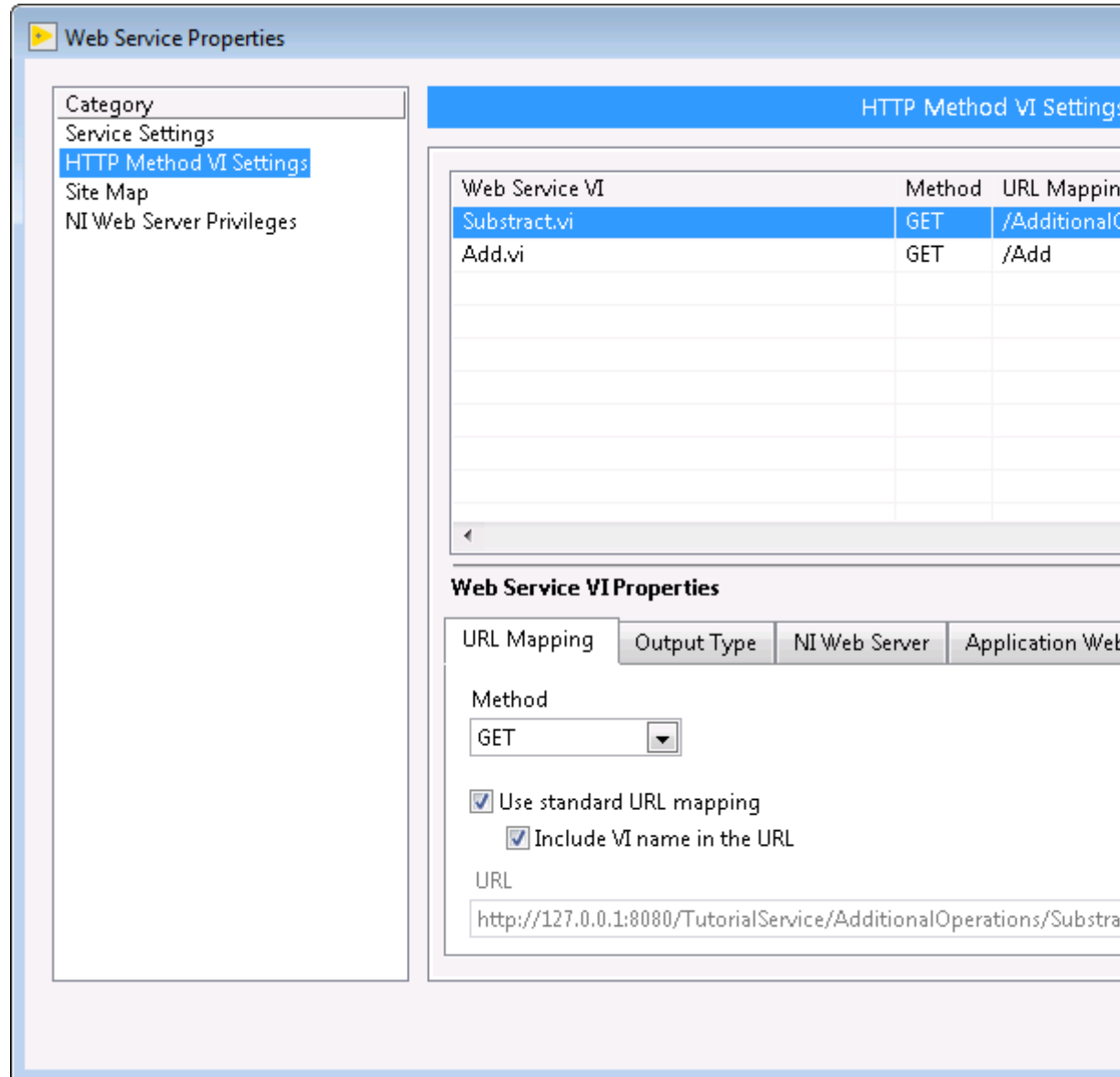
Web clients use URLs to exchange data with HTTP method VIs. Later in this tutorial, you will enter a specific URL in a web browser to send data directly to the controls on the connector pane of the HTTP method VIs.

Complete the following steps to define the URLs clients can use to access the HTTP method VIs:

1. Right-click the Web service project item  and select **Properties** to display the **Web Service Properties** dialog box.
2. On the HTTP Method VI Settings page, notice that HTTP method VIs from the **Web Resources** folder in the LabVIEW project appear in the table of Web service VIs. Further, notice how placing `Subtract.vi` under the `AdditionalOperations` project item affects the URL mapping compared to `Add.vi`.
3. Select `Subtract.vi` in the table and notice the following features on the **URL Mapping** tab at the bottom of the page:
 - You can change the HTTP request type in the **Method** pull-down menu. Use the default GET value for `Subtract.vi`.
 - You can choose to disable the standard format LabVIEW uses to map a URL to the VI and instead customize the part of the URL that appears after the

service name. For this tutorial, use the standard URL mapping and include the VI name.

- The **URL** field displays the base URL LabVIEW creates for the HTTP method VI. For example, for `Subtract.vi`, LabVIEW creates the default URL mapping `/AdditionalOperations/Subtract`.



NOTE

In a more complex Web service, you might use the remaining tabs on this page to configure the format that the VI returns output data to the web client, as well as security settings like privileges.

4. Click **OK** to close the **Web Service Properties** dialog box.
5. Save `Tutorial.lvproj`.

Related Information

[Understanding URL Mappings and Query Strings in Web Services](#)

Publishing the Web Service to the NI Web Server

You can publish the Web service you create to the NI Web Server. The NI Web Server is a production-grade web server that can host user-authored services, such as LabVIEW Web services, and SystemLink™ services created by NI. The NI Web Server protects web applications against common web security threats, provides high scalability to many enterprise-grade data services, and allows device management. In this tutorial, you publish the web service by including it in an NI package and installing the package.



NOTE

You can also publish your Web service to the Application Web Server. Refer to [Tutorial: Creating and Publishing a LabVIEW Web Service to the Application Web Server](#) for more details.

Complete the following steps to publish the Web service to the NI Web Server running on the local computer through a package:

1. Right-click the **Build Specifications** in the **Project Explorer** window and select **New»Package** to launch the Package Properties dialog box.
2. On the Information page, specify the location to build the package in the **Package output directory** text box.
3. On the Web Services page, place a checkmark in the TutorialService checkbox to include the Web service in the package.
4. Click **Build** and the Build Status dialog box appears.
5. When LabVIEW completes building the package, click **Explore** to open the destination directory where LabVIEW saves the package file .nipkg.
6. Double-click the package to install it through the NI Package Manager. When the installation is complete, the Web service is published to the NI Web Server on the host computer.
7. Close the LabVIEW project. Published Web services run on the LabVIEW Runtime and do not depend on the LabVIEW Development System.

Related Information

[Publishing Web Services](#)

Accessing the Web Service with a Client

Complete the following steps to send data to the published Web service from a web browser via a URL:

1. Paste the URL from step 7 in the [Testing and Debugging the Web Service](#) section into a standard web browser and replace the {value} variables with numeric values, such as: `http://127.0.0.1:80/TutorialService/Add?b=2&a=3`.

**NOTE**

If the client is located on a different system than where you publish the Web service, you must replace `127.0.0.1` with the actual IP address of the hosting system or the hostname you customize in NI Web Server Configuration.

2. The browser returns a JSON response with the output values from each indicator assigned to the connector pane.

Related Information[Sending Data to a Deployed Application Using URLs](#)**Where to Go Next**

- [Overview: Web-based Communication with a LabVIEW Application](#) —An overall introduction to LabVIEW Web Service
- [Publishing a Web Service through a Package](#) —Include a Web service in an NI package, so that the Web service runs when you install the package.
- [Publishing a Web Service through a Stand-Alone Application](#) —Include a Web service in a LabVIEW stand-alone application, so that the Web service runs when the application runs.
- [Developing HTTP Method VIs](#) or [Developing Startup VIs to Initialize and Communicate with HTTP Method VIs](#) —Develop more complex HTTP method VIs, as well as startup VIs that run as part of a Web service application but do not exchange data with a web client.
- [Integrating Static Content into a Web Service](#) —Publish static content such as HTML pages or images with a Web service.

Example Code

Refer to the following LabVIEW projects for examples of using Web services:

- `examples\Connectivity\Web Services\Weather Monitor\Web Services - Weather Monitor.lvproj`

Tutorial: Creating and Publishing a LabVIEW Web Service to the Application Web Server (Real-Time, Windows)

Complete the following steps to create, publish, and exchange data between a LabVIEW Web service and a web client, in this case, a web browser.



NOTE

For more information about concepts in this topic, refer to the [Web services introduction](#). LabVIEW Web services are available only in the LabVIEW Full Development System and the LabVIEW Professional Development System.

This tutorial contains the following sections:

1. [Creating a Web Service](#)
2. [Testing and Debugging the Web Service](#)
3. [Configuring How Clients Access the Web Service](#)
4. [Publishing the Web Service to the Application Web Server](#)
5. [Accessing the Web Service with a Client](#)
6. [Monitoring the Web Service](#)

Creating a Web Service

Develop a Web service in a LabVIEW project to group together and configure the VIs and other files that make up the Web service. Complete the following steps to create a Web service project and two HTTP method VIs.

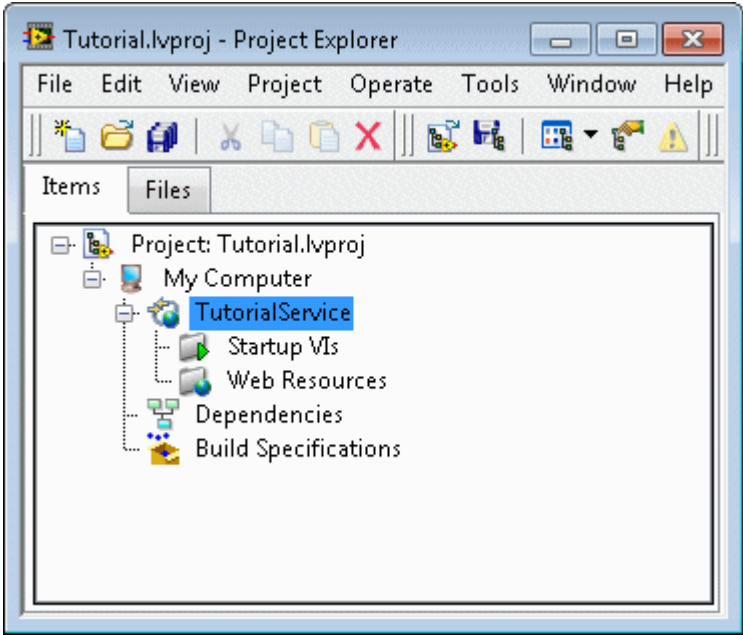
1. Create a LabVIEW project to organize the Web service files. Save the project as `Tutorial.lvproj`.
2. Right-click **My Computer** and select **New»Web Service**. LabVIEW adds a Web service project item and folders under the target.



NOTE

- In this tutorial, you create a Web service and publish it on the host computer. You also can publish a Web service to a Real-Time target by creating a Web service project under the RT target in a LabVIEW project.
- (Windows) NI recommends using the NI Web Server to host your Web service on a Windows computer. Refer to [Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server](#) for detailed instructions.

3. Right-click the Web service project item , select **Rename**, and name the Web service `TutorialService`. The project appears as follows:



4. Right-click **Web Resources** and select **New VI** to create a new HTTP method VI, which is a VI that receives HTTP requests from clients and returns data to clients. LabVIEW opens a new VI from a template.



NOTE

By default, an HTTP method VI returns data to clients via the connector pane. In this tutorial, the HTTP method VIs return the default JSON-formatted text responses. When you create an HTTP method VI outside this tutorial, you also can stream data to a client in a custom format, such as a customized HTML page or image data. Configure the output format type on the HTTP Method VI Settings page of the **Web Service Properties** dialog box.

5. In the new VI, build the block diagram, front panel, and connector pane as shown in the following figures.

Block Diagram	Front Panel	Co

When you publish this VI as part of a Web service, a web client can send an HTTP request to invoke it. When invoked, this VI performs the following actions:

- Receives the values of any controls assigned to the connector pane except **LabVIEW Web Service Request** from the HTTP request the client

made. **LabVIEW Web Service Request** acts like a refnum by identifying the current HTTP request.

- Computes the sum of **a** and **b**.
- Returns the sum of those values to the client via the **c** indicator assigned to the connector pane.
- Executes the Read Request Variable VI and returns the query string the client included in the current HTTP request, which **LabVIEW Web Service Request** identifies.



TIP

In a more complex Web service, you might use other VIs on the Web Services palette to enable features such as HTTP sessions, authentication, HTTP headers, and cookies.

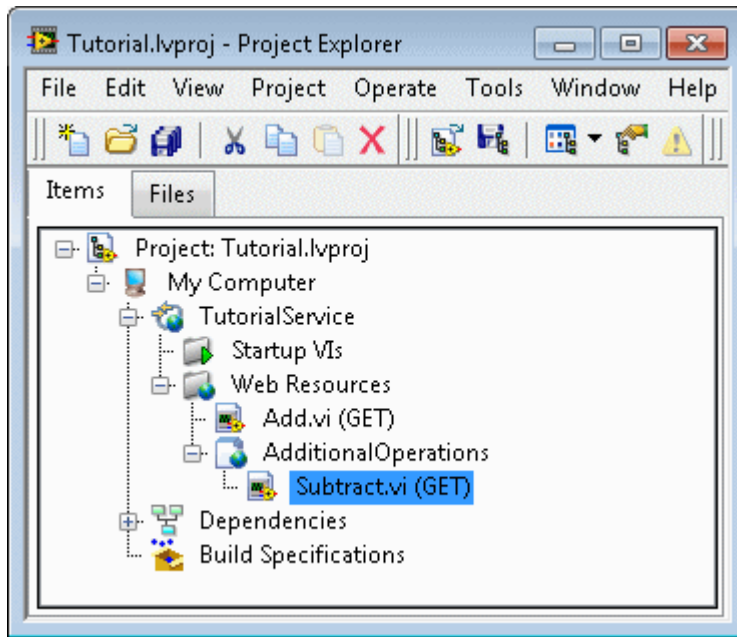
6. Notice that each control and indicator appears on the connector pane. With the default **Terminal** output type, Web services can receive data from clients or send data to clients only through inputs and outputs assigned to the connector pane of an HTTP method VI.



TIP

Assign recognizable names to each object. These names become part of the URL that web clients use to exchange data with the VI.

7. Save the VI as `Add.vi`.
8. Select **File»Save As** and select **Copy»Open additional copy** in the dialog box that prompts. Ensure the **Add copy to Tutorial.lvproj** checkbox contains a checkmark, and then click **Continue**.
9. Save the new copy as `Subtract.vi`.
10. Open the block diagram of `Subtract.vi` and replace the Add function with the Subtract function. Rewire the controls and indicators.
11. Return to the project, right-click **Web Resources**, and select **New Web Resource**. Rename the new project item `AdditionalOperations`.
12. Drag `Subtract.vi` under the **AdditionalOperations** project item. Save the project, which should appear as in the following figure.



Notice the following features in the project tree:

- LabVIEW appends an HTTP request type to the name of each HTTP method VI, which by default is GET. You can change the type of HTTP request that clients send to a specific HTTP method VI later in this tutorial.
- The Web service project item  also contains a **Startup VIs** folder. This tutorial does not use startup VIs, but you can find more information about them at the end of the tutorial.

Related Information

[Components of a Web Service](#)

[Assigning Terminals to Controls and Indicators](#)

[Developing HTTP Method VIs: Choosing the Format of Output Data](#)

[Configuring HTTP Headers, Streaming, and Buffering](#)

[Managing a Project in LabVIEW](#)

Testing and Debugging the Web Service

Before you publish the Web service to a target, you might want to test that the HTTP method VIs communicate with clients as expected. You can place the Web service on a debugging server directly from the LabVIEW project to allow a client to send requests to HTTP method VIs. The debugging server provides a sandbox-like environment.

1. (Windows) Right-click the Web service project item  under **My Computer** and select **Application Web Server»Start (Debug Server)**. (Real-Time) Right-click the Web service project item  under the RT target and select **Start (Debug Server)**.
2. Open Add.vi. Notice that the **Run** button indicates the VI is reserved for execution  because LabVIEW loads the Web service and makes it available on the debugging server for communication with clients.
3. Close Add.vi.
4. Complete the following steps to get the URL that clients use to invoke Add.vi :
 1. Right-click Add.vi and select **Show Method URL** to display the HTTP Method URL dialog box.
 2. In the **Available Servers** pull-down menu, select the item that contains **Local Debugging**, and then click the **Copy URL** button. Clients must communicate with Web services on the debugging server through a different port than if you publish the service to a web server.
 3. Close the dialog box.

**NOTE**

The Web service does not respond to any requests from the client if the **HTTP Method URL** dialog box is open.

5. Paste the URL from step 4 into a standard web browser and replace the {value} variables with numeric values, such as: `http://127.0.0.1:8001/TutorialService/Add?b=2&a=3`.
6. Browse to the URL. Add.vi receives the HTTP request from the browser, computes the sum of the values for **a** and **b**, and returns a response with the name and output value of each indicator assigned to the connector pane of Add.vi. In this tutorial, the response is a JSON string, which is the default **Terminal** output format. Recall that Add.vi has two indicators assigned to its connector pane, **c** and **value**, whose values the HTTP method VI returns in the response.

```
{"c":5,"value":"b=2&a=3"}
```

**NOTE**

If you notice unexpected behavior, you must stop the debugging session before you can edit the Web service files because you cannot edit the files while LabVIEW reserves them for execution.

7. Repeat steps 4 and 5 for Subtract.vi, or update the URL from step 5 to reflect the unique URL mapping for Subtract.vi: `http://127.0.0.1:8001/TutorialService/AdditionalOperations/Subtract?b=2&a=3`.
8. Repeat step 6 for Subtract.vi.
9. (Windows) Right-click the Web service project item and select **Application Web Server»Stop (Debug Server)**. (Real-Time) Right-click the Web service project item and select **Stop (Debug Server)**.

Related Information

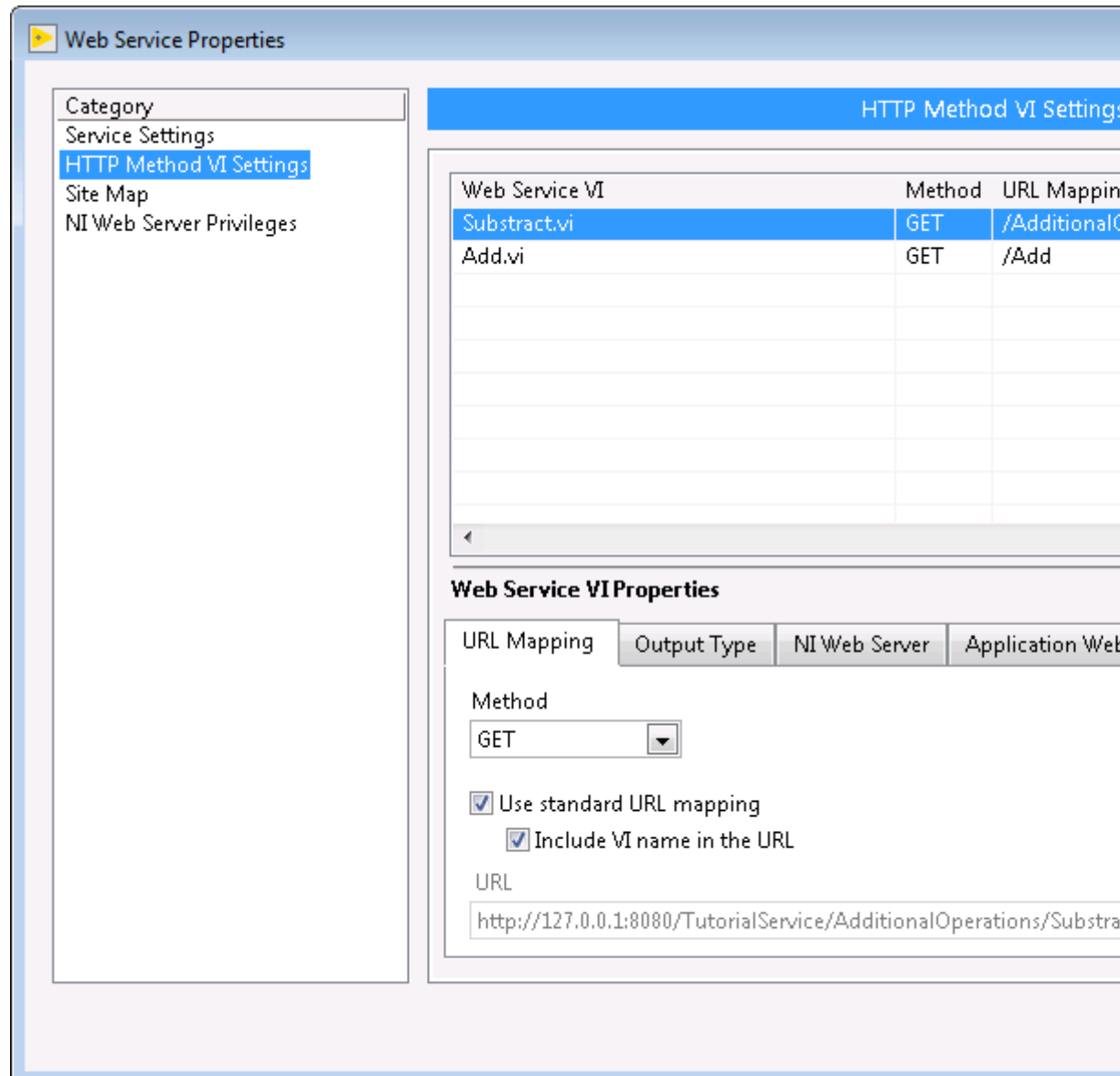
Testing and Debugging a Web Service on the Application Web Server

Configuring How Clients Access the Web Service

Web clients use URLs to exchange data with HTTP method VIs. Later in this tutorial, you will enter a specific URL in a web browser to send data directly to the controls on the connector pane of the HTTP method VIs.

Complete the following steps to define the URLs clients can use to access the HTTP method VIs:

1. Right-click the Web service project item  and select **Properties** to display the **Web Service Properties** dialog box.
2. On the HTTP Method VI Settings page, notice that HTTP method VIs from the **Web Resources** folder in the LabVIEW project appear in the table of Web service VIs. Further, notice how placing `Subtract.vi` under the `AdditionalOperations` project item affects the URL mapping compared to `Add.vi`.
3. Select `Subtract.vi` in the table and notice the following features on the **URL Mapping** tab at the bottom of the page:
 - You can change the HTTP request type in the **Method** pull-down menu. Use the default GET value for `Subtract.vi`.
 - You can choose to disable the standard format LabVIEW uses to map a URL to the VI and instead customize the part of the URL that appears after the service name. For this tutorial, use the standard URL mapping and include the VI name.
 - The **URL** field displays the base URL LabVIEW creates for the HTTP method VI. For example, for `Subtract.vi`, LabVIEW creates the default URL mapping `/AdditionalOperations/Subtract`.



NOTE

In a more complex Web service, you might use the remaining tabs on this page to configure the format that the VI returns output data to the web client, as well as security settings like permissions.

- Click **OK** to close the **Web Service Properties** dialog box.
- Save Tutorial.lvproj.

Related Information

[Understanding URL Mappings and Query Strings in Web Services](#)

Publishing the Web Service to the Application Web Server

You can publish the Web service you create to the Application Web Server. In this tutorial, you publish a stand-alone Web service, but in a more complex application, you might publish a Web service through a LabVIEW stand-alone application or an installer. The Application Web Server hosts Web services on a network and provides multiple security-related features to protect the network data exchange, including Secure Socket Layer (SSL) encryption.



NOTE

(Windows) You can also publish your Web service to the NI Web Server. Refer to [Tutorial: Creating and Publishing a LabVIEW Web Service to the NI Web Server](#) for detailed instructions.

Complete the following steps to publish the Web service to the Application Web Server running on the local computer:

1. Right-click the Web service project item and select **Application Web Server»Publish**. The **Deployment Progress** dialog box appears.
2. If LabVIEW publishes the Web service without errors, click the **Close** button. Web clients can now exchange data with the HTTP method VIs.

Related Information

[Publishing Web Services](#)

[Setting up the Application Web Server for Web Services](#)

Accessing the Web Service with a Client

Complete the following steps to send data to the published Web service from a web browser via a URL:

1. Right-click **Add.vi** and select **Show Method URL** to display the HTTP Method URL dialog box.
2. In the **Available Servers** pull-down menu, select the item that contains **Application**, and then click the **Copy URL** button. This URL contains a different port number than the URL for the debugging server. LabVIEW uses this port number for communication with the Web service files running on the Application Web Server.
3. Close the dialog box.
4. Paste the URL into a standard web browser and replace the {value} variables with numeric values, such as: `http://127.0.0.1:8080/TutorialService/Add?b=2&a=3`.



NOTE

If the client is located on a different system than where you publish the Web service, you must replace `127.0.0.1` with the actual IP address of the hosting system.

5. The browser returns a JSON response with the output values from each indicator assigned to the connector pane.

Related Information

[Sending Data to a Deployed Application Using URLs](#)

Monitoring the Web Service

Complete the following steps to check the status of Web services published to the Application Web Server:

1. Right-click the Web service project item and select **Application Web Server»Manage Web Server**. NI Web-based Configuration & Monitoring opens in a web browser.
2. Click the **Web Services Management** button in the left-hand sidebar .
3. Select **TutorialService** in the **Published Web Services** list. If the list is empty, click the **Refresh** button.
4. Notice that the status of the Web service is **Running**. To pause, resume, restart, and unpublish Web services, use the buttons at the bottom of the page.
5. On the right side of the page, notice that the URL mappings for the HTTP method VIs, **Add** and **AdditionalOperations/Subtract**, appear with the HTTP method they perform.

Where to Go Next

- *[Communicating with a LabVIEW Application from Web Clients](#)*
- *[Overview: Web-based Communication with a LabVIEW Application](#)* —An overall introduction to LabVIEW Web Service
- *[Publishing a Web Service through a Stand-Alone Application](#)* —Include a Web service in a LabVIEW stand-alone application, so that the Web service runs when the application runs.
- *[Publishing a Web Service through an Installer](#)* —Include a Web service in a Windows installer, so that the Web service runs when you finish the installation process.
- *[Developing HTTP Method VIs](#)* or *[Developing Startup VIs to Initialize and Communicate with HTTP Method VIs](#)* —Develop more complex HTTP method VIs, as well as startup VIs that run as part of a Web service application but do not exchange data with a web client.
- *[Integrating Static Content into a Web Service](#)* —Publish static content such as HTML pages or images with a Web service.

Example Code

Refer to the following LabVIEW projects for examples of using Web services:

- `examples\Connectivity\Web Services\Weather Monitor\Web Services – Weather Monitor.lvproj`

Adding New Browser Access List Entries

Complete the following steps to add a new entry to the **Browser Access List**.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Web Server from the **Category** list.
3. Type a valid browser address in the **Browser Address** text box. Enter a domain name, such as `mydomain.com`, or an IP address, such as `130.164.123.123`. You also can *use wildcards* in the browser address.



NOTE

If the Web Server runs on a computer that does not have access to a DNS server, do not use domain name entries in the **Browser Access List**. Requests to resolve the domain name or IP address fail, slowing down the computer.

1. Click the **Allow Viewing and Controlling**, **Allow Viewing**, or **Deny Access** buttons to allow or deny access to the selected item, respectively. Two green checkmarks appear to the left of the item when you allow viewing and controlling of the front panel, a single green checkmark appears when you allow only viewing of the front panel, and a red X appears when you deny access.
2. If an entry does not have one or two green checkmarks or a red X by its name, the syntax for the entry is incorrect. Edit the entry or click the **Remove** button to delete it.
3. Click the **OK** button to close the **Options** dialog box and save the changes.

Allowing and Denying Web Browser Access

Use the **Browser Access** section on the Web Server page of the Options dialog box to configure which browser addresses can view your VI front panels. Create a **Browser Access List** that allows and denies access to individual browser addresses. You also can use the Web Server:TCP/IP Access List property to allow and deny access to browser addresses programmatically.

When a Web browser attempts to obtain a VI front panel image, the Web Server compares the browser address to the entries in the **Browser Access List** to determine whether it should grant access. If an entry in the **Browser Access List** matches the browser address, the Web Server permits or denies access based on how you set up the entry. By default all browsers have access to the LabVIEW Web Server. When you add new **Browser Access List**

entries, edit existing entries, or remove entries, use the correct syntax, order, and wildcards in the entries.



RELATED INFORMATION

[Syntax for Browser Address Lists](#) on page 1579

[Adding New Browser Access List Entries](#) on page 1570

[Editing Existing Browser Access List Entries](#) on page 1571

[Removing Browser Access List Entries](#) on page 1461

[Syntax for Browser Address Lists](#) on page 1579

Editing Existing Browser Access List Entries

Complete the following steps to edit a browser address in the **Browser Access List**.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Web Server from the **Category** list.
3. Select the address you want to modify in the **Browser Access List** in the **Browser Access** section. The address appears in the text box.
4. Edit the address in the text box or click the **Allow Viewing and Controlling, Allow Viewing**, or **Deny Access** buttons. You also can [use wildcards](#) in the browser address.
5. Click the **OK** button to close the **Options** dialog box and save the changes.

Front Panels

Controlling an Application or Front Panel Remotely Using a Browser or LabVIEW

Complete the following steps to control an application or front panel remotely using a Web browser or LabVIEW.



NOTE

The application or VI you want to view must be in memory on the server computer.

1. View an [application or front panel remotely in a browser](#) or in [LabVIEW](#).
2. Click the bottom of the remote front panel window or right-click the front panel anywhere and select **Request Control of VI** from the shortcut menu. If no other client is currently in control, you have control of the front panel. If another client is currently controlling the VI, the server queues the request until the other client relinquishes control or until the control time limit times out. Only the user at the server computer can [monitor the queue list](#).

To completely close the connection to the Web Server running on the server computer, click the close glyph on the browser window or LabVIEW window border. However, if you want to release control of the front panel but maintain a connection so you can view the front panel, right-click the front panel in any open space and select **Release Control of VI** from the shortcut menu or right-click a front panel object, such as a graph, and select **Remote Panel Client»Release Control of VI** from the shortcut menu.



Note If you want to save the data generated by a VI running on a remote computer, use [DataSocket](#) or [TCP](#) instead of remote front panels.

You also can use the Application methods to release and request control of a remote front panel programmatically.

Enabling and Configuring the Web Server for Remote Front Panels

Complete the following steps to enable and configure the LabVIEW Web Server for [viewing and controlling front panels remotely](#). The Web server only can run while LabVIEW remains open on the host machine.

1. Select **Tools»Options** to display the Options dialog box.
2. Select the Web Server page from the **Category** list of the **Options** dialog box.
3. Place a checkmark in the **Enable Remote Panel Server** checkbox in the **Remote Panel Server** section. The default Web Server configuration is suitable for most applications.



NOTE

(Linux) By default, LabVIEW uses port 8000 so the Web Server can run in accounts that do not have root access. If you have root access, you can change the port to 80. If you use any port other than 80, you must specify the port number in URLs that refer to the server, as shown in the following example:

```
http://hostname:8000/index.htm
```

1. To [control browser access](#) to the front panels or [control which VIs are visible](#) on the Web, configure the **Browser Access** and **Visible VIs** sections of the **Web Server** page.
2. Click the **OK** button to close the **Options** dialog box and save the changes. The Web Server is running.

Use the Web Server properties to enable and configure the Web Server programmatically. You also can [enable and configure the Web Server in a built application](#).

Enabling TLS/SSL Security for Remote Front Panels (Windows)

You can enable [Transport Layer Security \(TLS\)](#), previously known as Secure Sockets Layer (SSL), encryption for network communication involving [remote front panels](#).



NOTE

To use TLS/SSL encryption with [LabVIEW Web services](#), you must [enable TLS/SSL on the Application Web Server](#).

Complete the following steps to enable TLS/SSL on the LabVIEW Web Server for remote front panels.

1. Navigate to the **Remote Panel Server** section on the Web Server page of the Options dialog box. If necessary, [enable the LabVIEW Web Server for use with remote front panels](#).
2. Place a checkmark in the **SSL** checkbox.
3. Enter the network port to encrypt with TLS/SSL in the **SSL Port** field or use the default TLS/SSL port 443. You cannot enable TLS/SSL on network communication that uses the default port 80.
4. Leave the **SSL certificate file** text box blank to use the [default LabVIEW self-signed certificate](#).
5. (Optional) Specify the address of the system or target that contains the certificate in the **Server address** text box.



NOTE

You also can navigate directly to the certificate file that you created using the **SSL certificate file** text box.

6. (Optional) Click the **Query** button to display available certificates located on the server in the **Discovered certificates** listbox.
7. (Optional) Select the certificate from the **Discovered certificates** listbox.

Errors When Connecting to Remote Front Panels

The following table lists errors that might occur when you [connect to remote front panels in LabVIEW](#) or in a [Web browser](#).

Error	Solution
Invalid server IP address	Contact the server administrator to get the correct IP address or name of the computer to which you want to connect.
Remote panel connection refused by specified server	The server administrator must enable the LabVIEW Web Server . Verify that the Server IP Address and Port you entered in the Connect to Remote Panel dialog box are correct.

Error	Solution
Client does not have access to remote panel server	The server administrator must <i>specify the IP addresses</i> for all potential clients wanting to view or control a remote front panel. If the server administrator does not specify any IP addresses, LabVIEW allows all clients to access remote front panels.
Remote panel protocol version incompatible	The client and server computers must be running the same version of LabVIEW. If you are using a browser to view and control a remote front panel, you must <i>use a version of the LabVIEW Run-Time Engine</i> compatible with the version of LabVIEW on the server computer. Also, contact the server administrator to verify that the HTML document <i>specifies the correct version</i> of the LabVIEW Run-Time Engine.
LabVIEW client version is incompatible with LabVIEW server version	You must use a version of LabVIEW compatible with the version of LabVIEW on the server computer. If you are using a browser to view and control a remote front panel, you must <i>use a version of the LabVIEW Run-Time Engine</i> compatible with the version of LabVIEW on the server computer. Also, contact the server administrator to verify that the HTML document <i>specifies the correct version</i> of the LabVIEW Run-Time Engine.
Server does not support Remote Panels	Make sure you entered the IP address or computer name of the server correctly.
Remote panel server failed to send requested VI	A memory or network problem occurred. This error might occur if you try to connect to a large front panel. Try to connect to the front panel again, or request that the server administrator modify the front panel so it uses less memory. This error also might occur if you deploy a VI to a Real-time target without the Web Server enabled, and subsequently enable the Web Server. The error occurs because the deployed VI does not have a front panel in this case, since LabVIEW only preserves the front panel of a VI deployed to a Real-Time target when the Web server is enabled. This behavior is for performance optimization.
Client does not have access to specified VI	The server administrator must <i>specify the VIs</i> that clients can view or control.
Requested VI is not loaded into memory on the server computer	A VI is considered to be in memory whenever the VI is open on the server machine. The VI does not have to be running. Also, make sure the VI name you entered matches the host machine exactly, including the.vi extension.

Functionality Not Supported in Viewing and Controlling Remote Front Panels

The following list includes functionality not supported and recommendations to consider when viewing and controlling remote front panels.

- Because of the constraints of a Web browser, user interface applications that attempt to manipulate the dimensions and location of a front panel do not work properly when that front panel is displayed as a part of a Web page. Although the Web Server and the LabVIEW browser plug-in attempt to preserve the fidelity of complex user interface applications—in particular, those that present dialog boxes and subVI windows—some applications might not work properly in the context of a Web browser. National Instruments recommends that you do not export these types of applications for use in a Web browser.
- Avoid exporting VIs that have While Loops but no wait function. These VIs prevent background tasks from performing in a reasonable amount of time, making front panels unresponsive when viewed or controlled remotely.
- Some VIs might not work exactly the same way from a remote computer as they do when run locally. Embedded.NET and ActiveX controls do not display on a remote client because they draw and operate almost completely independently of LabVIEW. If a VI presents the standard file dialog box, the controller receives an error because you cannot browse a file system remotely. Also, the browse button of a path control is disabled in remote panels.
- Clients viewing a front panel remotely might see different behavior depending on whether the front panel they are connecting to is from a built application. Specifically, if the front panel is from a built application, any programmatic changes to the front panel made before the client connects to the front panel are not reflected on the client computer. For example, if a Property Node changes a caption on a control before a client connects to that front panel, the client will see the original caption of the control, not the changed caption.
- Only a controller can remotely view the front panel of a VI dynamically opened and run using the VI Server or the front panel of a subVI configured to display the front panel when called. Clients not controlling the VI cannot view the front panel.
- Block diagrams that achieve certain user interface effects by polling properties of a front panel control might experience decreases in performance when you control the VI from a remote computer. You can improve the performance of these VIs by using the Wait for Front Panel Activity function.
- LabVIEW cannot generate a Panel Close event for a VI being viewed or controlled remotely. If you are viewing or controlling a VI remotely, LabVIEW can generate events only in the Control class, that is not in the Application or VI class.
- If you are remotely viewing or controlling a VI which displays the front panel of a subVI when calling the subVI, avoid manually closing the front panel of the subVI. Otherwise, you might lose control of the VI on both the client computer and the server computer. If you must view the front panel of the subVI when called, configure LabVIEW to *close the subVI after it runs*.

Locking and Unlocking Remote Front Panels

You can lock and unlock a front panel that a client views remotely so the client cannot or can change the values of the front panel controls.

Right-click anywhere on the front panel of the running VI and select **Lock Control** to lock a front panel that a client views remotely. When you lock a front panel that clients view remotely, LabVIEW queues any requests from clients to control the front panel.

You also can use the Remote Panel:Lock Control method to lock control of a remote front panel programmatically.

Right-click anywhere on the front panel of the running VI and select **Unlock Control** to allow a client viewing the front panel remotely to change the value of front panel controls. When you unlock a front panel that clients view remotely, LabVIEW grants control to the next client in the queue. If no clients are in the queue, the method unlocks the front panel.

You also can use the Remote Panel:Unlock Control method to unlock control of a remote front panel programmatically.

Managing Client Traffic to a Front Panel

If you are managing a server that provides front panels for clients to view and control from remote computers, you can monitor all client traffic to the server. Select **Tools»Remote Panel Connection Manager** to display the Remote Panel Connection Manager window.

Use this window to manage the client queue list, disconnect a client, and specify how long to retain network client traffic data. To disconnect all clients, select a VI name or **Total Network Traffic** and click the **Disconnect Client** button.

You also can use the Remote Panel:Connections To Servers and the Remote Panel:Connections To Clients properties to retrieve information about remote front panel connections.

Publishing VIs and Stand-Alone Applications on the Web

Use the LabVIEW Web Server to publish images of front panels from a VI or a stand-alone application to the Web. By default, after you enable the Web Server, all VIs and applications are visible to all Web browsers. However, you can control browser access to the published front panels and configure which VIs are visible on the Web. To display front panels on the Web, the VIs must be in memory on the computer.

The Web Server can generate images of front panels in the JPEG and PNG image formats. The JPEG format compresses graphics well but can lose some graphic detail. This format works best for photos. For line art, front panels, and block diagrams, JPEG compression can result in fuzzy graphics and uneven colors. The PNG format also compresses graphics well, although not always as well as the JPEG format. However, PNG compression does not lose any detail.

Use the Web Publishing Tool to create an HTML document and embed static or animated images of the front panel or to embed a front panel in an HTML document so a client computer can view and control the front panel remotely. You also can embed static images of the front panel in an existing HTML document. To deploy an HTML document with the application, enable and configure the Web Server in a stand-alone application.

Use the **Snapshot** option in the Web Publishing Tool to return a static image of the front panel of a VI currently in memory on the server computer. Use the **Monitor** option in the Web Publishing Tool to return an animated image of the front panel of a VI or application currently in memory on the server computer. Use the **Embedded** option in the Web

Publishing Tool to embed a front panel in an HTML document so a client computer can view and control the front panel remotely.



NOTE

The front panel of the VI or application you want to publish might be large. It takes the Web Server longer to create a large image, which can slow the execution of the VI. Also, a large image takes a significant amount of time to transmit to the browser, especially over a slow modem connection. Therefore, make sure the front panels of VIs and applications you want to publish are a reasonable size.

After you publish a VI or an application, you can control the application or front panel remotely or view either using a browser or using LabVIEW.

You can also share VIs with registered users on ni.com.



RELATED INFORMATION

[Enabling and Configuring the Web Server for Remote Front Panels](#) on page 1572

[Allowing and Denying Web Browser Access](#) on page 1570

[Specifying Which VIs Are Visible on the Web](#) on page 1579

[Creating HTML Documents and Embedding Front Panel Images](#) on page 1449

[Viewing an Application or Front Panel Remotely Using a Browser](#) on page 1582

[Embedding a Front Panel Image in an Existing HTML Document](#) on page 1450

[Enabling the Web Server in Stand-Alone Applications](#) on page 1509

[Controlling an Application or Front Panel Remotely Using a Browser or LabVIEW](#) on page 1571

[Viewing an Application or Front Panel Remotely Using a Browser](#) on page 1582

[Viewing an Application or Front Panel Remotely Using LabVIEW](#) on page 1583

Sending an Email from a VI

You can use the Send Email Express VI to quickly send emails from LabVIEW to a list of recipients you specify.

Complete the following steps to configure the Send Email Express VI to send an email:



NOTE

If you want to configure headers, set Transport Layer Security (TLS) settings, or include an attachment, you can use the standard SMTP Email VIs to configure and send an email with more advanced settings. Refer to the `labview\examples\Data Communication\Protocols\SMTP\Send Email using SMTP Client.vi` for an example of using the standard SMTP Email VIs to send an email.

1. Add the Send Email Express VI to the block diagram. The Configure Send Email dialog box automatically opens.
2. Specify your email address or the email address from which you want to send the email in the **Sender's Email Address** textbox located in the **User Information** section.
3. Specify the host name or IP address of the SMTP server, such as `my.mailserver.com`, you want to use to send the email in the **Outgoing Mail Server (SMTP)** textbox. If you do not know which mail server to use, contact the network administrator for the name of a valid server.
4. Specify the port number of the SMTP server you want to use to send the email in the **Port** textbox. The default is 25, which is standard for open connections. However, if you use a secure connection, you may need to change the port number.
5. (Optional) Place a checkmark in the **Require Log-in** checkbox to transmit log-in credentials to the SMTP server. Most public SMTP servers require log-in credentials.
 1. Specify the log-in username, in the **Username** textbox, for the SMTP server you specify. On most public servers, this is your email address or the email address from which you want to send the email.
 2. Specify the log-in password, in the **Password** textbox, for the SMTP server you specify.
6. (Optional) Place a checkmark in the **Use Secure Connection (TLS\SSL)** checkbox to require a TLS\SSL secured connection when LabVIEW communicates with the SMTP server.
7. Specify the email address(es) of the recipient(s) in the **Recipients** textbox located in the **Email** section. To send an email to multiple recipients, separate each email address with a comma. You must enter at least one email address. You also can wire a recipient list to the **Recipients** input on the block diagram.
8. (Optional) Specify the subject of the message in the **Subject** textbox. You also can wire a subject line to the **Subject** input on the block diagram.
9. (Optional) Specify a message in plain text, in the **Message** textbox, that you want to send to the recipient(s). You also can wire a message to the **Message** input on the block diagram.
10. (Optional) Click the **Send Test Email** button to test the configuration of the VI by sending a test email to the email address you specify in **Sender's Email Address**.
11. Click the **OK** button to save the configuration and close the Configure Send Email dialog box.
12. Wire all error in and error out terminals on the block diagram.
13. Run the VI. LabVIEW sends an email according to the configuration you set.

Setting Time Limits on Client Control

With the Web Server, you can set a time limit on how long a remote client can control a VI when multiple clients are waiting to control the VI. You should set a time limit to guarantee that every client has a chance to obtain control of the VI. When a particular client reaches

the time limit, the next client in the queue is the controller. However, all clients still can view the front panel.



NOTE

LabVIEW does not begin monitoring the time limit set on a particular VI until a second client requests control of the same VI. If a second client never requests control of the VI, then the initial client never loses control of the VI.

Complete the following steps to set the time limit on how long each client can control a particular VI.

1. Select **Tools»Options** to display the Options dialog box.
2. Select Web Server from the **Category** list.
3. Click a VI in the **Visible VIs** list in the **Visible VIs** section. If no VIs appear in the list, you need to [add visible VI entries](#) to the list and then select it.
4. Place a checkmark in the **Allow access** checkbox.
5. If you want to change the default time limit, remove the checkmark from the **Use default** checkbox. The default is 300 seconds.
6. In the **Control time limit (sec)** field, enter the amount of time in seconds you want to allow each client to control the front panel of the VI you selected in the **Visible VIs** list.
7. Click the **OK** button.

Specifying Which VIs Are Visible on the Web

Use the **Visible VIs** section of the Web Server page to configure which VIs are visible on the Web. Create a **Visible VIs** list that allows and denies access to individual VIs, groups of VIs, or directory paths. You also can use the VI Access List property to allow and deny access to VIs programmatically.

When a Web browser attempts to obtain a VI front panel image, the Web Server compares the VI name to the entries in the **Visible VIs** list to determine whether it should grant access. If an entry in the **Visible VIs** list matches the VI name, the Web Server permits or denies access to that VI image based on how you [set up the entry](#). By default, the front panel images of all VIs are visible. When you [add](#) new **Visible VIs** list entries, [edit](#) existing entries, or [remove](#) entries, use the correct [syntax, order, and wildcards](#) in the entries.

Syntax for Browser Address Lists

You can enter an IP address, such as 130.164.140.12, or a domain name, such as www.ni.com, in the **Browser Access List** in the **Browser Access** section on the Web Server page of the Options dialog box or the **Machine access list** in the VI Server page of the **Options** dialog box.

Use the * wildcard to specify a group of browser addresses. For example, you can specify all browsers within the domain domain.com with the entry *.domain.com. You can

specify all browsers in the subnet whose first two octets are 130 . 164 with the entry 130 . 164 . *. You can only use the * wildcard at the beginning of a domain name or at the end of an IP address.

The permission for an entry later in the **Browser Access List** or the **Machine access list** overrides previous permissions. You can click and drag an entry in the list to change the order.

For example, if you deny access to all browser addresses ending in *. test . site . com, but after that entry you give a . test . site . com and b . test . site . com access, those two browsers still have access, because the entries later in the list override the first entry.

In general, you should use the * wildcard to set up general allowances or denials, and follow those entries with more specific entries that reverse part of the previous permission. For best performance, place the most frequently matched entries toward the bottom of the list.

The following example illustrates how to use the * wildcard correctly.

Access List Entry	Permission Status	
✓	*	Allows access to all browser addresses.
✓	* . site . com	Allows access to all browser addresses ending with . site . com.
X	public . site . com	Denies access to this browser address even though previous entry allows access.
X	* . test . site . com	Denies access to any browser address ending with . test . site . com.
✓	a . test . site . com	Allows access to this browser address even though previous entry denies access.
X	130 . 164 . 123 . *	Denies access to all browser addresses whose IP addresses begin with 130 . 164 . 123 .
✓	130 . 164 . 123 . 123	Allows access to this browser address even though previous entry denies access.

Syntax for the VIs Lists

Each entry in the **Visible** and **Exported VIs** lists describes a VI name or a VI path and can contain wildcard characters. The Web and VI Servers compare entries that contain path separators against VI paths and compare entries that do not contain path separators against VI names only.

Use the following wildcard characters when specifying entries in the **Visible** and **Exported VIs** lists.

Wildcard	Action
?	Matches exactly one arbitrary character except for the path separator.
*	Matches zero or more arbitrary characters except for the path separator.
**	Matches zero or more arbitrary characters including the path separator.
\	Cancels the interpretation of special characters.

The permission for an entry later in the list overrides previous permissions. You can click and drag an entry in the list to change the order.

For example, if you deny access to all VI front panels in `c:\labview\server`, but after that entry you give access to `c:\labview\server\local.vi`, that VI is still visible, because the entry later in the list overrides the first entry.

In general, you should use the `*` wildcard to set up general allowances or denials, and follow those entries with more specific entries that reverse part of the previous permission. For best performance, place the most frequently matched entries toward the bottom of the list.

The following example illustrates how to use the wildcards correctly.

VIs List Entry	Permission Status	
✓	*	Allows access to all VIs.
✓	<code>c:\labview\server*</code>	Allows access to all VIs in <code>c:\labview\server</code> .
✓	<code>c:\labview\test**</code>	Allows access to all VIs in <code>c:\labview\test</code> and its subdirectories.
X	<code>c:\labview\test\private.vi</code>	Denies access to this VI even though the previous entry allows access.
✓	<code>srvr_*.vi</code>	Allows access to any VI that begins with the string <code>srvr_</code> and ends with the string <code>.vi</code> .
X	<code>Are You Finished\?.vi</code>	Denies access to <code>Are You Finished?.vi</code> .
X	<code>Open?.vi</code>	Denies access to all VIs named <code>Open?</code> where <code>?</code> represents one arbitrary character except for the path separator.

Viewing an Application or Front Panel Remotely Using a Browser

To view an application or front panel remotely using a Web browser, you must first configure the server. Complete the following steps to configure the server so a client computer can view an application front panel remotely using a browser.



TIP

For VIs that will be viewed or controlled remotely, create as small front panels as possible. The Web Server uses more system resources to load and display large front panel images than small front panel images, which can slow the execution of the VI itself.

1. Enable the Web Server on the server computer where the *application* or *VI* you want to view is located.
2. Configure the *browser access list* so the client computer can connect to the server.
3. Make sure the application is running or the VIs whose front panels you want clients to view are in memory on the server computer.
4. *Create an HTML file* using the Web Publishing Tool. You also can use an *existing HTML file*.
5. If you did not use the Web Publishing Tool, place the HTML file in the *labview\www* directory. Change the **LVFPPVINAME** parameter of the <OBJECT> and <EMBED> tags to the name of the top-level VI for the application or the top-level VI you loaded into memory. If you used the Web Publishing Tool, this step is completed automatically for you.

You also can change the WIDTH and HEIGHT parameters of the <OBJECT> and <EMBED> tags to customize the application or front panel appearance in the client browser.

Complete the following steps if you are on a client computer and do not have LabVIEW installed but want to view a front panel remotely using a browser.



NOTE

When you view a front panel remotely from a browser, the application or VI you want to view must be running on the server computer.

1. Install the *LabVIEW Run-Time Engine*.
2. Launch the default browser.
3. In the URL of the browser, type the IP address or computer name of the server followed by the name of the HTML file. For example, `http://ipaddress/filename.html`. If access is denied, an *error message* appears in the browser. You now can *request control* of the application or remote front panel.

To completely close the connection to the Web Server running on the server computer, click the close glyph on the browser window border.

You also can use the Application methods to open a connection and request control of a remote front panel programmatically.

Viewing an Application or Front Panel Remotely Using LabVIEW

To view an application or front panel remotely using LabVIEW, you must first configure the server. Complete the following steps to configure the server so a client computer can view a front panel remotely using LabVIEW.



TIP

For VIs that will be viewed or controlled remotely, create as small front panels as possible. The Web Server uses more system resources to load and display large front panel images than small front panel images, which can slow the execution of the VI itself.

1. Enable the Web Server on the server computer where the **VI** or **application** you want to view is located.
2. Configure the **browser access list** so the client computer can connect to the server.
3. Configure the **visible VIs list** so the client computer can access the VI or application front panels remotely. If you do not configure the visible VIs list, the client can access all VIs that are loaded into memory on the server computer.
4. Make sure the application is running or the VIs whose front panels you want clients to view are in memory on the server computer.

Complete the following steps if you are on a client computer and want to view a front panel remotely using LabVIEW.

1. Select **Operate»Connect to Remote Panel** to display the Connect to Remote Panel dialog box.
2. In the **Server address** field, enter the IP address or computer name of the server to which you want to connect.
3. In the **VI name** field, enter the name of the VI whose front panel you want to view. Or enter the name of the top-level VI of the application you want to view.
4. Enter the HTTP Port. The default is 80.
5. If you want to immediately request control of the front panel, place a checkmark in the **Request control** checkbox. You also can **request control** at the bottom of the remote front panel window when it appears on your computer.
6. Click the **Connect** button. The front panel appears on your computer allowing you to view it. If access is denied, an **error message** appears in the **Connection status** section of the **Connect to Remote Panel** dialog box.



NOTE

If you want to save the data generated by a VI running on a remote computer, use **DataSocket** or **TCP** instead of remote front panels.

To completely close your connection to the Web Server running on the server computer, click the close glyph on the LabVIEW window border.



Note The server computer managing remote front panels must have the LabVIEW Full or Professional Development System or must build the front panels into a stand-alone application. Clients can use any LabVIEW development system to view and control those front panels remotely.

You also can use the Application methods to open a connection and request control of a remote front panel programmatically.

Web Server Syntax Examples

Use the Web Publishing Tool to [embed a static or animated image of a front panel in an HTML document](#) or to embed a front panel in an HTML document so a client computer can [view and control the front panel remotely](#). To embed the static image of a front panel in an existing HTML document, enter the top-level VI name of the application in the following scenarios.

To embed the static image of a VI named `Example.vi` running on the same computer in an HTML document, use the following code in the HTML document:

```
<IMG SRC="/.snap?Example.vi">
```

To embed the static image of a VI named `Example.vi` running on the computer `foo` in an HTML document, use the following code in the HTML document:

```
<IMG SRC="http://foo/.snap?Example.vi">
```

To embed the static image of a VI named `Test 1.vi` running on the computer `foo` using image **type** = png and image **depth** = 24, use the following code in the HTML document:

```
<IMG SRC="http://foo/.snap?Test%201.vi&type=png&depth=24">
```

Creating a TCP Client

Complete the following steps to create a TCP client using the TCP functions.

1. (Optional) Create a TCP server. When you create a TCP server, you specify the remote port or service name to represent the port you want the server to use for communication.
2. Use the TCP Open Connection function to open a connection to a server. You must specify the **remote port or service name** for the server. The **remote port or service name** identifies a communication channel on the computer that the server uses to listen for communication requests. If you specify a service name, LabVIEW queries the NI Service Locator for the port number that the server registered. If you want to

establish a connection to a remote computer, you must specify the Internet address of the server. The **address** identifies a computer on the network. If you do not specify an address, LabVIEW establishes a connection to the local computer.

3. Use the TCP Read function to read a message from the server. You must specify the number of bytes you want to read.
4. Use the TCP Write function to send a message to a server.
5. Use the TCP Close Connection function to close the connection to the server.

Refer to the following VIs for examples of a TCP client:

- `labview\examples\Data Communication\Protocols\TCP\TCP Named Service\TCP Named Service.lvproj`



RELATED INFORMATION

[Creating a TCP Server](#) on page 1585

[NI Service Locator](#) on page 750

Creating a TCP Server

Complete the following steps to create a TCP server using the TCP functions.

1. Use the TCP Listen VI or the TCP Create Listener and TCP Wait On Listener functions to wait for a connection. You must specify the port and optionally a service name. This port or service name must be the same port or service name that the client attempts to access.
 - TCP Listen VI
 - TCP Create Listener Function
 - TCP Wait On Listener Function
2. If a connection is established, use the TCP Write function to send a message to the client. The data must be in a form that the client can accept.
3. Use the TCP Read function to read a message from the client.
4. Use the TCP Close Connection function to close the connection.

Refer to the following VIs for examples of a TCP server:

- `labview\examples\Data Communication\Protocols\TCP\TCP Named Service\TCP Named Service.lvproj`



RELATED INFORMATION

[Creating a TCP Client](#) on page 1584

Domains

Using the Domain Account Manager (Windows)

Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**. Use the **Domain Account Manager** to *create* and *destroy* a local domain, *export and import a local domain*, *add and remove network domains*, *import a Lookout account*, and manage domain *users* and *groups*.

The **Domain Account Manager** includes the following components:

- **My Computer** —Displays the local domain.
- **Network Domains** —Lists network domains the administrator configures.
- **User list** —Lists users in the selected domain.
- **Group list** —Lists groups in the selected domain.



(Windows) To view related topics, click the **Locate** button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the **Contents** tab so you can navigate the related topics.

Adding a Network Domain (Windows)

Use the **Domain Account Manager** to *create* and *destroy* a local domain, *export and import a local domain*, add and remove network domains, *import a Lookout account*, and manage domain *users* and *groups*.

Complete the following steps to use the **Domain Account Manager** to add a network domain to the network domain list.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.
2. Select **Domain»Browse for Domains** or right-click the network domain list and select **Browse for Domains** from the shortcut menu to display the Browse for Domains dialog box.
3. Select the network domain you want to manage.
4. Click the **OK** button to close the dialog box. LabVIEW adds the network domain to the network domain list.

You also can find a network domain by entering the name or IP address of the host machine in the Find Domain dialog box. Select **Domain»Find Domain** or right-click the network domain list and select **Find Domain** from the shortcut menu to display the **Find Domain** dialog box.

To remove a network domain from view, click the network domain and select **Edit»Remove From View** or right-click the domain and select **Remove From View** from the shortcut menu. Removing a domain from view does not *destroy the domain*.

Creating a Local Domain (Windows)

Use the *Domain Account Manager* to create and *destroy* a local domain, *export and import a local domain*, *add and remove network domains*, *import a Lookout account*, and manage domain *users* and *groups*.

Complete the following steps to use the **Domain Account Manager** to create a local domain on a computer.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.



NOTE

If a local domain already exists on the computer, the **Domain Account Manager** loads the local domain automatically. To create a new local domain, first destroy the existing local domain.

2. Select **File»New»Local Domain** or right-click the domain list and select **New Local Domain** from the shortcut menu to display the Domain Properties dialog box.
3. Click the **General** tab and enter a name for the domain in the **Domain** text box.
4. Click the **Passwords** tab and configure the minimum and maximum lengths of passwords and the password expiration date.



Note The password length and expiration date apply as the default setting for all user accounts in the domain.

1. Click the **Access Control** tab and add machines or IP addresses to the **Grant List** or **Deny List**. **Select list to activate** specifies which list the domain is currently using.
2. Click the **OK** button to display the Change Password dialog box.
3. Enter a password for the Administrator account and click the **OK** button. By default, the user name for the account that you are creating is Administrator.



Note The **Domain Account Manager** creates a list of built-in users and groups when you create a local domain. The built-in users include Administrator and Guest, and the built-in groups include Administrators, Guests, and Operators.

Creating and Configuring Domain Groups (Windows)

Use the *Domain Account Manager* to *create* and *destroy* a local domain, *export and import a local domain*, *add and remove network domains*, *import a Lookout account*, and manage domain *users* and *groups*.

Complete the following steps to use the **Domain Account Manager** to create, configure, and delete domain groups. You must be *logged in* to the domain to create, configure, or delete groups.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.
2. Click the group list and select **File»New»Group** or right-click the group list and select **New** from the shortcut menu to display the Group Properties dialog box.
3. Enter the group name in the **Group name** text box.
4. Enter a description in the **Description** text box.
5. Add a user to the group by selecting the user in the **Users not in group** list and clicking the right arrow button. To remove a user from the group, select a user in the **Users in group** list and click the left arrow button.
6. Click the **OK** button.

You also can configure domain group properties with the **Group Properties** dialog box. Click the group and select **Edit»Properties** or right-click the group and select **Properties** from the shortcut menu to display the **Group Properties** dialog box.

To delete a domain group, click the group and select **Edit»Delete** or right-click the group and select **Delete** from the shortcut menu. You cannot delete a built-in group. Deleting a group invalidates all previous groups by that name, and entries in the Access Control List might become invalid. Even if you recreate a group with the same name, you must add user accounts to the new group manually.

Creating and Configuring Domain Users (Windows)

Use the *Domain Account Manager* to *create* and *destroy* a local domain, *export and import a local domain*, *add and remove network domains*, *import a Lookout account*, and manage domain users and *groups*.

Complete the following steps to use the **Domain Account Manager** to create, configure, and delete domain users. You must be *logged in* to the domain to create, configure, or delete users.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.
2. Click the user list and select **File»New»User** or right-click the user list and select **New** from the shortcut menu to display the User Properties dialog box.
3. Click the **General** tab and enter the user name in the **User name** text box.
4. Enter the user's full name in the **Full name** text box.
5. Enter a description in the **Description** text box.
6. Place a checkmark in the **Account is locked** checkbox if you want to lock a user account without removing the user from the domain.
7. Click the **Change Password** button to display the Change Password dialog box.
8. Enter and re-type a password for the user and click the **OK** button.

9. Click the **Memberships** tab and add the new user to a group by selecting the group in the **Groups user can join** list and clicking the right arrow button. To remove the user from a group, select a group in **User is member of these groups** list and click the left arrow button.
10. Click the **OK** button.

You also can configure domain user properties. Click the user and select **Edit»Properties** or right-click the user and select **Properties** from the shortcut menu to display the **User Properties** dialog box and configure user settings.

To delete a domain user, click the user and select **Edit»Delete** or right-click the user and select **Delete** from the shortcut menu. You cannot delete a built-in user. Deleting a user invalidates all previous users by that name, and entries in the Access Control List might become invalid. Even if you recreate a user with the same name, the internal user identification number is different.

Destroying a Local Domain (Windows)

Use the Domain Account Manager to create and destroy a local domain, export and import a local domain, add and remove network domains, import a Lookout account, and manage domain users and groups.

Complete the following steps to use the **Domain Account Manager** to destroy a local domain. You must log in to the local domain before you can destroy it.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.
2. Click the local domain and select **Domain»Destroy Local Domain** or right-click the local domain and select **Destroy Local Domain** from the shortcut menu.
3. When a dialog box prompts you to confirm that you want to destroy the domain, click the **Yes** button.
4. Enter the password in the **Enter Administrator Password** dialog box and click the **OK** button.

If you cannot destroy a domain through the Domain Account Manager, such as if you do not have the password for the domain, you can delete the domain directly from the **National Instruments/Shared/Domain** directory. You must log on to the

computer as an administrator to delete files from the National Instruments directory.



RELATED INFORMATION

[Using the Domain Account Manager \(Windows\)](#) on page 1586

[Creating a Local Domain \(Windows\)](#) on page 1587

[Exporting and Importing a Local Domain Using the Domain Account Manager \(Windows\)](#) on page 1590

[Adding a Network Domain \(Windows\)](#) on page 1586

[Importing a Lookout Account \(Windows\)](#) on page 1591

[Creating and Configuring Domain Users \(Windows\)](#) on page 1588

[Creating and Configuring Domain Groups \(Windows\)](#) on page 1587

[Logging In to a Domain \(Windows\)](#) on page 1592

[Using the Domain Account Manager \(Windows\)](#) on page 1586

Exporting and Importing a Local Domain Using the Domain Account Manager (Windows)

Use the [Domain Account Manager](#) to [create](#) and [destroy](#) a local domain, export and import a local domain, [add and remove network domains](#), [import a Lookout account](#), and manage domain [users](#) and [groups](#).

Exporting a Local Domain

Complete the following steps to use the **Domain Account Manager** to export a local domain. You must [log in](#) to the local domain before you can export it.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.
2. Select **File»Import and Export»Export Local Domain To File** or right-click the local domain and select **Import and Export»Export Local Domain To File** from the shortcut menu. A file dialog box appears.
3. Navigate to the location where you want to export the local domain.
4. Enter a filename for the domain and click the **Save** button.
5. Enter the password in the **Enter Administrator Password** dialog box and click the **OK** button.

Importing a Local Domain

Complete the following steps to use the **Domain Account Manager** to import a local domain.



NOTE

You cannot import a domain that already exists on the network.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.
2. Select **File»Import and Export»Import Local Domain From File** or right-click the domain list and select **Import and Export»Import Local Domain From File** from the shortcut menu. A file dialog box appears.
3. Select the domain you want to import and click the **Open** button.

Note If a local domain already exists on the computer, a dialog box prompts you to save a copy of the existing local domain. Click the **Yes** button to save a copy of the existing local domain before you import the new domain. Click the **No** button to replace the existing local domain with the new domain. Click the **Cancel** button to cancel the import.

4. Enter the password in the **Enter Administrator Password** dialog box and click the **OK** button.

Importing a Lookout Account (Windows)

Use the **Domain Account Manager** to *create* and *destroy* a local domain, *export and import a local domain*, *add and remove network domains*, import a Lookout account, and manage domain *users* and *groups*.

Complete the following steps to use the **Domain Account Manager** to import a Lookout account.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.
2. Select **File»New»Import Lookout Security File** or right-click the domain list and select **Import Lookout Security File** from the shortcut menu to display the Domain Properties dialog box.



NOTE

If a local domain already exists on the computer, a dialog box prompts you to save a copy of the existing local domain. Click the **Yes** button to save a copy of the existing local domain before you import the Lookout account. Click the **No** button to replace the existing local domain with the Lookout account. Click the **Cancel** button to cancel the import.

3. Click the **General** tab and enter a name for the domain in the **Domain** text box.
4. Click the **Passwords** tab and configure the minimum and maximum lengths of passwords and the password expiration date.



Note The password length and expiration date apply as the default setting for all user accounts in the domain.

1. Click the **Access Control** tab and add machines or IP addresses to the **Grant List** or **Deny List**. **Select list to activate** specifies which list the domain is currently using.
2. Click the **OK** button to display the Change Password dialog box.
3. Enter a password for the Administrator account and click the **OK** button.
4. Enter the default password for all users and click the **OK** button.

Logging In to a Domain (Windows)

Use the Domain Account Manager to create and destroy a local domain, export and import a local domain, add and remove network domains, import a Lookout account, and manage domain users and groups.

Complete the following steps to use the **Domain Account Manager** to log in to a domain.

1. Select **Tools»Security»Domain Account Manager** to display the **Domain Account Manager**.
2. Click the domain you want to log in to and select **Domain»Log In** or right-click the domain and select **Log In** from the shortcut menu to display the Log In to Domain dialog box.
3. Enter the user name and password and click the **Log In** button.



NOTE

The login user must be an Administrator or a member of the Administrators group.

To log out of a domain, click the domain and select **Domain»Log Out** or right-click the domain and select **Log Out** from the shortcut menu.



RELATED INFORMATION

[Using the Domain Account Manager \(Windows\)](#) on page 1586

[Creating a Local Domain \(Windows\)](#) on page 1587

[Destroying a Local Domain \(Windows\)](#) on page 1589

[Exporting and Importing a Local Domain Using the Domain Account Manager \(Windows\)](#) on page 1590

[Adding a Network Domain \(Windows\)](#) on page 1586

[Importing a Lookout Account \(Windows\)](#) on page 1591

[Creating and Configuring Domain Users \(Windows\)](#) on page 1588

[Creating and Configuring Domain Groups \(Windows\)](#) on page 1587

LabVIEW Measurement Concepts

This section provides instructions for performing measurements and controlling instruments with LabVIEW.

Overview of Virtual Instruments and Measurements

Virtual instrumentation combines hardware and software with industry-standard computer technologies to create user-defined instrumentation solutions. National Instruments specializes in developing plug-in and distributed hardware and driver software for data acquisition (DAQ), IEEE 488 (GPIB), PXI, serial, and industrial communications. The driver software is the application programming interface (API) to the hardware and is consistent across National Instruments application software, such as LabVIEW, LabWindows™/CVI™, and Measurement Studio. These platforms deliver the sophisticated display and analysis capabilities that virtual instrumentation requires.

You can use virtual instrumentation to create a complete and customized system for test, measurement, and industrial automation by combining different hardware and software components. If the system changes, you often can reuse the virtual instrument components without purchasing additional hardware or software.

Taking Measurements with Virtual Instruments

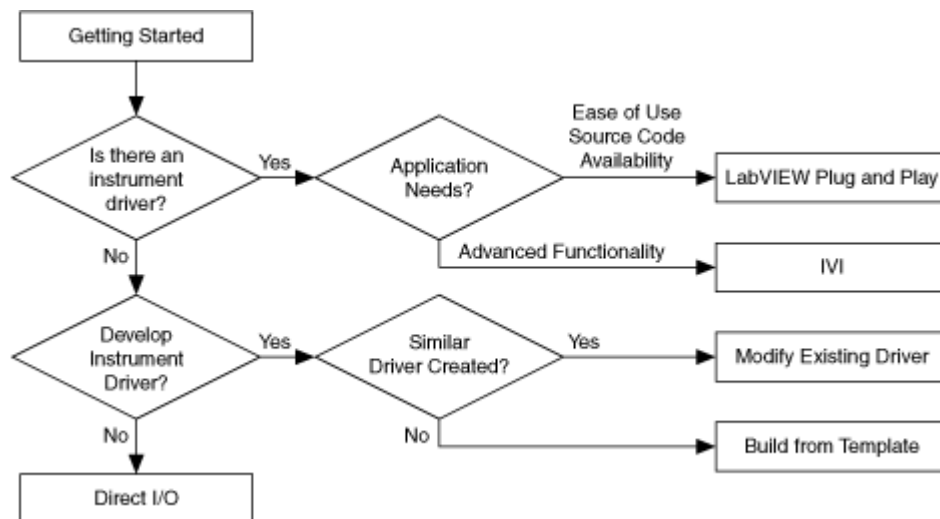
Different hardware and software components can make up a virtual instrumentation system. You can use a variety of hardware components to monitor or control a process or to test a device. If you can connect the hardware to the computer and understand how the hardware takes measurements, you can incorporate hardware into a virtual instrumentation system.

Controlling Instruments

You control instruments by sending commands and data between the instrument and the computer. You can develop various LabVIEW applications to configure and control many *types of instruments*.

Choosing Method of Instrument Control

Due to the wide array of instruments and instrument control interfaces, it is important to choose the proper method of instrument control. The following flowchart can help you choose the appropriate method of instrument control.



Using Instrument Drivers

Use **instrument drivers** to control and communicate with instrument hardware in a system. LabVIEW instrument drivers are sets of VIs that simplify instrument control and reduce test program development time by eliminating the need for you to learn the complex, low-level programming commands for each instrument.

When you need easy-to-use drivers with block diagram source code, use **LabVIEW Plug and Play** instrument drivers. LabVIEW Plug and Play instrument drivers include error handling, front panels, block diagrams, icons, and help. (Windows) When you need advanced functionality such as instrument interchangeability, simulation, and state-caching, use Interchangeable Virtual Instruments (IVI) instrument drivers. Refer to IVI on the National Instruments Web site for more information about IVI instrument drivers.

Instruments and DAQ Devices

General-Purpose DAQ

A general-purpose DAQ device is a device that acquires or generates data and can contain multiple channels. You also can use general-purpose DAQ devices to generate analog signals, such as a sine wave, and digital signals, such as a pulse. Typically, you connect these devices directly to the internal bus of a computer through a plug-in slot.

A general-purpose DAQ measurement system is different from other measurement systems because the software installed on the computer performs the actual measurements. The DAQ device only converts the incoming signal into a digital signal the computer can use. This means that the same DAQ device can perform a multitude of measurements simply by changing the software application that reads the data. In addition to acquiring the data, the application for a DAQ measurement system also uses the software that processes the data and displays the results. Although this flexibility allows you to have one hardware device for many types of measurements, you must spend more time developing the different applications for the different types of measurements. LabVIEW includes many acquisition and analysis functions to help you develop different applications.

How Computers Communicate with DAQ Devices

Before a computer-based measurement system can measure a physical signal, such as temperature, a sensor or transducer must convert the physical signal into an electrical one, such as voltage or current. The plug-in DAQ device is only one system component of the entire measurement system. You cannot always directly connect signals to a plug-in DAQ device. In these cases, you must use signal conditioning accessories to condition the signals before the plug-in DAQ device converts them to digital information. The software controls the DAQ system by acquiring the raw data, analyzing, and presenting the results.

A DAQ system has the following options:

- The plug-in DAQ device resides in the computer. You can plug the device into the PCI slot of a desktop computer or the PCMCIA slot of a laptop computer for a portable DAQ measurement system.
- The DAQ device is external and connects to the computer through an existing port, such as the serial port or Ethernet port, which means you can quickly and easily place measurement nodes near sensors.

Software and DAQ Devices

The computer receives raw data through the DAQ device. The application software you write presents and manipulates the raw data in a form you can understand. The software also controls the DAQ system by commanding the DAQ device when to acquire data from channels, and from which channels to acquire the data.

Typically, DAQ software includes drivers and application software. Drivers are unique to the device or type of device and include the set of commands the device accepts. Application software, such as LabVIEW, sends the drivers commands, such as a command to acquire and return a thermocouple reading. The application software also displays and analyzes the acquired data.

NI measurement devices include NI-DAQ driver software and NI Measurements VIs and functions that you can use to configure, acquire data from, and send data to the measurement devices.

NI-DAQ

NI-DAQ 7. x contains two NI-DAQ drivers—Traditional NI-DAQ (Legacy) and NI-DAQmx—each with its own application programming interface (API), hardware configuration, and software configuration. NI-DAQ 8.0 and later come with only NI-DAQmx, the replacement for Traditional NI-DAQ (Legacy).

- Traditional NI-DAQ (Legacy) is an upgrade to NI-DAQ 6.9. x, the earlier version of NI-DAQ. Traditional NI-DAQ (Legacy) has the same VIs and functions and works the same way as NI-DAQ 6.9. x. You can use Traditional NI-DAQ (Legacy) on the same computer as NI-DAQmx, which you cannot do with NI-DAQ 6.9. x. However, you cannot use Traditional NI-DAQ (Legacy) on Windows Vista.
- NI-DAQmx is the latest NI-DAQ driver with new VIs, functions, and development tools for controlling measurement devices. The advantages of NI-DAQmx over

previous versions of NI-DAQ include the DAQ Assistant for configuring channels and measurement tasks for a device; increased performance, including faster single-point analog I/O and multithreading; and a simpler API for creating DAQ applications using fewer functions and VIs than earlier versions of NI-DAQ.

**NOTE**

(Windows) LabVIEW supports NI-DAQmx and the DAQ Assistant. (Mac OS X) LabVIEW supports NI-DAQmx Base but not the DAQ Assistant. (Linux) LabVIEW supports NI-DAQmx but not the DAQ Assistant.

Traditional NI-DAQ (Legacy) and NI-DAQmx support different sets of devices. Refer to Data Acquisition (DAQ) Hardware on the National Instruments Web site for the list of supported devices.

Instrument I/O

Many instruments are external to the computer and do not rely on a computer to take a measurement. By connecting instruments to a computer, you can control and monitor the instruments programmatically and collect data that you can process further or store in files. You can install some instruments in a computer similar to general-purpose DAQ devices. These internal instruments are called modular instruments.

Regardless of how you connect to an instrument, the computer must use a specific protocol to communicate with the instrument. How the computer controls the instrument and acquires data from the instrument depends on the type of the instrument. GPIB, serial port, and PXI are common types of instruments.

Like general-purpose DAQ devices, instruments digitize data, but they have a special purpose or are designed for a specific type of measurement. For standalone instruments, you generally cannot modify the software that processes the data and calculates the result because the software usually is built into the instrument.

Because modular instrumentation uses software running on standard PC technology, you can modify the behavior of these instruments more easily. For example, with some digital multimeter modular instruments, you can program the instruments to acquire a buffer of data at a high rate of speed, much like an oscilloscope.

How Computers Control Instruments

Computers control instruments by sending commands to the instruments over a bus, such as GPIB, PXI, or RS232. For example, you can send a command to the instrument to measure a signal and then send another command to send the resulting data over the bus back to the computer.

Instrument Drivers

Instrument drivers are not required to use an instrument, but can help save you time as you develop an application.

Using instrument drivers provides the following advantages:

- You can use instrument drivers to quickly build complete systems. Instrument drivers receive, parse, and scale the response strings from instruments into scaled data that you can use in test programs.
- Instrument drivers can reduce software development costs because developers do not need to spend time understanding low-level command syntax needed to program instruments.
- Instrument drivers can help make test programs easier to maintain because instrument drivers contain all the I/O for an instrument within one library, separate from other code, which is easy to upgrade when or if the hardware changes.

Waveform Control and Digital Waveform Control

Use the waveform control, the digital waveform control, the waveform graph, and the digital waveform graph to represent the waveforms and digital waveforms you acquire or generate. LabVIEW represents an analog waveform, such as a sine wave or a square wave, with the waveform data type by default. A 1D array of waveform data type represents multiple waveforms. LabVIEW represents a digital waveform with the digital waveform data type by default.

The waveform and digital waveform controls consist of components that include a start time, a delta t , the waveform data, and attributes. Use the Waveform VIs and functions to access and manipulate individual components.

The start time (t_0) is a time stamp associated with the first measurement point in the waveform. Use the start time to synchronize plots on a multi-plot waveform graph or digital waveform graph and to determine delays between waveforms. Delta t (dt) is the time interval in seconds between any two points in the signal. The waveform data and the digital waveform data are the values that represent the waveform. An array of any numeric data type other than the fixed-point numeric data type can represent analog waveform data. Generally, the number of data values in the array corresponds directly to the number of scans from a DAQ device. The digital data type represents a digital waveform and displays the digital data in a table. Attributes include information about the signal, such as the name of the signal and the device acquiring the signal. NI-DAQ automatically sets some attributes for you. Use the Set Waveform Attribute function to set attributes, and use the Get Waveform Attribute function to read attributes.



NOTE

If you convert the waveform data type to the *dynamic data type*, you can set and read attributes of the dynamic data by using the Set Dynamic Data Attributes Express VI and the Get Dynamic Data Attributes Express VI, respectively. These Express VIs automatically convert the dynamic data type to the waveform data type before you can set and read attributes. The dynamic data type is for use with Express VIs.

Displaying Waveforms

To represent waveform data in the front panel window, use the waveform control or the waveform graph. To represent digital waveform data, use the digital waveform control or the digital waveform graph.

Use the waveform control and the digital waveform control to manipulate the t_0 , dt , and Y components of the waveform or to display those components as an indicator. When you wire a waveform to a graph, the t_0 component is the initial value on the x-axis. The number of scans acquired and the dt component determine the subsequent values on the x-axis. The data elements in the Y component comprise the points on the plot of the graph.

If you want to let a user control a certain component, such as the dt component, create a front panel control and wire it to the appropriate component in the Build Waveform (Analog Waveform) function.

If you want to organize digital waveform data into a digital waveform array, use the Group Digital Signals Express VI.

Using the Waveform Control

You can use several VIs to accept, operate on, and/or return waveforms. In addition, you can wire the waveform data type directly to many controls, including the graph, chart, numeric controls, and numeric array controls.

The AI Acquire Waveform VI acquires a specified number of samples at a specified sample rate at a particular time from a single input channel and returns a waveform. The Digital IIR Filter VI accepts an array of waveforms and filters the data of each waveform. The AI Sample Channel VI acquires a single sample from a channel and returns a single-point waveform. The Sine Waveform VI generates a sine waveform, and the AO Generate Waveform VI sends a waveform to a device.

Use the Get Waveform Components (Analog Waveform) function to extract and manipulate the components of a waveform you generate. The Negate function negates waveform data and plots the results to a graph.

Using the Digital Waveform Control

Use the Digital Waveform VIs and functions to manipulate digital data by extracting and editing the components of the digital signal. Use the NI-DAQmx VIs on the **Digital I/O** palette to acquire and send a digital signal. The **Digital Waveform** palette also includes VIs that convert analog data to digital signals, search a digital signal for a pattern, append a digital signal(s) to another digital signal, and perform other digital tasks.

Refer to the Taking an NI-DAQmx Measurement in LabVIEW tutorial for more information about creating typical DAQ applications.

Data and String Manipulation Techniques

Communicating with instruments involves sending data to instruments and retrieving data from instruments. You rarely need to format the data sent to or retrieved from the instrument when using instrument drivers because the instrument driver handles the formatting for you. However, you might need to format the data when writing VIs to communicate with instruments.

When you communicate with a message-based instrument, you must format and build the correct command strings for the instrument to perform the appropriate operation or return a response.

No standards exist for register-based instrument communication. Each device operates differently, and the instrument documentation is the best resource for learning how to program the device.

Typically, a command string, or query, is a combination of text and numeric values. Some instruments require text-only command strings, requiring you to *convert the numeric values to text* and append them to the command string. Similarly, to use the data an instrument returns in LabVIEW, you must convert the data to a format a VI, function, or indicator can accept.

(Windows) You can use the Instrument I/O Assistant to send queries to an instrument and to format the data an instrument returns. Place the Instrument I/O Assistant Express VI on the block diagram to access the Instrument I/O Assistant.

You can use the Format Into String function to build command strings to send to an instrument or to append other strings or numeric data types to an initial string.

Formatting Data Retrieved from an Instrument

Just as the commands strings you send to an instrument contain headers and trailers, most instruments return data with headers and/or trailers. The header usually contains information such as the number of data points returned or the instrument settings. Trailer information often contains units or other instrument settings at the end of the data string. The instrument documentation should describe what header and trailer information to expect from each data transfer. You must first remove the header and trailer information before you can display or analyze the returned data in LabVIEW.

You can use String Subset functions to parse the header and trailer information in a string and display them in string indicators. You can use the Scan From String function to parse the data from the string.

Waveform Transfers

Instruments also can return data in other formats, such as ASCII, 1-byte binary, and 2-byte binary formats. The instrument documentation describes available formats and how to convert each one to usable data.

ASCII Waveforms

If an instrument returns data in ASCII format, you can view the data as a character string. However, if you need to manipulate the data in numeric format or if you need to graph the data, you must convert the string data to numeric data. As an example, consider a waveform composed of 1,024 points and in which each point has a value between 0 and 255. Using ASCII encoding, you need a maximum of 4 bytes to represent each data value (a maximum of 3 bytes for the value and 1 byte for the separator, such as a comma). You need a maximum of 4,096 bytes (4 bytes 1,024) plus any header and trailer bytes to represent the waveform as an ASCII string. You can use the Fract/Exp String To Number function to convert the ASCII string to a numeric array.

1-Byte Binary Waveforms

Some instruments do not have the option of sending data in ASCII format or send all waveform data in binary format. Because no standard exists for binary format, refer to the instrument documentation to determine exactly how the instrument stores the data values. One common binary format is 1-byte binary. With this type of data encoding, the instrument converts each piece of data to an 8-bit binary value before sending it.

When you read 1-byte binary data from the bus, the instrument returns the data as a character string. However, the characters do not appear to correspond to the expected data. LabVIEW interprets the binary numbers as ASCII character values, and displays the corresponding characters. For example, if you send the value of 65 as one data value, you read the character A from the bus. For a value of 13, no printable ASCII character exists because 13 corresponds to an invisible carriage return character.

You can display these invisible characters in a string indicator by right-clicking the indicator and selecting **'\Codes Display** from the shortcut menu. The carriage return character is `\r` in the string indicator.

To use the numeric data in an ASCII string with the Analysis VI or to display the numeric data in a graph or chart, you must convert the binary string to a numeric array. If the instrument sends a binary string that contains 1,024 1-byte binary encoded values, the waveform requires only 1,024 bytes plus any header information. Using binary encoding, you need only 1 byte to represent each data value, assuming each value is an unsigned 8-bit integer. You can use String Subset functions and the String To Byte Array function to convert the binary string to an array of integers.



NOTE

When acquiring binary data, extract the data using the data size rather than searching for the first character of the trailer information because that character might be part of the binary values.

2-Byte Binary Waveforms

When data is in 2-byte binary format, it is binary encoded and sent as ASCII characters similar to the 1-byte binary format. However, 16 bits of data, or two ASCII characters, represent each data value. Although this format uses twice as much space as the 1-byte binary format, it is more efficiently packed than ASCII-formatted data.

As an example, consider an oscilloscope that transfers waveform data in binary notation. For this example, assume the waveform consists of 1,024 data points where each value is a 2-byte signed integer. Therefore, the entire waveform requires 2,048 bytes plus a 5-byte header and a 2-byte trailer. After you remove the 5-byte header, use the Unflatten From String function to convert the waveform string to an array of 16-bit integers.

It is important to know the order of the bytes you receive when data is transferred in 2-byte binary format. The 2-byte combination qH has the corresponding integer value of 29,000, but the opposite byte order of Hq has the corresponding integer value of 18,545.



NOTE

You can configure the byte order for some instruments, and some instruments have a fixed byte order. Refer to the instrument documentation for specific instrument byte information.

If you receive the high byte first, you must reverse the order of the bytes before you convert them to an integer value. Use the Unflatten From String to reverse the byte order.

Comparing DAQ Devices and Computer-Based Instruments for Data Acquisition

The fundamental task of all measurement systems is the measurement and/or generation of real-world physical signals. Measurement devices help you acquire, analyze, and present the measurements you take.

Through data acquisition, you acquire and convert physical signals, such as voltage, current, pressure, and temperature, into digital formats and transfer them into the computer. Popular methods for acquiring data include plug-in [DAQ](#) and [instrument](#) devices, GPIB instruments, PXI (PCI eXtensions for Instrumentation) instruments, and RS232 instruments.

Through data analysis, you transform raw data into meaningful information by using curve fitting, statistical analysis, frequency response, or other numerical operations.



NOTE

Many of these operations are available only in the LabVIEW Full and Professional Development Systems.

Through data presentation, you display data in a graph, thermometer, table, or other visual display.

Configuring DAQ Devices, Instruments, and Other Devices

You can configure DAQ devices, instruments, and other devices to control the device or instrument using an [I/O control](#). After you configure the device or instrument, configure the I/O control to pass Traditional NI-DAQ (Legacy) virtual channels, NI-DAQmx resources (such

as tasks, channels, and scales), IVI logical names, VISA resource names, FieldPoint I/O points, or Motion resource names to I/O VIs to communicate with an instrument or a measurement device.

The NI-DAQ driver software provides LabVIEW with a high-level interface to National Instruments DAQ devices. Driver software, such as NI-DMM, NI-SCOPE, NI-FGEN, and NI-SWITCH, provides LabVIEW with high-level interfaces to modular instrumentation. Use driver software such as NI-488.2, NI-VISA, and Interchangeable Virtual Instruments (IVI) to communicate with stand-alone instruments. The LabVIEW VIs call into the driver software, which communicates with the measurement devices. Refer to the DAQ Getting Started Guide for more information about installing NI-DAQmx.

Configuring Devices and Instruments Using MAX and the DAQ Assistant (Windows)

You can use MAX to configure the following items:

- Traditional NI-DAQ (Legacy) virtual channels
- IVI logical names
- VISA resource names
- FieldPoint I/O points
- Motion resource names



NOTE

You can access MAX on multiple platforms, but LabVIEW supports the DAQ Assistant on Windows only. You cannot use Traditional NI-DAQ (Legacy) on Windows Vista.

To launch MAX, you can select **Tools»Measurement & Automation Explorer** or double-click the **NI MAX** icon on the desktop. You also can create a new Traditional NI-DAQ (Legacy) virtual channel from the front panel by right-clicking the traditional DAQ channel control and selecting **New Channel** from the shortcut menu.



NOTE

The **Measurement & Automation Explorer** option is available only if you install Measurement & Automation Explorer.

You can right-click an IVI logical name, FieldPoint IO point, VISA resource name, or Motion resource name control and select **Allow Undefined Names** from the shortcut menu to enter logical names you have not configured in MAX or another configuration utility. You also can use the Allow Undefined Names property to programmatically configure the IVI logical name control to accept undefined IVI logical names.

You can right-click the VISA resource name or Motion resource name control and select **Allow Multiple Names** from the shortcut menu to configure the controls to allow the user to enter more than one resource name.

The DAQ Assistant allows you to create and configure the following for NI-DAQmx.

- Tasks
- Global channels
- Scales

Refer to the Using NI-DAQmx Name Controls book for more information about configuring tasks, channels, and scales for NI-DAQmx.

Configuring Devices and Instruments Using Other Utilities (Mac OS X and Linux)

You can use the configuration utilities for the instrument. Refer to the instrument documentation for information about the configuration utilities. You also can configure I/O controls to pass VISA resource names that have not been configured.

Types of Instruments

Types of Instruments

When you use a PC to automate a test system, you are not limited to the type of instrument you can control. You can mix and match instruments from various categories. The most common categories of instrument interfaces are [GPIB](#), [serial](#), and [modular instruments](#). Additional types of instruments include image acquisition, motion control, [USB](#), [Ethernet](#), parallel port, NI-CAN, and other devices.

When you use PCs to control instruments, you need to understand properties of the instrument, such as the communication protocols to use. Refer to the instrument documentation for information about the properties of an instrument.

GPIB Communication

GPIB, or General Purpose Interface Bus, instruments offer test and manufacturing engineers the widest selection of vendors and instruments for general-purpose to specialized vertical market test applications. GPIB instruments are often used as stand-alone benchtop instruments where measurements are taken by hand. You can automate these measurements by using a PC to control the GPIB instruments.

Use [instrument drivers](#) and the VISA VIs to control GPIB instruments. When you use the VISA VIs, you need to know the instrument address. Use the VISA resource name control or MAX on Windows to verify communication with the instrument and determine the instrument address. You also need to know the command set for the instrument. Most instruments use SCPI, or Standard Commands for Programmable Instruments: a single, comprehensive command set suitable for all instruments. Refer to the NI Developer Zone for more information about common instrument commands and protocols.

The GPIB protocol categorizes devices as controllers, talkers, or listeners to determine which device has active control of the bus. Each device has a unique GPIB primary address between 0 and 30. The Controller defines the communication links, responds to devices

that request service, sends GPIB commands, and passes/receives control of the bus. Controllers instruct Talkers to talk and to place data on the GPIB. You can address only one device at a time to talk. The Controller addresses the Listener to listen and to read data from the GPIB. You can address several devices to listen.

The GPIB is a digital, 24-conductor parallel bus. The GPIB uses an 8-bit parallel, byte-serial, asynchronous data transfer scheme, which means that whole bytes are sequentially handshaked across the bus at a speed that the slowest participant in the transfer determines. Because the unit of data on the GPIB is a byte, the messages transferred are frequently encoded as ASCII character strings. Refer to the GPIB documentation for more information about the hardware specifications of GPIB devices.

You can obtain faster data rates with HS488 devices and controllers. HS488 is an extension to GPIB that most NI controllers support.

Refer to the NI Developer Zone for more information on communicating with GPIB instruments.

Modular Instruments

NI modular instruments are high-speed hardware products that include the functionality of many traditional benchtop instruments for test and measurement. NI modular instruments include high-speed digitizers, signal generators, high-speed digital I/O devices, high-frequency (RF) devices, digital multimeters (DMMs), programmable power supplies and source measure units (SMUs), and switches. NI modular instrument platforms include PCI, PCI Express, PXI, PXI Express, SCXI, and USB.

You can use the instrument drivers that ship with NI modular instruments products to create custom measurement applications and user interfaces. Each instrument driver includes a palette of LabVIEW VIs that you can use to control a modular instrument from LabVIEW.

Because NI modular instruments combine flexible, user-defined software and scalable, general-purpose hardware components, you can modify measurements as your needs change. For example, you can make modifications to your program or substitute one instrument for another. You also can create custom measurements and define the algorithms used to make those measurements.

Refer to Modular Instruments at ni.com for more information about NI modular instruments.

Serial Port Communication

Serial communication transmits data between a computer and a peripheral device, such as a programmable instrument or another computer. Serial communication uses a transmitter to send data one bit at a time over a single communication line to a receiver. Use this method when data transfer rates are low or you must transfer data over long distances. Most computers have one or more serial ports, so you do not need any extra hardware other than a cable to connect the instrument to the computer or to connect two computers to each other.

You must specify four parameters for serial communication: the baud rate of the transmission, the number of data bits that encode a character, the sense of the optional parity bit, and the number of stop bits. A character frame packages each transmitted character as a single start bit followed by the data bits.

Baud rate is a measure of how fast data moves between instruments that use serial communication.

Data bits are transmitted upside down and backwards, which means that inverted logic is used and the order of transmission is from least significant bit (LSB) to most significant bit (MSB). To interpret the data bits in a character frame, you must read from right to left and read 1 for negative voltage and 0 for positive voltage.

An optional parity bit follows the data bits in the character frame. The parity bit, if present, also follows inverted logic. This bit is included as a means of error checking. You specify ahead of time for the parity of the transmission to be even or odd. If you choose for the parity to be odd, the parity bit is set in such a way so the number of 1s add up to make an odd number among the data bits and the parity bit.

The last part of a character frame consists of 1, 1.5, or 2 stop bits that are always represented by a negative voltage. If no further characters are transmitted, the line stays in the negative (MARK) condition. The transmission of the next character frame, if any, begins with a start bit of positive (SPACE) voltage.

Rate of Data Transfer

You can calculate the maximum transmission rate in characters per second for a given communication setting by dividing the baud rate by the bits per character frame.

Serial Hardware Overview

The following examples are the most common recommended standards of serial port communication:

- RS232 (ANSI/EIA-232 Standard) is used for many purposes, such as connecting a mouse, printer, or modem. It also is used with industrial instrumentation. Because of improvements in line drivers and cables, applications often increase the performance of RS232 beyond the distance and speed in the standards list. RS232 is limited to point-to-point connections between PC serial ports and devices.
- RS422 (AIA RS422A Standard) uses a differential electrical signal as opposed to the unbalanced (single-ended) signals referenced to ground with RS232. Differential transmission, which uses two lines each to transmit and receive signals, results in greater noise immunity and longer transmission distances as compared to RS232.
- RS485 (EIA-485 Standard) is a variation of RS422 that allows you to connect up to 32 devices to a single port and define the necessary electrical characteristics to ensure adequate signal voltages under maximum load. With this enhanced multidrop capability, you can create networks of devices connected to a single RS485 serial port. The noise immunity and multidrop capability make RS485 an attractive choice in industrial applications that require many distributed devices networked to a PC or other controller for data collection and other operations.

Connecting a Serial Device

If you have a serial device in a system, you first must obtain the pinout for that device and make sure you have the correct cable to connect the serial device to the computer. Determine if the device is data communications equipment (DCE) or data terminal equipment (DTE) and what settings it uses to communicate—baud rate, data bits, stop bits, parity, or handshaking (flow control).

More Information About Serial Communication

Refer to the NI Developer Zone for more information on communicating with Serial instruments.

Ethernet Communication

You can use [NI-VISA](#) to communicate with Ethernet devices, such as NI LXI-based instruments. Ethernet devices are used within measurement systems for tasks such as general networking and remote data storage.

Refer to the NI Developer Zone for information on communicating with Ethernet instruments using NI-VISA.

USB Communication

You can use NI-VISA to communicate with a USB device. Universal Serial Bus (USB) is a message-based communication bus. Each USB device has its own command set. You can use VISA Read and VISA Write functions to send these commands to an instrument and read the response from an instrument.

Refer to the NI Developer Zone for information on communicating with USB instruments using NI-VISA.



RELATED INFORMATION

[Using VISA in LabVIEW](#) on page 1616

Other Types of Communication

In addition to instrument drivers, you can use the Instrument I/O Assistant to communicate with message-based instruments and graphically parse the response. For example, you can use the Instrument I/O Assistant to send a query to an instrument to verify communication with that instrument.

If you do not create an instrument driver, use VISA to control GPIB, serial, USB, Ethernet, LXI, PXI, or VXI instruments. VISA is a standard API that you can use to control a wide range of instruments. VISA makes the appropriate driver calls depending on the type of

instrument you use so you do not have to learn instrument-specific communication protocols.

Use NI-488.2 to develop and debug an application program. The NI-488.2 driver has high-level commands that automatically handle all bus management, so you do not have to learn the programming details of the GPIB hardware board or the IEEE 488.2 protocol. Low-level commands are also available for maximum flexibility and performance.

Use the instrument and device drivers provided on the National Instruments Device Drivers DVD to control NI modular instruments and devices for industrial automation, such as NI-CAN.



RELATED INFORMATION

[Using VISA in LabVIEW](#) on page 1616

[GPIB Communication](#) on page 1603

[Serial Port Communication](#) on page 1604

[USB Communication](#) on page 1606

[Ethernet Communication](#) on page 1606

[Modular Instruments](#) on page 1604

Instrument Drivers

Using Instrument Drivers

An instrument driver is a set of high-level functions that control and communicate with instrument hardware in a system. In LabVIEW, an instrument driver is a set of VIs that communicate with an instrument using LabVIEW built-in VISA I/O functions. Each VI corresponds to a programmatic operation, such as configuring, reading from, writing to, and triggering an instrument. LabVIEW instrument drivers simplify instrument control and reduce test program development time by eliminating the need for you to learn the complex, low-level programming commands for each instrument. You also can [control instruments directly](#).

The Instrument Driver Network contains instrument drivers for a variety of programmable instruments that use the GPIB, Ethernet, serial, and other interfaces. You also can search for National Instruments drivers at NI Drivers and Updates on the National Instruments Web site. You can use an instrument driver for a particular instrument as is. However, LabVIEW Plug and Play instrument drivers are distributed with their block diagram source code, so you can customize them for a specific application. You can create instrument control applications and systems by programmatically linking instrument driver VIs on the block diagram. Use the NI Instrument Driver Finder to search for and install LabVIEW Plug and Play instrument drivers without leaving the LabVIEW development environment. Select **Tools»Instrumentation»Find Instrument Drivers** or **Help»Find Instrument Drivers** to launch the Instrument Driver Finder.

LabVIEW instrument drivers usually use VISA functions to communicate with instruments. You can use VISA for many different instrument types, such as GPIB, serial, USB, LXI, PXI,

and VXI. Once you learn how to use VISA for one type of instrument, you do not have to learn a different way to communicate when you use another type of instrument. You do have to learn about the specific command set for the two instruments, but the method by which you send and receive the commands does not change.



NOTE

You can use the Instrument I/O Assistant to communicate with message-based instruments and graphically parse the response.

NI provides more than 8,000 instrument drivers from more than 275 vendors. Refer to the Instrument Driver Network on the National Instruments Web site for a list of available instrument drivers. You also can search for National Instruments drivers at NI Drivers and Updates on the National Instruments Web site.

You can [search for and install](#) existing LabVIEW instrument drivers.



(Windows) To view related topics, click the **Locate** button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the **Contents** tab so you can navigate the related topics.

Types of Instrument Drivers

You can use Plug and Play and Interchangeable Virtual Instrument (IVI) driver types to control instruments in LabVIEW. National Instruments considers drivers Certified or Noncertified depending on if they meet specific requirements.

Plug and Play Instrument Drivers

A LabVIEW Plug and Play instrument driver is a set of VIs that controls and communicates with a programmable instrument. Each VI corresponds to a programmatic operation, such as configuring, reading from, writing to, or triggering an instrument. LabVIEW Plug and Play instrument drivers include error handling, front panels, block diagrams, icons, and help. Because LabVIEW Plug and Play drivers maintain a common architecture and interface, you can quickly connect to and communicate with an instrument with very little or no code development. Project-style drivers take advantage of the LabVIEW project and library. You can use project-style drivers in the same way as previous LabVIEW Plug and Play drivers.

Use the NI Instrument Driver Finder to search for and install LabVIEW Plug and Play instrument drivers without leaving the LabVIEW development environment. Select **Tools»Instrumentation»Find Instrument Drivers** or **Help»Find Instrument Drivers** to launch the Instrument Driver Finder. If you want to download IVI and contributed drivers, use the Instrument Driver Network. You also can search for National Instruments drivers at NI Drivers and Updates on the National Instruments Web site.

IVI Instrument Drivers

National Instruments IVI drivers are sophisticated, DLL-based drivers developed in LabWindows™/CVI™ that allow for simulation and instrument interchangeability. You do

not have to rewrite an application to use it with a similar instrument type. For example, you can write a VI that works with several different brands of oscilloscopes, even if those oscilloscopes use different bus connections. To achieve interchangeability, the IVI Foundation, which develops IVI standards, defines specifications for the following instrument classes: DMM, oscilloscope, arbitrary waveform/function generator, DC power supply, switch, power meter, spectrum analyzer, and RF signal generator. Use IVI drivers for the following additional benefits: instrument state caching for improved performance, multithread safety, and instrument attribute access. Refer to IVI on the National Instruments Web site for more information about IVI instrument drivers.

To convert an instrument driver written in LabWindows/CVI, download the LabVIEW Instrument Driver Import Wizard from the National Instruments Web site.

Finding and Installing Instrument Drivers

Use the NI Instrument Driver Finder to search for and install LabVIEW Plug and Play instrument drivers without leaving the LabVIEW development environment. Select **Tools»Instrumentation»Find Instrument Drivers** or **Help»Find Instrument Drivers** to launch the Instrument Driver Finder.

If you want to download an IVI driver or browse for noncertified drivers, you can download instrument drivers from the Instrument Driver Network. You also can download National Instruments drivers from NI Drivers and Updates on the National Instruments Web site.

LabVIEW installs instrument driver files to the `labview\instr.lib` directory. Access the instrument driver VIs installed to this directory on the **Instrument Drivers** palette.



NOTE

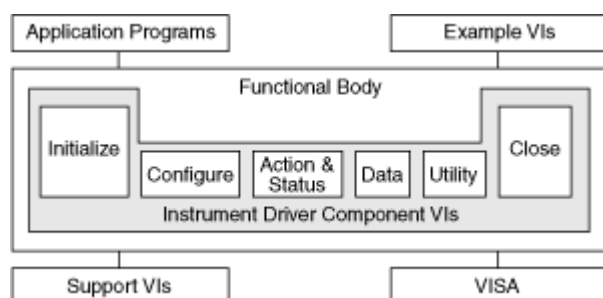
To install, create, or modify instrument drivers, the `labview\instr.lib` directory must have read and write access.

Accessing Instrument Driver VIs

The Instrument Driver VIs are located on the **Instrument Drivers** palette.

Organization of Instrument Driver

The following illustration shows the organization of a typical instrument driver.



Typical Instrument Driver Components

Most instrument drivers have menu palettes that include the following components.

- The Initialize VI, the first instrument driver VI you call, establishes communication with the instrument. Additionally, it can perform any necessary actions to place the instrument in its default power on state or in another specific state. Generally, you need to call the Initialize VI only once at the beginning of an application.
- The Configuration, Action, Data, and Utility VIs provide numerous functionalities to *control and communicate with* the instrument.
- The Close VI terminates the software connection to the instrument and frees system resources. Generally, you need to call the Close VI only once at the end of an application or when you finish communication with an instrument. Make sure that for each successful call to the Initialize VI, you use a matching Close VI to avoid maintaining unnecessary memory resources.

Use the NI Example Finder to search or browse for project-style *instrument driver examples*. You can find non-project style instrument driver examples in the **Instrument Drivers** palette and subpalettes.

Many instrument drivers have a VI Tree VI you can use to view the entire instrument driver hierarchy. The VI Tree VI is a non-executable VI that illustrates the functional structure of the instrument driver.

You can *test instrument driver VIs* to interactively determine the appropriate instrument configuration settings.

Getting Started with Instrument Drivers

To verify communication with an instrument and test a typical programmatic instrument operation, open an *instrument driver example VI*. Look over each of the controls and set them appropriately. Generally, with the exception of the resource name control, the defaults for most controls will be sufficient for the first run. You will need to set the resource name appropriately.

Use the NI Example Finder to search or browse for project-style instrument driver examples. You can find non-project style instrument driver examples in the **Instrument Drivers** palette and subpalettes.

After running the VI, check to see that the VI returned reasonable data and did not report an error in the error cluster. Instrument driver examples most commonly fail for one of the following reasons:

- NI-VISA is not installed. You install NI-VISA from the National Instruments Device Drivers DVD.
- The resource name is incorrect. The instrument driver example VI requires you to *specify the correct resource* for the instrument. Instruments that are powered on at LabVIEW launch time appear in the **VISA resource name** pull-down menu. If the

instrument does not appear in this list, launch [Measurement & Automation Explorer](#) to configure the device.

- The instrument driver does not support the exact model you are using. You might need to double-check that the instrument driver supports the instrument model you are using.

After you have verified basic communication with the instrument using the example VI, you might want to customize instrument control for your application. If the application requirements are similar to the example VI, the simplest means of creating a customized VI is to save a copy of the example VI by selecting **Save As** from the **File** menu. You can change the default values in the front panel window by selecting **Edit»Make Current Values Default**. Block diagram changes might include changing the constants wired to the instrument driver subVIs.

Interactively Testing Instrument Driver VIs

You might want to test [instrument driver VIs](#) interactively before you include them in an application. Testing instrument driver VIs helps to select appropriate instrument configuration settings. To run the driver VIs from their front panels, you must first run the Initialize VI. For subsequent VIs, you must first select the resource name from the **VISA resource name** pull-down menu.

After you have selected a resource, you can interactively run the driver VI from the front panel multiple times without resetting the resource selection.

In general, you should run the instrument driver VIs in the order you want to call them in the application. First you run the Initialize VI, followed by one or more configuration VIs. If you are using a trigger for the measurement, you might need to call an action VI to arm the trigger. Calls to data VIs then collect the measured value(s). When you are finished testing the instrument driver component VIs, you should run the Close VI to deallocate resources.

Creating Instrument Control Applications

Instrument drivers include many component VIs for use in building an instrument control application. Wire the VIs together similar to a typical LabVIEW application.

Establishing Communication

The Initialize VI, the first instrument driver VI you call, establishes communication with the instrument. Additionally, it can perform any necessary actions to place the instrument in its default power on state or in another specific state. Generally, you need to call the Initialize VI only once at the beginning of an application.



NOTE

Use the resource name and resource name out terminals to connect the Initialize VI to subsequent VIs that interact with or close the instrument.

Interacting with the Instrument

The Configuration VIs are a collection of software routines that configure the instrument to perform the operation you want. Numerous Configuration VIs can exist, depending on the particular instrument. After you call these VIs, the instrument is ready to take measurements or to simulate a system.

The Action VIs initiate or terminate test and measurement operations, such as arming the trigger system or generating a stimulus. Action VIs are different from Configuration VIs because they do not change the instrument settings but order the instrument to carry out an action based on its current configuration. The Status VIs obtain the current status of the instrument or the status of pending operations.

The Data VIs transfer data to or from the instrument. Examples include VIs for reading a measured value or waveform and VIs for downloading waveforms or digital patterns to a source instrument.

The Utility VIs perform a variety of operations that are auxiliary to the most often used instrument driver VIs. These VIs include the majority of the instrument driver template VIs, such as reset, self-test, revision, error query, and error message. The Utility VIs might also include other custom instrument driver VIs that perform operations such as calibration or storage and recall of setups.

Closing the connection

The Close VI terminates the software connection to the instrument and frees system resources. Generally, you need to call the Close VI only once at the end of an application or when you finish communication with an instrument. Make sure that for each successful call to the Initialize VI, you use a matching Close VI to avoid maintaining unnecessary memory resources.

Error Handling

It is important to perform [error handling](#) in instrument control applications because there are several potential sources for errors.

Modifying an Existing Instrument Driver

You might want to modify an existing driver to optimize the code. In general, you should optimize only those VIs that are called repeatedly in a loop. Configuration VIs generally are called only once and have little effect on application speed.

Follow the [Instrument Driver Modification Instructions](#) to modify an existing instrument driver.

If you need to modify only one or two VIs, consider renaming each VI by selecting **File»Save As** and selecting the **Rename** option in the Save As dialog box. After you modify

the name, you can modify the block diagram and front panel controls. Most changes to the block diagram should affect the String functions.

If you need to modify several VIs and the existing driver is a project-style driver, consider creating a new driver using the existing driver as the starting point. Select **Tools»Instrumentation»Create Instrument Driver Project** to launch the Create New Instrument Driver Project wizard and create a new driver using the existing driver as the source.

If you intend to submit the modified driver to National Instruments for certification, refer to the NI Developer Zone for more information on Instrument Driver Certification.

Creating Instrument Drivers

Before you build an instrument driver, you can [check for existing drivers](#) for the instrument you are using. You can search for existing drivers using the NI Instrument Driver Finder, the manufacturer's Web site, and the Instrument Driver Network on the National Instruments Web site. You also can search for National Instruments drivers at NI Drivers and Updates on the National Instruments Web site.

Use the Create New Instrument Driver Project wizard to create a new instrument driver project. Select **Tools»Instrumentation»Create Instrument Driver Project** to launch the Create New Instrument Driver Project wizard. After creating the new instrument driver, follow the [Instrument Driver Modification Instructions](#) to complete the driver.

Use the Instrument Driver VI Wizard to create an instrument driver VI for an instrument driver project. Right-click a LabVIEW instrument driver project library `.lvlib` in the [Project Explorer](#) window and select **New»Instrument Driver VI** from the shortcut menu to launch the Instrument Driver VI Wizard.



NOTE

To install, create, or modify instrument drivers, the `labview\instr.lib` directory must have read and write access.

Instrument Control Error Handling

It is important to perform error handling in instrument control applications because there are several potential sources for errors. Error handling with instrument driver VIs is similar to error handling with other I/O VIs in LabVIEW. Each instrument driver VI contains an **error in** input and an **error out** output for passing error clusters from one VI to another. The error cluster contains a Boolean flag that indicates if an error occurred, an error code number, and a string that contains the location of the VI where the error occurred.

- VISA functions can return errors because VISA or the underlying software or hardware is not properly installed. For example, when communicating with GPIB instruments, NI-488.2 must be installed correctly to use a National Instruments GPIB controller card. Similarly, if the board is not installed or is not correctly configured, the instrument driver VIs return an error.

- VISA functions can return errors if the device you are accessing is not responding to the commands you have sent. The instrument could be incorrectly addressed, malfunctioning, or unable to understand the commands that are being sent.

Use the Simple Error Handler or General Error Handler VIs to detect and report these errors in an application.

Measurement Applications

Creating a Typical DAQ Application

Use the *I/O controls* on the **I/O** and **Classic I/O** palettes to specify the instrument or device resource you want to communicate with. Use the VIs on the **Measurement I/O** palette to develop DAQ applications.

Complete the following basic programmatic steps to create a DAQ application.

1. *Configure the measurement hardware.*
2. Create tasks and channels.
3. (Optional) Set timing.
4. (Optional) Set triggering.
5. Read or write data.
6. Clear.

Notice that timing and triggering are optional. Include the timing step if you want to specify hardware timing instead of software timing. If you are using NI-DAQmx, you can use the DAQ Assistant to set timing parameters for a task.



NOTE

(Windows) LabVIEW supports NI-DAQmx and the DAQ Assistant. (Mac OS X) LabVIEW supports NI-DAQmx Base but not the DAQ Assistant. (Linux) LabVIEW supports NI-DAQmx but not the DAQ Assistant.

Use triggering if you want the device to acquire samples only when certain conditions are met. For example, you might want to acquire samples if the input signal goes higher than 4 V. If you are using NI-DAQmx, you can use the DAQ Assistant to configure triggering for a task.

Many NI-DAQmx applications also can include steps to start, stop, and clear the task. For example, for applications that use a counter/timer to count edges or to measure period, use the Start VI to arm the counter.

In NI-DAQmx, LabVIEW clears the task automatically when the VI hierarchy that created the task finishes executing.

Traditional NI-DAQ (Legacy) and NI-DAQmx include VIs for timing, triggering, reading, and writing samples. Use the Data Acquisition VIs and Functions to acquire data from DAQ devices. To use these VIs, you must have the [NI-DAQ driver software and DAQ hardware](#) installed. After you acquire data, you can use the built-in Signal Processing, Report Generation, and Mathematics VIs and functions to analyze, generate reports, and perform mathematical operations on that data.

You can [use the NI-DAQmx properties](#) to extend the functionality of the NI-DAQmx VIs.

Many of the Traditional NI-DAQ (Legacy) and NI-DAQmx VIs are polymorphic and can accept or return data of various types, such as scalar values, arrays, or waveforms. You use other polymorphic NI-DAQmx VIs to configure various triggers and methods of sample timing, and to create virtual channels. By default, NI-DAQmx VIs appear with the [polymorphic VI](#) selector.

Physical and Virtual Channels

A physical channel is a terminal or pin at which you can measure or generate an analog or digital signal. Every physical channel on a device that supports NI-DAQmx has a unique name.

A virtual channel is a collection of property settings that can include a name, a physical channel, input terminal connections, the type of measurement or generation, and scaling information. In Traditional NI-DAQ (Legacy) and earlier versions, configuring virtual channels is an optional way to record which channels are being used for different measurements, but virtual channels are integral to every NI-DAQmx measurement.

Tasks

A task in NI-DAQmx is a collection of one or more virtual channels with timing, triggering, and other properties. A task represents a measurement or a generation you want to perform. You can set up and save all of the configuration information in a task and use the task in an application.

In NI-DAQmx, you can configure virtual channels as part of a task or separate from a task.

Complete the following steps to perform a measurement or a generation with a task.

1. Create a task and channels.
2. (Optional) Configure the channel, timing, and triggering properties.
3. Read or write samples.
4. Clear the task.

Repeat steps 2 and 3, if it is appropriate for the application. For example, after reading or writing samples, you can reconfigure the channel, timing, or triggering properties and then read or write additional samples based on this new configuration.

Creating a Typical FieldPoint Application

Use the *I/O controls* on the **I/O** and **Classic I/O** palettes to specify the instrument or device resource with which you want to communicate. Use the VIs on the **FieldPoint** palette to develop FieldPoint distributed I/O applications. Most FieldPoint applications require only the FP Read VI and the FP Write VI. The FP Read VI returns data from the FieldPoint I/O channel or group of channels the FieldPoint IO Point function represents. The FP Write VI sends data to the FieldPoint I/O channel or group of channels the FieldPoint IO Point function represents. You also can *use properties* to extend the functionality of FieldPoint VIs programmatically.

Channels versus Items

FieldPoint modules consist of physical I/O points called channels. Items represent the channels or groups of channels. You can create items in Measurement & Automation Explorer (MAX) and use the FieldPoint I/O Point control to access the items in LabVIEW.

Using the FieldPoint I/O Point Control

Use the FieldPoint I/O Point control to communicate with the FieldPoint items you create in MAX. Place the FieldPoint I/O Point control in the front panel window. Right-click the control and select the items you want to read from or write to from the shortcut menu. If you do not see the FieldPoint items you want, configure the items in MAX.

Using Properties with Measurement Applications

You can write most measurement applications using only the VIs of the NI-DAQmx, NI-VISA, and IVI Instrument Driver APIs. You also can use properties with these APIs to extend the functionality to include less commonly used features. For example, you can use the VISA Configure Serial Port VI to set several commonly used serial port settings in a VISA Session, including the baud rate. However, if you want to change only the baud rate, you can use a Property Node.

Use the Property Node on the **DAQmx** palette to configure various low-level settings for NI-DAQmx. Use the Property Node on the **VISA Advanced** palette for any VISA property. Use the Property Nodes on the **Modular Instrument** palette and the **IVI Instrument Drivers** palette for these APIs, respectively.

VISA in LabVIEW

Using VISA in LabVIEW

VISA is a standard I/O API for instrumentation programming.

Versatility of VISA

VISA can control GPIB, serial, USB, Ethernet, PXI, or VXI instruments, making the appropriate driver calls depending on the type of instrument you use so you do not have to learn instrument-specific communication protocol. It is platform independent, bus independent, and environment independent. In other words, the same API is used regardless of device type, platform, or programming language.

Before you begin *using VISA*, make sure you *choose the appropriate method of instrument control*.

Message-Based Communication

GPIB, serial, USB, Ethernet, and some VXI instruments use message-based communication. You program message-based instruments with high-level ASCII character strings. The instrument has a local processor that parses the command strings and sets the appropriate register bits to perform the operations you want. The Standard Commands for Programmable Instruments (SCPI) standardizes the ASCII command strings used to program compliant instruments. Similar instruments often use similar commands. Instead of learning different command messages for each type of instrument from each manufacturer, you need to learn only one command set. The most common message-based functions are VISA Read, VISA Write, VISA Assert Trigger, VISA Clear, and VISA Read STB.

Register-Based Communication

PXI and many VXI instruments use register-based communication. You program register-based instruments at a low level using binary information that you write directly to the instrument control registers. Speed is the advantage of this type of communication because the instrument no longer needs to parse the command strings and convert the information to register-level programming. Register-based instruments literally communicate at the level of direct hardware manipulation. The most common register-based functions are VISA In, VISA Out, VISA Move In, and VISA Move Out.



(Windows) To view related topics, click the **Locate** button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the **Contents** tab so you can navigate the related topics.

Selecting a Resource Name

The front panel of most instrument driver VIs includes a VISA resource name control and a **VISA resource name out** indicator. These *controls and indicators* pass session information between instrument driver subVIs. The VISA resource name identifies the resource on which the VI operates. It also differentiates between different sessions of the instrument driver.

VISA resource name is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and passes all necessary configuration information required to perform the I/O.

VISA resource name out contains the same identifier information as **VISA resource name**. It passes the reference out of the VI and into other subsequent VIs that access the same instrument. Wiring VISA resource names together establishes data dependencies.

You select the VISA resource name in the Initialize VI from the **VISA resource name** pull-down menu. Instruments that are powered on and connected to your computer are visible in the **VISA resource name** pull-down menu. If the instrument does not appear in this list, enter the instrument descriptor using the following format:

Interface Type[board index]::Address::INSTR. Examples resource names: GPIB: GPIB::22::INSTR, GPIB board 0, primary address 22
 Serial: COM1 TCP/IP Instr: TCPIP::10.0.32.48::INSTR TCP/IP
 Socket: TCPIP::10.0.32.45::3800::SOCKET

(Windows) Use [MAX](#) to determine the available resources and instrument addresses and set aliases.

Verifying VISA Communication

To verify that VISA and the device driver are installed correctly and VISA detects the instrument, first view the resource names that appear in the **VISA resource name** pull-down menu. If no resource names appear in this list, VISA or the device driver might not be installed correctly. (Windows) Use [Measurement & Automation Explorer](#) to configure and test instrument communication. If resource names appear in the **VISA resource name** pull-down menu, but a particular instrument does not appear in the list, you might have a cable connection problem.

If VISA is installed correctly and VISA detects the instrument, but you receive errors when you run the instrument control application, you must identify what sequence of VIs produces the error in LabVIEW. If you are using an instrument driver, try to [run each VI interactively](#) and verify that each control is set appropriately.

NI I/O Trace is a tool for monitoring instrument I/O communications while your application runs. You can use NI I/O Trace to capture instrument I/O calls and their results while LabVIEW VIs run. Capturing calls and their results can help you debug problems with instrument communications. Select **Tools»Instrumentation»NI I/O Trace** to launch NI I/O Trace. (Windows) You also can select **Start»All Programs»National Instruments»NI I/O Trace** to launch NI I/O Trace. (Mac OS X) You also can select **Applications»National Instruments»NI I/O Trace** and double-click the NI I/O Trace icon to launch NI I/O Trace. (Linux) You also can type `niotrace` in the command line to launch NI I/O Trace.

Creating a Typical VISA Application

Use the [I/O controls](#) on the **I/O** and **Classic I/O** palettes to specify the instrument or device resource with which you want to communicate.

VISA Functions and Properties

Use the VIs and functions on the VISA palette to build VIs that control instruments. Use VISA properties to extend the functionality of VISA VIs programmatically. Before you begin using VISA, make sure you *choose the appropriate method of instrument control*.

For most simple instrument applications, you need only two VISA functions, VISA Write and VISA Read. Refer to the Simple Serial VI in the `labview\examples\Instrument IO\Serial` directory for an example of how to use VISA functions.



Open example



Find related examples

Use the VIs in the VISA Advanced VIs and functions palette to build advanced VISA VIs. Refer to the folders in the `labview\examples\Instrument IO\VISA` directory for examples of using advanced VISA VIs.

Refer to the `labview\examples\Instrument IO\GPIB\GPIB.lvproj` for examples of using VISA VIs with GPIB devices.



Open example



Find related examples

LabVIEW Shared Libraries

This section provides information on using shared libraries in LabVIEW.

Calling Shared Libraries

A shared library is a file containing executable program modules that any number of different programs can use to perform some function. You can call these program modules, or shared library functions, from LabVIEW by using the Call Library Function Node. Call a shared library function when you want to accomplish any of the following tasks:

- Using functions written in another programming language within LabVIEW
- Performing tasks that LabVIEW cannot perform with its own functions and VIs, such as calling system routines
- Completing tasks that are better suited for text-based languages

Shared Libraries on Different Platforms

(Windows) A shared library is called a DLL.

(OS X) A shared library is called a Framework.

(Linux) A shared library is called a Shared Library function.

General Approach to Calling Shared Libraries

Calling a shared library function in LabVIEW includes the following actions:

1. You obtain or create a shared library.
2. You configure a Call Library Function Node to call a function within the shared library.
3. LabVIEW calls the function when the Call Library Function Node executes.
4. LabVIEW passes input data from the block diagram to the executable code.
5. LabVIEW returns data from the executable code to the block diagram.

Limitations on Calling Shared Libraries in LabVIEW

- **Shared libraries must use either the stdcall or C calling convention.** You can use any language to write shared libraries as long as the shared libraries can be called using one of these two calling conventions.
- **Shared libraries execute synchronously.** You cannot reset a VI that is running a shared library until execution completes. If you want to write a shared library that performs a long task, be aware that LabVIEW cannot perform other tasks in the same thread while the shared library executes.
- **Vis that call shared libraries are platform specific.** VIs that call functions within shared libraries execute on different platforms only if you have a corresponding library for each platform. If you move a VI that call a shared library with a Call Library Function Node to another platform, you must update the node to specify the version of the shared library that is compiled for the new platform.

Examples and troubleshooting information help you build and use shared libraries and successfully configure the Call Library Function Node in LabVIEW. The general methods described here for DLLs also apply to other types of shared libraries. Refer to the `labview\examples\dll` directory for examples of using shared libraries.

Configuring the Call Library Function Node

Use the Call Library Function Node to directly call a 32-bit Windows DLL, a OS X Framework, or a Linux Shared Library function. With this node, you can create an interface in LabVIEW to call existing libraries or new libraries specifically written for use with LabVIEW. National Instruments recommends using the Call Library Function Node to create an interface to external code.



NOTE

Be aware when using the Call Library Function Node or writing code that is called by the Call Library Function Node that LabVIEW reserves Windows messages WM_USER through WM_USER+99 for internal use only.

Right-click the Call Library Function Node and select **Configure** from the shortcut menu to display the Call Library Function dialog box. Use the **Call Library Function** dialog box to

specify the library, function, parameters, return value for the node, calling conventions, and function callbacks on Windows. When you click the **OK** button in the **Call Library Function** dialog box, LabVIEW updates the Call Library Function Node according to your settings, displaying the correct number of terminals and setting the terminals to the correct data types.



NOTE

If you want to run applications or shared libraries created with different versions of LabVIEW on the same computer, the computer *must have a version of the LabVIEW Run-Time Engine that is compatible* with each version of LabVIEW used to create the applications or shared libraries.

Configuring Library Name or Path

You must specify either the name of the shared library or the path to the shared library on disk. However, these two techniques produce different consequences when you distribute or relocate your VIs and applications.



CAUTION

Do not specify the location of a system shared library, such as `kernel32.dll`, by path. Because LabVIEW automatically includes by-path shared libraries in build specifications, you might distribute a by-path system shared library to another computer inadvertently. Redistributing some shared libraries, such as `kernel32.dll`, might cause the target computer to crash. Therefore, always specify system shared libraries by name.

If you want to run applications or shared libraries created on different platforms, use the * wildcard to make the reference to the shared library platform independent. Use * for the file extension and * or ** to the left of the file extension, depending on how you name 32-bit and 64-bit libraries.

The following example illustrates how to use the * wildcard.

Example	Translation
<code>myshared.*</code>	LabVIEW replaces the reference with the appropriate file extension to match the platform that is running the Call Library Node, for example, <code>myshared.dll</code> , <code>myshared.so</code> , and <code>myshared.framework</code> .
<code>myshared*.*</code>	LabVIEW replaces the reference with <code>myshared32.*</code> on 32-bit platforms and <code>myshared64.*</code> on 64-bit platforms. You can place the * anywhere to the left of the file extension. For example, <code>my*shared.*</code> can translate to <code>my32shared.*</code> . LabVIEW replaces <code>.*</code> with the appropriate file extension.
<code>myshared**.*</code>	LabVIEW replaces the reference with <code>myshared.*</code> on 32-bit platforms and <code>myshared_64.*</code> on 64-bit platforms. You can place the ** anywhere to the left of the file extension. For example, <code>my**shared.*</code> can translate to <code>my_64shared.*</code> . LabVIEW replaces <code>.*</code> with the appropriate file extension.

**NOTE**

Keep the C++ compiler from introducing platform dependence in exported function names through a process called name mangling by using the C++ compiler function export directive, `extern "C"{};` in your header file.

Related Information*Specifying the Location of Shared Libraries on Disk***Configuring for Multiple Thread Operation**

In a multithreaded operating system, you can make multiple calls to a DLL or shared library simultaneously. You can select the thread to execute the library call from the **Thread** section on the **Function** tab of the **Call Library Function** dialog box. The thread options are **Run in UI thread** and **Run in any thread**. If you select **Run in UI thread**, the Call Library Function Node switches from the thread the VI is currently executing in to the user interface thread. If you select **Run in any thread**, the Call Library Function Node continues in the currently executing thread. By default, all Call Library Function Nodes run in the user interface thread.

Before you configure a Call Library Function Node to run in any thread, make sure that multiple threads can call the function simultaneously. In a shared library, code can be considered thread-safe when:

- It does not store any global data, such as global variables, files on disk, and so on.
- It does not access any hardware. In other words, the code does not contain register-level programming.
- It does not make any calls to any functions, shared libraries, or drivers that are not thread safe.
- It uses semaphores or mutexes to restrict access to global resources.
- It is called by only one non-reentrant VI.

**NOTE**

All calls to LabVIEW-built shared libraries should specify **Run in any thread**. If you configure the Call Library Function Node using LabVIEW-built shared libraries and specify **Run in UI thread**, LabVIEW might hang and require you to restart.

Related Information*Multitasking, Multithreading, and Multiprocessing***Setting the Calling Convention**

Calling conventions define the way to pass information from a piece of code to a function. Use the **Calling convention** control on the **Function** tab of the **Call Library Function** dialog box to select the calling convention for the function. The default calling convention is C. The C calling convention allows variable-length parameter lists.

(Windows) You also can use the standard Windows calling convention, `stdcall`. The number of parameters passed to the function is fixed.

Refer to the documentation for the DLL you want to call for the appropriate calling conventions.



CAUTION

Using the incorrect calling convention can cause an irregular shutdown of LabVIEW.

Configuring Parameters

This section discusses the return value and how to add parameters to the Call Library Function Node.

To configure parameters for the Call Library Function Node, navigate to the **Parameters** tab of the **Call Library Function** dialog box. Initially, the Call Library Function Node has no parameters and has a return type of **Void**.

As you configure parameters, the **Function Prototype** text box displays the C prototype for the function you are building. This text box is a read-only display.



NOTE

If a type library is found, the parameters are updated to match the parameters found in the type library for the selected function. The order of the parameters must match the prototype of the function found in the library.

The return type for the Call Library Function Node returns to the right terminal of the top terminal. If the return type is **Void**, the top terminal is unused. Each additional pair of terminals corresponds to a parameter in the **Parameters** list of the Call Library Function Node. To pass a value to the Call Library Function Node, wire to the left terminal of a terminal pair. To read the value of a parameter after the Call Library Function Node call, wire from the right terminal of a terminal pair. The following illustration shows a Call Library Function Node that has a return type of **Void**, a string parameter, and a numeric parameter.



Configuring Return Type

For return type, you can set **Type** to **Void**, Numeric, or String. **Void** is only available for return type and is not available for other parameters. Use **Void** for the return type if your function does not return any values.

Even if the function you call returns a value, you can use **Void** for the return type. When the function returns a value and you select **Void** as the return type, the value returned by the function is ignored.

**NOTE**

The function you are calling can return a C string pointer. If you want to deallocate the pointer, you must do so explicitly as LabVIEW does not automatically deallocate the C string pointer for you.

**TIP**

If the function you are calling returns a data type not listed, choose a return data type the same data size as the one returned by the function. For example, if the function returns a char data type, use an 8-bit unsigned integer. A call to a function in a DLL cannot return a pointer because there are no pointer types in LabVIEW. However, you can specify the return type as an integer that is the same size as the pointer. LabVIEW then treats the address as a simple integer, and you can pass it to future DLL calls.

Related Information*String Return Type***Adding and Deleting Parameters**

To add parameters to the Call Library Function Node, navigate to the **Parameters** tab of the **Call Library Function** dialog box. Click the **Add a parameter** button. To remove a parameter, click the **Delete the selected parameter** button. To change the order of the parameters, use the **Move the selected parameter up one** and **Move the selected parameter down one** buttons to the right of the parameter list.

Editing Parameters

Select the parameter from the **Parameters** list to edit the data type or parameter name. You can edit the parameter name to something more descriptive, which makes it easier to distinguish between parameters. The parameter name does not affect the call, but it is propagated to output wires. Also, you can edit all fields in the **Current parameter** section for the selected parameter.

Selecting the Parameter Type

Use the **Type** pull-down menu to indicate the data type of each parameter. You can select from the following parameter types:

- **Numeric**
- **Array**
- **String**
- **Waveform**

- **Digital Waveform**
- **Digital Data**
- **ActiveX**
- **Adapt to Type**
- **Instance Data Pointer**



NOTE

If the library function expects a data type not included in the previous list, you must convert the LabVIEW data into the type the function expects. Common data types that require conversion include structures, arrays containing pointers to other data, and pointers to callback functions.

After you select an item from the **Type** pull-down menu, you see more items you can use to indicate details about the parameter and about how to pass the data to the library function. The Call Library Function Node has a number of different items for parameter types because of the variety of data types required by different libraries. Refer to the documentation for the library you call to determine which parameter types to use.

The following sections discuss the different parameter types available from the **Type** pull-down menu.

(Windows) Refer to the `labview\examples\Connectivity\Libraries and Executables\Libraries and Executables.lvproj` for an example of using data types in shared libraries.

Related Information

Calling Functions That Expect Other Data Types

Numeric

For numeric data types, you must indicate the exact numeric type by using the **Data Type** pull-down menu. You can choose from the following data types:

- 8-, 16-, 32-, 64-bit, and pointer-sized signed and unsigned integers
- 4-byte, single-precision numbers
- 8-byte, double-precision numbers

If you use pointer-sized integers, the Call Library Function Node adapts to the specific operating system it is being executed on and passes data of the appropriate size to and from the library function. LabVIEW represents the data in 64 bits and, on 32-bit platforms, translates the numeric data types to 32-bit integer types.



NOTE

You can pass extended-precision numbers and complex numbers by selecting **Adapt to Type** from the **Type** pull-down menu. However, standard libraries generally do not use extended-precision numbers and complex numbers.

Use the **Pass** pull-down menu to indicate whether you want to pass the value or a pointer to the value.

Array

Use the **Data Type** pull-down menu to indicate the data type of the array. You can choose from the same data types available for numeric parameters.

Specify the dimensions of the array in **Dimensions**.

Use the **Array Format** pull-down menu to make one of the following choices:

- **Array Data Pointer** —passes a pointer to the array data, allowing the called library to access the array data as the specified data type of the array data.
- **Array Handle** —passes a pointer to a pointer that points to a four-byte value for each dimension, followed by the data.
- **Array Handle Pointer** —passes a pointer to an array handle.

Use the **Minimum size** control to have LabVIEW check at run time that the memory LabVIEW allocated for an array data pointer is at least the **Minimum size**. To indicate the **Minimum size** of a 1D array, you can enter a numeric value, or, if you configure an integer parameter in the **Parameters** list, you can select the parameter from the pull-down menu. This option is available only for array data pointers.



NOTE

If you pass in an array that is smaller than the **Minimum size**, LabVIEW enlarges the size of the array to the minimum. However, if you pass in an array that is bigger than the minimum, the array retains the larger size.



CAUTION

Do not attempt to resize an array with system functions, such as `realloc`. Doing so might crash your system. Instead, use one of the memory manager functions, such as `NumericArrayResize`.

Related Information

Memory Manager Functions

NumericArrayResize Function

String

Use the **String Format** pull-down menu to indicate the string format. You can choose from the following string formats:

- **C String Pointer** —a string followed by a null character.

- **Pascal String Pointer** —a string preceded by a length byte.
- **String Handle** —a pointer to a pointer to four bytes for length information, followed by string data.
- **String Handle Pointer** —a pointer to an array of string handles.

Select a string format that the library function expects. Most standard libraries expect either a C string or a Pascal string. If the library function you are calling is written for LabVIEW, you might want to use the **String Handle** format. When configuring a Pascal string pointer, you must wire a value to the string input on the block diagram. That value must be initialized with enough characters to hold any new string that may be written to that Pascal string. When configuring a C string pointer, you have two options:

- Wire a value to the string input that is initialized with enough characters to hold any new string that may be written to that string.
- Specify the string size in the **Minimum size** pull-down menu on the **Parameters** tab of the **Call Library Function** dialog box.

Use the **Minimum size** control to have LabVIEW check at run time that the memory LabVIEW allocated for a C string pointer is at least the **Minimum size**. To indicate the **Minimum size** of a string, you can enter a numeric value, or, if you configure an integer parameter in the **Parameters** list, you can select the parameter from the pull-down menu. This option is available only for C string pointers.



NOTE

If you pass in a string that is smaller than the **Minimum size**, LabVIEW enlarges the size of the string to the minimum. However, if you pass in a string that is bigger than the minimum, the string retains the larger size.



CAUTION

Do not attempt to resize an array with system functions, such as `realloc`. Doing so might crash your system. Instead, use one of the memory manager functions, such as `NumericArrayResize`.

Waveform

When you call a shared library that includes a waveform data type, you do not have to specify a numeric value from the **Data Type** pull-down menu; the default is **8-byte Double**. However, you must specify **Dimensions**. If the parameter is a single waveform, specify **Dimensions** as 0. If the parameter is an array of waveforms, specify **Dimensions** as 1. LabVIEW does not support an array of waveforms greater than one-dimensional.



NOTE

Pointer-sized signed and unsigned integers are not available in the **Data Type** pull-down menu for waveforms.

Digital Waveform

Specify **Dimensions** as 0 if the parameter is a single digital waveform. Specify **Dimensions** as 1 if the parameter is an array of digital waveforms. LabVIEW does not support an array of digital waveforms greater than one-dimensional.

Digital Data

Specify **Dimensions** as 1 if the Parameter is an array of digital data. Otherwise, specify **Dimensions** as 0. LabVIEW does not support an array of digital data greater than one-dimensional.



NOTE

You can pass waveforms, digital waveforms, and digital data through shared libraries, but you cannot access the data inside the shared libraries.

ActiveX

Select one of the following items from the **Data Type** pull-down menu:

- **ActiveX Variant Pointer** —passes a pointer to ActiveX data.
- **IDispatch* Pointer** —passes a pointer to the IDispatch interface of an ActiveX Automation server.
- **IUnknown* Pointer** —passes a pointer to the IUnknown interface of an ActiveX Automation server.

Related Information

[Using ActiveX with LabVIEW](#)

Adapt to Type

Use **Adapt to Type** to pass arbitrary LabVIEW data types to DLLs in the following ways:

- Scalars are passed by reference. A pointer to the scalar is passed to the library.
- Arrays and strings are passed according to the **Data Format** setting. You can choose from the following **Data Format** settings:
 - **Handles by Value** passes the handle to the library. The handle is not NULL.
 - **Pointers to Handles** passes a pointer to the handle to the library. If the handle is NULL, treat the handle as an empty string or array. To set a value when the handle is NULL, you must allocate a new handle.
 - **Array Data Pointer** passes a pointer to the first element of the array, allowing the called library to access the array data as the data type of the array data.

- **Interface to Data** allows you to inspect LabVIEW data and data type information in C++ code. For more information about this setting, refer to the `ILVDataInterface.idl` and `ILVTypeInterface.idl` files in the `labview\cintools` directory.
- Clusters are passed by reference.
- Scalar elements in arrays or clusters are in line. For example, a cluster containing a numeric is passed as a pointer to a structure containing a numeric.
- Clusters within arrays are in line.
- Strings and arrays within clusters are referenced by a handle.



NOTE

When one or more of the parameters of the function you want to call in a DLL are of types that do not exist in LabVIEW, ensure that each parameter is passed to the function in a way that allows the DLL to correctly interpret the data. Create a skeleton `.c` file from the current configuration of the Call Library Function Node. By viewing the `.c` file, you can determine whether LabVIEW will pass the data in a manner compatible with the DLL function. You then can make any necessary adjustments.

Related Information

Completing the .c File

Instance Data Pointer

Use **Instance Data Pointer** to access data allocated for each instance of the Call Library Function Node. The **Instance Data Pointer** references a pointer sized allocation that you may use at your own discretion. This allocation is also passed to each of the callback functions on the **Callbacks** tab.

Configuring Callbacks

When you configure a Call Library Function Node to call a function, you can use the **Callback** tab to specify other functions within the same library to call at the following times:

- **Reserve time**—When the top-level VI that causes the Call Library Function Node to execute begins executing. Specify a Reserve callback when you need to perform initialization tasks before the primary function executes.
- **Unreserve time**—When the top-level VI that caused the Call Library Function Node to execute stops executing. Specify an Unreserve callback when you save or analyze information or carry out clean-up operations.
- **Abort**—The specified function executes if the VI that called the primary function aborts.

Instance Data Pointer Instance Data Pointer



NOTE

You cannot use the **Callback** tab to pass callback functions as parameters to library functions. The Call Library Function Node cannot directly call library functions that require a callback function as a parameter, but you can [create a wrapper library](#) to implement such calls.

Calling Functions That Expect Other Data Types

You might encounter a function that expects data in a form that the Call Library Function Node cannot pass. Specifically, the Call Library Function Node does not support structures or arrays containing a pointer to other data or structures containing flat arrays that can be variably sized. You can call a function that expects an unsupported data type in the following ways:

- If the data contains no pointers, you might be able to use the Flatten To String function to create a string containing the binary image of the data required and pass this string as a C string pointer. You will probably want to use the **byte order** input to Flatten To String to specify that the data be flattened in native byte order.
- Write a library function that accepts the data in the form used by LabVIEW and builds the data structure expected by the other library. This function then can call the other library and retrieve any returned values before returning. Your function will probably accept the data from the block diagram as Adapt to Type, so that any block diagram data type can be passed.
- If the function expects a callback function as a parameter, complete the following steps to call the function from LabVIEW.
 1. Write a wrapper library that defines at least two new functions:
 - A wrapper function that accepts from LabVIEW any data required by the original function. This wrapper function must call the original function, passing the data from LabVIEW as well as a callback function that you define in the same wrapper library.
 - The callback function that you want to pass to the original function.
 2. Call the wrapper function with the Call Library Function Node.

Related Information

Flatten To String Function

Configuring Error Checking

Use error checking to ensure no errors occur if you call a DLL or shared library with the Call Library Function Node.

The **Maximum** and **Default** controls on the **Error Checking** tab of the Call Library Function dialog box allow LabVIEW to recover from unhandled exceptions that occur in the configuration of the Call Library Function Node or during a call to a shared library or DLL.

The **Disabled** control on the **Error Checking** tab disables error checking but improves the execution speed of the Call Library Function Node.

If you use the Call Library Function Node to call a shared library generated by LabVIEW, the node automatically calls the LVDLLStatus function embedded in the shared library. For these LabVIEW-generated shared libraries, this function detects certain errors, including the following:

- A VI inside the shared library uses licensed features that are not installed on the target computer.
- A VI inside the shared library uses a Call Library Function Node whose associated shared library is not installed on the target computer.
- The VIs inside the shared library were compiled using the SSE2 optimizations but the target computer does not support SSE2 instructions.

In these cases, the Call Library Function Node returns error 1003.

Specifying the Location of Shared Libraries on Disk

To call a shared library with the Call Library Function node, you must specify either the name of the shared library or the path to the shared library on disk. However, these two techniques produce different consequences when you distribute or relocate your VIs and applications. Use the following table to determine which technique addresses your particular use case.

Technique for Specifying Location	Relevant use cases	Basis of Search Algorithm	Effects on Distribution
By name	The shared library is in the system search path. Common examples include system shared libraries, such as <code>kernel32.dll</code> , and third party shared libraries designed for componentized reuse.	System search algorithm	LabVIEW does not include the shared library automatically in build specifications.
By path	The shared library is private to your VI. Common examples include shared libraries that you wrote specifically for a particular VI.	The path you specify	LabVIEW automatically includes the shared library in build specifications.

To specify the location of a shared library, you can enter the name or path in the **Library name or path** text box of the **Call Library Function** dialog box. You can also wire the name or path to the **path in** input of the Call Library Function Node.

**CAUTION**

Do not specify the location of a system shared library, such as `kernel32.dll`, by path. Because LabVIEW automatically includes by-path shared libraries in build specifications, you might distribute a system shared library to another computer inadvertently. Redistributing some shared libraries, such as `kernel32.dll`, might cause the target computer to crash. Therefore, always specify system shared libraries by name.

Common Problems Loading a Shared Library by Name

If LabVIEW cannot locate a shared library by name, consider the following possible causes:

- To locate shared libraries that you specify by name, LabVIEW uses a modified version of the system search algorithm defined by your operating system. Refer to the *Related information* for more about how LabVIEW shared library directories are used by the `Call Library Function` node.
- Specifying a shared library by name is not the same as specifying a relative path to a shared library within the same directory as the calling VI. To specify the latter, enter `./ shared library name` as the path to the shared library.

Common Problems Loading a Shared Library by Path

If LabVIEW cannot locate a shared library by path, consider the following possible causes:

- If you use a relative path to specify the location of the shared library, LabVIEW interprets the path relative to the location of the VI on disk.
- If you use an absolute path to specify the location of the shared library, LabVIEW saves the relative location of the shared library rather than the absolute path. Therefore, if you change the location of the shared library relative to the VI, LabVIEW cannot locate the shared library and must search for it.

How LabVIEW Locates Missing Shared Libraries

If LabVIEW cannot locate a shared library that you specified either by name or by path, LabVIEW searches the VI search path for the missing shared library. If LabVIEW finds the shared library, it updates the path to the shared library within the `Call Library Function` dialog box.

Searching for missing shared libraries requires extra time, displays an unavoidable search dialog box, and causes unsaved changes in the calling VI that you must save yourself.

Therefore, configure VIs with the correct name or path information for shared libraries whenever possible.



NOTE

You can configure stand-alone applications to search for shared libraries in specific locations on a target computer by using a `.ini` file. Refer to the *Related information* for more about how to change the VI search path for LabVIEW executables.



RELATED INFORMATION

[Call Library Function Node](#)

[Call Library Function Dialog Box](#)

[Paths Page \(Options Dialog Box\)](#)

[LabVIEW Shared Library Directories Used by Call Library Function Node](#)

[How Can I Change or Set the VI Search Path for LabVIEW Executables?](#)

Calling External APIs

You might need to access external APIs from within LabVIEW code. Most often, you access external APIs to obtain functionality that the operating system provides. Normally, you can use the LabVIEW [Call Library Function Node](#) to accomplish this goal. You must provide the following information to the Call Library Function Node to access external APIs from within LabVIEW code:

- Function name as it appears in the library
- [Function prototype](#)
- Library or module in which the function resides
- Calling conventions of the function
- Thread-safe status of the function



CAUTION

Do not [specify the location](#) of a system shared library, such as `kernel32.dll`, by path. Because LabVIEW automatically includes by-path shared libraries in build specifications, you might distribute a system shared library to another computer inadvertently. Redistributing some shared libraries, such as `kernel32.dll`, might cause the target computer to crash. Therefore, always specify system shared libraries by name.

Common Pitfalls with the Call Library Function Node

The function reference documentation for any API should provide most of the information that the Call Library Function Node requires. However, you should keep in mind the common errors listed in this section.



NOTE

Configure the Call Library Function Node to return an error when some of the pitfalls described below occur by selecting the **Maximum** or **Default** control on the **Error Checking** tab of the **Call Library Function** dialog box.

Incorrect Function Name

Your library call can fail when the name of the function as it appears in the library is different than is expected. Usually this error occurs due to function name redefinition, or to function name decoration, as in the following examples:

- **Redefinition** —This pitfall appears when an API manufacturer uses a define mechanism, such as `#define` directive in ANSI C, to define an abstracted function name to one of many functions present in the library, based on some external condition such as language or debug mode. In such cases, you can look in the header .h file for the API to determine whether a `#define` directive redefined the name of a function you want to use.
- **Function Name Decoration** —This pitfall appears when certain functions have their names decorated when they are linked. A typical C compiler tracks name decoration, and when it looks for a function in a shared library, it recognizes the decorated name. However, because LabVIEW is not a C compiler, it does not recognize decorated names. If you suspect that function name decoration is causing difficulty, inspect the shared library's exported functions. If the function name that appears in the function prototype section has characters such as `@` appended to it, the function was decorated when the DLL was built. This is most common with C++ compilers. In LabVIEW, the **Function name** control in the Call Library Function dialog box is a pull-down menu where you can access a list of all functions within the library you have selected. In addition, most operating systems have a utility you can use to view a library's exports, for example, QuickView on the Windows operating system and the `nm` command on most Linux systems. If the **Function name** list contains entries but the function you want to call does not appear in the list, the most likely reason is that the function has not been exported. Refer to the documentation for your compiler for information about how to mark functions for export.

Data Types

Your library call can fail when you do not use the correct data types. LabVIEW only supports basic numeric data types and C strings. Also, you can select **Adapt to Type** from the **Type** pull-down menu of the **Call Library Function** dialog box and direct LabVIEW to pass its own internal data types for a given parameter. You might encounter the following specific problems:

- **Non-Standard Data Type Definitions** —Frequently, other APIs use non-standard definitions for data types. For example, instead of using `char`, `short`, and `long`, the Windows API uses `BYTE`, `WORD`, and `DWORD`. If an API that you are using makes use of such data types, you need to [find the equivalent basic C data type](#) so that you can properly configure the Call Library Function Node.
- **Structure and Class Data Types** —Some APIs have structure and, in the case of C++, class data types. LabVIEW cannot use these data types. If you need to use a function that has a structure or class as an argument, you should write a shared library wrapper function that takes as inputs the data types that LabVIEW supports and that appropriately packages them before LabVIEW calls the desired function.
- **ActiveX Objects** —If you want to call a shared library that contains [ActiveX](#) objects, use the Automation Open function with the Property Node and the Invoke Node instead of the Call Library Function Node.

(Windows) Refer to the `labview\examples\Connectivity\Libraries and Executables\Libraries and Executables.lvproj` for an example of using data types in shared libraries.

Constants

Your library call can fail when your external code uses identifiers in place of constants. Many APIs define identifiers for constants to make the code easier to read. LabVIEW must receive the actual value of the constant rather than the identifier that a particular API uses. Constants are usually numeric, but they might also be strings or other values. To identify all constants, inspect the header file for the API to [find the definitions](#). The definition might either be in `#define` statements or in enumerations, which use the `enum` keyword.

Calling Conventions

Your library call can fail when certain operating systems use calling conventions other than the C calling convention and the Standard `__stdcall` calling convention. The calling convention defines how data is passed to a function and how clean up occurs after the function call is complete. The documentation for the API should say which calling convention(s) you must use. The Standard `__stdcall` calling convention is also known as the WINAPI convention, or the Pascal convention.

Use of calling conventions other than the C or Standard calling conventions frequently causes the failure of library calls in LabVIEW because those other calling conventions use an incompatible method for maintaining the stack.

Example 1: Call a Shared Library That You Built

This section describes the tasks necessary to complete the Array Average VI you started building in [Building a Shared Library to Call from LabVIEW](#) so the VI can call the `avg_num` function in `myshared.dll`.



NOTE

(Linux) The shared library file has a `.so` extension. You cannot call functions in static libraries `.a` files) on Linux. You can only call shared libraries `.so` files) dynamically.

You must complete the following tasks before the Array Average VI can call the `avg_num` function in `myshared.dll`.

- Complete configuration of the Call Library Function Node.
- Build the front panel window.
- Complete the block diagram.

Complete Configuration of the Call Library Function Node



NOTE

If you want to run applications or shared libraries created with different versions of LabVIEW on the same computer, the computer *must have a version of the LabVIEW Run-Time Engine that is compatible* with each version of LabVIEW used to create the applications or shared libraries.

Complete the following steps to complete the configuration of the Call Library Function Node.

1. Open the Array Average VI block diagram.
2. Right-click the Call Library Function Node and select **Configure** from the shortcut menu to open the Call Library Function dialog box.
3. Click the **Browse** button to the right of the **Library Name or Path** control to display a file dialog box.
4. Navigate to the location of your `myshared.dll` file.
5. Select `myshared.dll` and click the **Open** button. The file path to `myshared.dll` appears in the **Library Name or Path** control. The **Library Name or Path** control specifies the library name or path for the function you are calling. Enter only the library name if the library is in the search path of the system. Enter the entire path if the library is not in the search path of the system.

**NOTE**

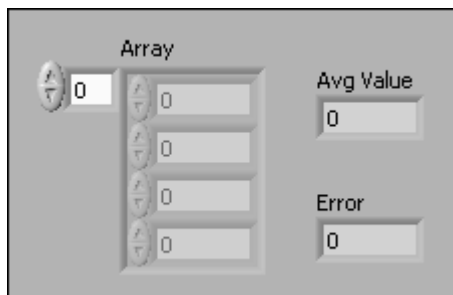
To make the reference platform independent, use the **wildcard*. Keep the C++ compiler from introducing platform dependence in exported function names through a process called name mangling by using the C++ compiler function export directive, `extern "C"{};` in your header file.

Build the Front Panel Window

Complete the following steps to create the front panel of the Array Average VI.

1. Place an array control on the front panel window and label it **Array**.
2. Place a numeric control in the array shell and resize the array to contain four elements.
3. Right-click the **Array** control and select **Representation»Single Precision** from the shortcut menu.
4. Place a numeric indicator on the front panel window and label it **Avg Value** to display the result of your averaging calculation.
5. Right-click the **Avg Value** indicator and select **Representation»Single Precision** from the shortcut menu.
6. Place a numeric indicator on the front panel and label it **Error** to display any errors that your VI generates.
7. Right-click the **Error** indicator and select **Representation»Long** from the shortcut menu.

The following front panel belongs to the Array Average VI.

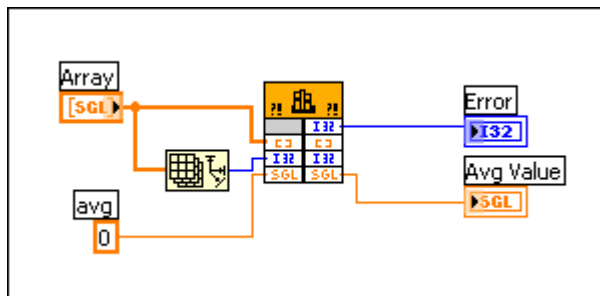
**Complete the Block Diagram**

Complete the following steps to complete the block diagram of the Array Average VI.

1. Wire **Array** to the **a** input of the Call Library Function Node.
2. Place an Array Size function on the block diagram.
3. Wire **Array** to the input of the Array Size function.
4. Wire the Array Size function output to the **size** input of the Call Library Function Node.

5. Right-click the **avg** input of the Call Library Function Node and select **Create»Constant** from the shortcut menu. Set the constant value to zero.
6. Wire the **avg** output of the Call Library Function Node to **Avg Value**.
7. Wire the **error** output of the Call Library Function Node to **Error**.

The following block diagram shows the completed Array Average block diagram.



Run the VI

On the front panel, enter values in **Array** and run the VI to calculate the average of those values. Save your work and close the VI.

If your shared library *returns incorrect results or crashes*, verify the data types and wiring to see if you wired the wrong type of information.

Example 2: Call a Hardware Driver API

You might want to access an API associated with hardware you have purchased. In this example, you call a hypothetical interface card for a databus called X-bus.



NOTE

You do not need to use the Call Library Function Node to gain access to the APIs of National Instruments hardware. All National Instruments products come with LabVIEW interfaces.

Configure the Call Library Function Node

The X-bus interface card comes with a software driver for your operating system. The X-bus documentation provides the following standard information:

- A listing of all functions you can use to access the hardware
- Description of the shared library file `xbus.dll` that contains these functions

- Instructions on including a header file `xbus.h`

**NOTE**

Although LabVIEW does not permit you to include such header files, you can open header files and extract information about function prototypes and constants.

- (Windows) A statement about the Standard `__stdcall` calling convention that the X-bus library uses

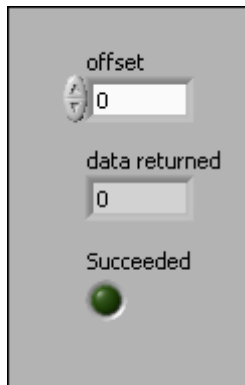
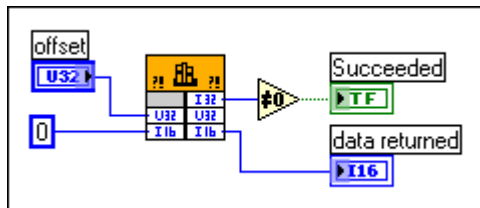
One of the functions you want to use with this hypothetical hardware is `XBusRead16`, which reads a 16-bit integer from a certain address. The documentation describes `XBusRead16` as follows:

<pre>long XBusRead16(unsigned long offset, short*data);</pre>	Puts 16 bits from the register at "offset" into the memory location pointed to by "data." Returns 1 if successful, or 0 if it fails.

Given the preceding information from the X-bus documentation, complete the following steps to configure the LabVIEW Call Library Function Node.

1. Open a new, blank VI and save the VI as Read Data.
2. Place a Call Library Function Node on the block diagram.
3. Right-click the Call Library Function Node object and select **Configure** from the shortcut menu.
4. Make the following settings in the Call Library Function dialog box:
 1. Enter `XbusRead16` in the **Function name** control.
 2. Select the **stdcall (WINAPI)** control from the **Calling Convention** field.
 3. Navigate to the **Parameters** page of the **Call Library Function** dialog box. Select **Numeric** from the **Type** pull-down menu for return type.
 4. Select **Signed 32-bit Integer** from the **Data Type** pull-down menu for return type.
 5. Add a parameter and name it **offset**.
 6. Select **Numeric** from the **Type** pull-down menu.
 7. Select **Unsigned 32-bit Integer** from the **Data Type** pull-down menu.
 8. Add a parameter and name it **data**.
 9. Select **Numeric** from the **Type** pull-down menu.
 10. Select **Signed 16-bit Integer** from the **Data Type** pull-down menu.
 11. Select **Pointer to Value** from the **Pass** pull-down menu.
5. Inspect the function prototype that appears in the **Function Prototype** field. If the prototype you see does not match the definition of the function in the API you are calling, you must change your settings in the **Call Library Function** dialog box.

Place a numeric control, numeric indicator, and round LED indicator on the front panel window. Label the control and indicators and complete the block diagram as shown in the following block diagram and front panel.



Example 3: Call the Win32 API

You might want to access the 32-bit Windows platform API (Win32 API). In Win32 environments, various DLLs permit your application to interact with the operating system and with the graphical user interface. Because the API offers thousands of functions, programmers must rely on the documentation for the Microsoft Software Development Kit (SDK). Microsoft Visual Studio products give you access to the SDK documentation. You also can access this information at the Microsoft Developer Network (MSDN) website.



TIP

Instead of using the Windows DLL as described in this example, you could easily create this message box in LabVIEW.

In this example, you call the Windows `MessageBox` function, a function which illustrates several of the typical complexities of the Win32 API. `MessageBox` is a simple SDK function that presents a small dialog box with a message, and has the following prototype:

```
int MessageBoxHWND hWnd, // handle to owner window

LPCTSTR lpText, // text in message box

LPCTSTR lpCaption, // message box title

UINT uType // message box style;
```

Notice the non-standard data types, such as `HWND` and `LPCTSTR`. The Win32 API uses hundreds of data types in the SDK, and very few of them are standard C data types. However, many of the non-standard data types are merely aliases for standard C data types. The API uses the aliases to identify the context of a particular data type. The following table lists the data types in the preceding prototype and the corresponding standard C data types.

WIN32 SDK Data Type	Standard C Data Type
<code>HWND</code>	<code>int **</code>
<code>LPCTSTR</code>	<code>const char *</code>
<code>UINT</code>	<code>unsigned int</code>

To properly call the `MessageBox` function in LabVIEW, you need to identify the equivalent LabVIEW data types, which you can usually infer from the C data types. Mapping `LPCTSTR` and `UINT` to LabVIEW is straightforward: `LPCTSTR` is a C String and `UINT` is a `U32`.

Mapping `HWND` is more complex. The previous table lists `HWND` as a double pointer to an integer. However, inspection of the function shows that `MessageBox` uses `HWND` merely as a reference number that identifies the owner of the window. Because of this fact, you do not need to know the integer value for which the `HWND` is a handle. Instead, you need to know the value of the `HWND` variable itself. Because it is a double pointer, and hence a pointer, you can treat it as a 32-bit unsigned integer, or in LabVIEW terms, a `U32`. Handles such as `HWND` are common in the Win32 SDK. In LabVIEW, you are almost always interested in the handle itself and not the data to which it points. Therefore, you can usually treat handles as `U32`. Handle names always begin with the letter H in the Win32 API.

If the SDK documentation does not make clear what C data type corresponds to a Win32 type, search `windef.h` for the appropriate `#define` or `typedef` statement.

The following table lists the Win32 SDK data types from the previous table and their corresponding LabVIEW data types.

WIN32 SDK Data Type	LabVIEW Data Type
<code>HWND</code>	<code>uInt32</code>
<code>LPCTSTR</code>	<code>CStr (C string pointer)</code>
<code>UINT</code>	<code>uInt32</code>

(Windows) Refer to the `labview\examples\Connectivity\Libraries and Executables\Libraries and Executables.lvproj` for a list of more Win32 API data types.

Constants

This section presents methods for finding the numerical values of constants in the Win32 API, using `MessageBox` constants as examples. The following table lists selected constants for `MessageBox`.

Constant	Description
<code>MB_ABORTRETRYIGNORE</code>	An Abort, Retry, Ignore message box.
<code>MB_CANCELTRYCONTINUE</code>	A Cancel, Try Again, Continue message box on Windows 2000. An alternative to <code>MB_ABORTRETRYIGNORE</code> .
<code>MB_HELP</code>	A Help button to add to a message box on Windows XP and later. The system sends a <code>WM_HELP</code> message to the owner whenever the user clicks the Help button or presses<F1>.
<code>MB_OK</code>	A message box with an OK button. This is the default message box.

In Visual Studio, programmers do not use the actual values of constants. In LabVIEW, however, you need to pass the actual numeric value of the constant to the function. You find these values in the header files that come with the SDK. The SDK documentation normally lists the relevant header file at the bottom of the help topic for a given function. For `MessageBox`, the SDK documentation specifies the header file as `winuser.h`.

The header file named in the preceding statement usually declares the constants. Searching through that header file, you should be able to find a `#define` statement or an enumeration that assigns the constant text a value. `winuser.h` defines values for some of the `MessageBox` constants as follows:

```
#define MB_OK 0x00000000L
#define MB_ABORTRETRYIGNORE 0x00000002L
#define MB_ICONWARNING MB_ICONEXCLAMATION
```

Thus, `MB_OK` has the decimal value 0. `MB_ABORTRETRYIGNORE` has the decimal value 2. `MB_ICONWARNING` is defined as `MB_ICONEXCLAMATION`. Elsewhere in `winuser.h` you find the following statement defining `MB_ICONEXCLAMATION`:

```
#define MB_ICONEXCLAMATION 0x00000030L
```

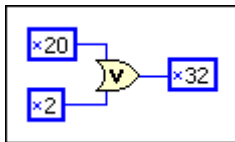
A hexadecimal value of 30 translates to a decimal value of 48.

Keep in mind that constants in the SDK often are used in bitfields. A bitfield is usually a single integer in which each bit controls a certain property. The `uType` parameter in `MessageBox` is an example of a bitfield. Often, you can combine multiple constants in order to set multiple properties through one parameter. In order to combine these

constants, you use a bitwise OR operation `|`. For example, to set the `MessageBox` to have a warning icon and the buttons **Abort**, **Retry**, and **Ignore**, you pass the following value of `uType` to `MessageBox`:

`MB_ABORTRETRYIGNORE | MB_ICONEXCLAMATION = 0x32`

In LabVIEW, you combine multiple constants by wiring integer types to the OR operator, as shown in the following block diagram.



Determining the Proper Library and Function Name

Before you can configure the call to the Win32 API, you must identify the DLL that contains `MessageBox` and the specific name of `MessageBox` within the DLL. Refer to the description of `MessageBox` in the documentation that comes with your SDK or search for `MessageBox Function ()` on the Microsoft Web site. The Requirements section of the documentation on the Microsoft Web site contains the following information:

Minimum supported client	Windows 2000 Professional
Minimum supported server	Windows 2000 Server
Header	Winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
Unicode and ANSI names	MessageBoxW (Unicode) and MessageBoxA (ANSI)

The DLL line specifies `user32.dll` as the DLL that contains `MessageBox`. The Unicode and ANSI names line specifies the name of the `MessageBox` function for each type of character encoding.

Unicode Versions and ANSI Versions of Functions

`MessageBox` uses two string arguments. The SDK implements two versions of functions that use string arguments, a Unicode version and an ANSI version. One of the items in the Requirements section of the `MessageBox` documentation says, “Unicode: Implemented as Unicode and ANSI version on Windows.” You can distinguish the two versions in the DLL because each has a W (Unicode) or an A (ANSI) appended to the end of the function name. `winuser.h` contains the following code:

```
#ifdef UNICODE
```



```
#define MessageBox MessageBoxW

#else

#define MessageBox MessageBoxA

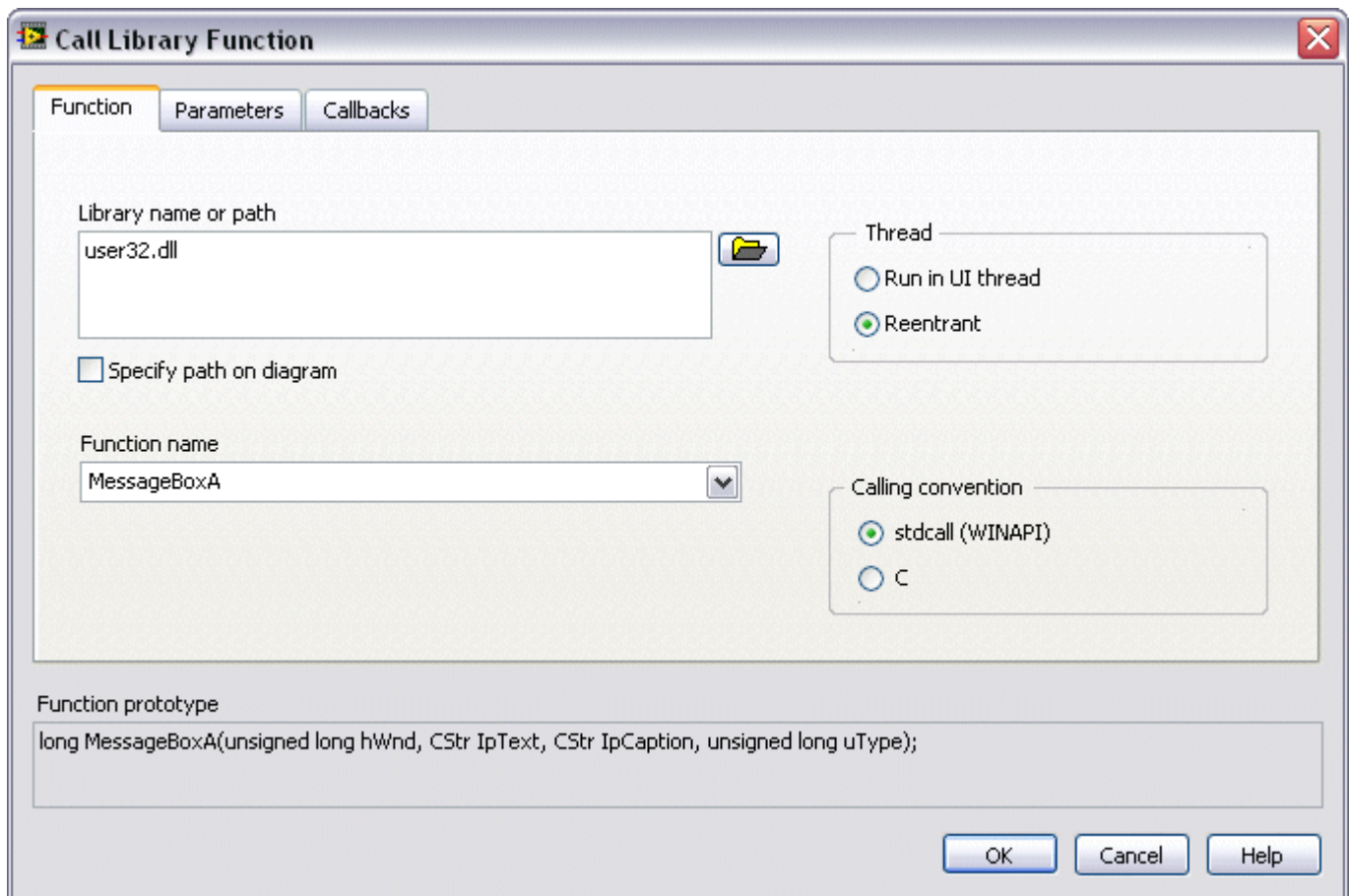
#endif // !UNICODE
```

The preceding code defines `MessageBox` to be either `MessageBoxA` or `MessageBoxW`, depending on whether the application is a Unicode application. In effect, a `MessageBox` function does not exist in `user32.dll`. Instead, `user32.dll` contains a function `MessageBoxA` and a function `MessageBoxW`. In most cases in LabVIEW, you use the ANSI version of the function because the LabVIEW strings are based on ANSI, not Unicode. For this example, you use the `MessageBoxA` function.

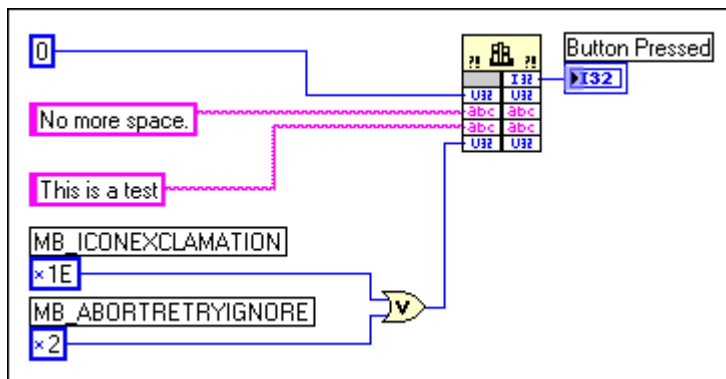
Configuring a Call to the Win32 API

Now that you are familiar with many aspects of the Win32 API, you can configure a LabVIEW *Call Library Function Node* to call the `MessageBox` function. Remember that you must use the Standard `__stdcall` calling convention in calls to any function in the Windows SDK.

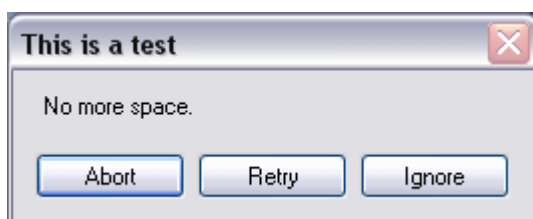
The following front panel shows a *correctly configured instance* of the Call Library Function Node. Make your Call Library Function dialog box match the settings in the graphic.



The following block diagram shows a VI designed to call the Win32 API. Configure your block diagram to match the following block diagram.



The VI generates the following message box.



Additional Windows Examples of LabVIEW Calls to DLLs

(Windows) The following examples can help you learn more about calling DLLs from LabVIEW:

- The Call DLL VI located in `labview\examples\Connectivity\Libraries and Executables\Libraries and Executables.lvproj` allows you to browse examples of C and C++ external code data types and how they interface with LabVIEW.
- The Sound Player VI located in the `labview\examples\Graphics and Sound\Sound` directory lets you play Windows .WAV sound files on your computer from LabVIEW, if you have a sound card with Windows sound drivers installed on your system.
- If you do not have a sound card you can generate a sound in your PC speaker by calling the `MessageBeep` function in `User32 . DLL`. The function prototype is as follows:

```
VOID MessageBeep(UINT uType);
```

- You can programmatically position your cursor anywhere on your monitor using the `SetCursorPos` function in `User32 . DLL`. The function prototype is as follows:

```
BOOL SetCursorPos(INT x, INT y);
```

`x` and `y` are the coordinates you want, referenced from the upper left corner of the screen. The return value is `TRUE` if the function was successful and `FALSE` if it was unsuccessful. Remember that the value returned is type `BOOL`, which is defined in the Win32 API as a 32-bit signed integer with values 0 = `FALSE` and 1 = `TRUE`.

Debugging Shared Libraries and Calls to Shared Libraries

When you debug your LabVIEW calls to shared libraries, you must be prepared to trace problems in the shared library you are calling and in the implementation of the Call Library Function Node in LabVIEW.

Troubleshooting the Call Library Function Node

When the LabVIEW calls to shared libraries generate errors, check for the following problems in your use of the Call Library Function Node:

- Verify the Call Library Function Node has the correct name or path for the shared library. Also ensure that you specify the location on disk appropriately for your use case. Some use cases require you to specify the shared library by path, and other use cases require you to specify the shared library by name.

- For LabVIEW-built shared libraries, make sure the computer has a version of the LabVIEW Run-Time Engine that is compatible with each version of LabVIEW used to create the shared libraries.
- If you receive the error message `Function not found in library`, verify the Call Library Function Node has the correct spelling, syntax, and case sensitivity for the function name that you are calling.
- Be sure that the compiler has not decorated the function.
- If you receive an error message that a secondary shared library cannot be found, yet you properly specified the path to the primary shared library in the Call Library Function Node, the primary shared library needs additional functions from one or more other shared libraries. You need to find the other shared libraries and place them in the same directory as the shared library that needs them or in a directory that is in the search path. Refer to the KnowledgeBase at ni.com for more information.
- If your VI crashes, verify that return types and data types of arguments for functions in the Call Library Function Node exactly match the data types your function uses. Also, verify the Call Library Function Node uses the proper calling convention (C or `__stdcall`) using the Error Checking Level tab in the Call Library Function dialog box.
- Verify that all the parameters are defined to be passed by the correct method, such as value or pointer. Also, verify the Call Library Function Node passes arguments to the function in the correct order.
- For arrays or strings of data, always pass a buffer or array that is large enough to hold any results that the function places in the buffer. However, if you are passing them as LabVIEW handles, use LabVIEW Manager functions to resize them under Visual C++ or Xcode compilers.
- If you receive a message, the cause is almost always an error in the code of the shared library, such as writing past the end of the memory allocated for an array. Notice that these kinds of crashes might or might not occur at the time the shared library call actually executes on the block diagram.
- Use the Automation Open function with the Property Node and the Invoke Node instead of the Call Library Function Node if you want to call a shared library that contains ActiveX objects.
- All calls to LabVIEW-built shared libraries should specify **Run in any thread**. If you configure the Call Library Function Node using LabVIEW-built shared libraries and specify **Run in UI thread**, LabVIEW might hang and require you to restart.
- Pascal strings do not exceed 255 characters in length.
- Resizing of arrays and strings can take place only when the Call Library Function Node passes a LabVIEW array handle or LabVIEW string handle. Resize arrays and strings using functions exported through `labviewv.lib` to a Visual C++ project, and `liblveports.a` to an Xcode project.
- Remember that C strings are NULL terminated. If your DLL function returns numeric data in a binary string format, for example, through GPIB or the serial port, it might return NULL values as part of the data string.

Related Information

Calling Shared Libraries in LabVIEW

Configure the Call Library Function Node

Call Library Function Node

Call Library Function Dialog Box

LabVIEW Manager Functions

Specifying the Location of Shared Libraries on Disk

Using the LabVIEW Run-Time Engine

Building a Function Prototype

Automation Open Function

Property Node

Invoke Node

ActiveX Container Overview

Troubleshooting Your Shared Library

Check for the following problems in your shared library when LabVIEW calls to shared libraries generate errors:

- Verify that shared library functions that other applications call appear in the module definition file EXPORTS section, or you include the `_declspec (dllexport)` keyword in the function declaration. Keep the C++ compiler from introducing platform dependence in exported function names through a process called name mangling by using the C++ compiler function export directive, `extern "C"{}` , in your header file, as shown in the following example code:

```
extern "C" {

    /* your function prototypes here */

}
```

- Remember that you need to declare the function with the `_declspec (dllexport)` keyword in the header file and the source code, or define it in the EXPORTS section of the module definition file.
- When you use the `_declspec (dllexport)` keyword and you are also using the `__stdcall` calling convention, you must declare the shared library function name in the EXPORTS section of the module definition `.def` file. In the absence of a `.def` file, `__stdcall` might truncate function names in an unpredictable pattern, so the actual function name is unavailable to applications that call the shared library.

- When a function has not been properly exported, you must recompile the shared library. Never recompile the shared library while the shared library is loaded into memory by another application, for example, by your VI. Before recompiling a shared library, make sure that all applications making use of the shared library are unloaded from memory. This ensures that the shared library itself is not loaded into memory during a recompile. The shared library might fail to rebuild correctly if you forget this point and your compiler does not warn you. However, most compilers warn you when the shared library is in use by an application.
- Try to debug the shared library by using the source level debugger provided with your compiler. Using the debugger of your compiler, you can set breakpoints, step through your code, watch the values of the variables, and so on. Debugging using conventional tools can be extremely beneficial. Refer to the appropriate manual for your compiler for more information about debugging.
- Calling the shared library from another C program is also another way to debug the shared library. By calling the shared library from another C program, you have a means of testing the shared library independent of LabVIEW, thus helping you to identify any problems, sooner.
- When calling a LabVIEW shared library that passes a 2D array, you must first declare the handler variable and initialize the variable to NULL, as shown in the following C code:

```
main

{

/*LabVIEW data handler variable for the array */

TD1Hd1myArray = NULL;

...

/* Call to the LabVIEW shared library function */

DLLFunctionalCall(&myArray);

...

}
```

If you do not initialize the handler variable to NULL, the code produces a `General Protection Fault` when you call the shared library.

- To allow LabVIEW shared library functions to execute without interruption, LabVIEW delays the processing of operating system messages until the end of any calls to shared library functions or until you load a modal window from the shared library. Delaying the operating system messages, such as keyboard messages from the users, is useful in order to avoid calling the same shared library file while a shared library function runs. For example, if the shared library function is called in response to the user pressing a button, the user should not be able to press the button again until the shared library function has completed. When LabVIEW delays the processing of operating system messages, all function calls to the same LabVIEW shared library are put into a queue. That queue will be executed after the original

shared library function call completes. A modal window is a type of window that remains active or remains on top of all other LabVIEW windows until you close the window or open another modal window. If you load a modal window from the shared library, LabVIEW overrides the process delay and processes the messages to allow the modal window to become active. You cannot interact with other windows while a modal window is open. Most dialog boxes in LabVIEW are modal windows. You cannot open a non-modal window from a LabVIEW callback VI nor a shared library while any other process is running. If you want your callback VI or shared library to call a non-modal window, you must do so programmatically by completing the steps outlined in the Callback VIs topic.

- You can choose whether to delay operating system messages in shared libraries that you build. Before building the shared library, navigate to the **Advanced** page of the Shared Library Properties dialog box and remove the checkmark from the **Delay operating system messages in shared library** checkbox to process operating system messages while shared library functions run.
- The following scenarios might cause the `No debuggable applications or runtime libraries found` error to appear.
 - The debuggable shared library was not loaded or was unloaded by the application.
 - The configuration `ini` file of the shared library, created by the Application Builder, was not distributed with the shared library.

Related Information

Callback VIs

Shared Library Properties Dialog Box

Module Definition Files

When you [build a shared library](#), you configure LabVIEW to use the C calling convention in the `.c` source file you built with the [Call Library Function Node](#). In contrast, you use the `__stdcall` calling convention when you call the Win32 API. When you build a shared library with `__stdcall`, you normally use a module definition `.def` file to export the functions in your shared library. In the absence of a `.def` file, `__stdcall` might truncate function names in an unpredictable pattern, so the actual function name would be unavailable to applications that call the shared library.

You can associate a `.def` file with a shared library. The `.def` file contains the statements for defining a shared library, such as the name of the shared library and the functions that it exports, as shown in the following example code:

```
LIBRARY myshared
```

```
EXPORTS
```

```
avg_num
```

The preceding code example demonstrates the following key requirements for `.def` files:

- The only mandatory entries in the `.def` files are the `LIBRARY` statement and the `EXPORT` statement.
- The `LIBRARY` statement must be the first statement in the file.
- The name you specify in the `LIBRARY` statement identifies the library in the import library of the shared library.
- The names you specify in the `EXPORTS` statement identify the functions that the shared library exports.



NOTE

Instead of a `.def` file, many Windows programmers use the **LINK** option in Project Settings of the Visual C++ compiler to obtain equivalent command-line options for most module definition statements.

Using Arrays and Strings in the Call Library Function Node

This section reviews important concepts regarding array and string data in the [Call Library Function Node](#).

(Windows) Refer to the `labview\examples\Connectivity\Libraries and Executables\Libraries and Executables.lvproj` for an example of using arrays and strings in shared libraries (DLLs).

Arrays of Numeric Data

Arrays of numeric data can be comprised of any type of integers or floating point numbers with single (4-byte) or double (8-byte) precision. When you pass an array of data to a shared library function, you can pass the data as an array data pointer, as a LabVIEW array handle, or as a LabVIEW array handle pointer.

Array Data Pointers have the following characteristics whether you pass the Array Data Pointers in the Windows API or in another API:

- You can set the number of dimensions in the array, but you must not include information about the size of the array dimension(s). Instead, you must pass the size of the array dimension(s) information to your shared library in a separate variable.
- Never resize an array or perform operations that might change the length of the array data passed from LabVIEW. Resizing might cause a crash because the pointer sent is not an allocated block but points into the middle of an allocated block.
- To return an array of data, you should allocate an array of sufficient size in LabVIEW, pass the array to your function, and have this array act as the buffer. If the data takes less space, you can return the correct size as a separate parameter and then, on the calling diagram, use array subset to extract the valid data.

Remember that the Windows API does not use LabVIEW array handles, so with functions that are part of the Windows API you can use only Array Data Pointers.

If you pass the array data as a LabVIEW array handle, you can use LabVIEW Manager functions, such as `NumericArrayResize`, to resize the handle. If you need to resize a handle to an array of strings or clusters, consider alignment on different platforms.

Related Information

LabVIEW Manager Functions

`NumericArrayResize` Function

String Data

The types of your string pointers must match the types of string pointers that your function uses, or errors occur. The Call Library Function Node offers the following choices:

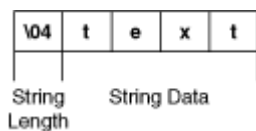
- **C String Pointer** is a pointer to the string, followed by a NULL character. Most Win32 API functions use this C-style string pointer.
- **Pascal String Pointer** is a pointer to the string, preceded by a length byte.
- **LabVIEW String Handle** is a pointer to a pointer to the string, preceded by four bytes of length information.
- **LabVIEW String Handle Pointer** is a pointer to a handle for a string, preceded by four bytes of length information.

You can think of a string as an array of characters. Assembling the characters in order forms a string. LabVIEW stores a string in a special format in which the first four bytes of the array of characters form a 32-bit signed integer that stores how many characters appear in the string. Thus, a string with *n* characters requires *n* + 4 bytes to store in memory. For example, in the following illustration, the string `text` contains four characters.

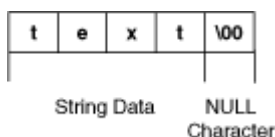
\00	\00	\00	\04	t	e	x	t
String Length				String Data			

When LabVIEW stores the string, the first four bytes contain the value 4 as a 32-bit signed number, and each of the following four bytes contains a character of the string. The advantage of this type of string storage is that NULL characters are allowed in the string. Strings are virtually unlimited in length, up to 2³¹ characters. This method of string storage is shown in the previous illustration. If you pass a LabVIEW string handle from the Call Library Function Node to the shared library, then you can use the LabVIEW Manager functions, such as `DSSetHandleSize`, to resize the LabVIEW string handle.

The Pascal string format is nearly identical to the LabVIEW string format, but instead of storing the length of the string as a 32-bit signed integer, the string length is stored as an 8-bit unsigned integer. Storing the string length as an 8-bit unsigned integer limits the length of a Pascal-style string to 255 characters. A Pascal string that is *n* characters long will require *n* + 1 bytes of memory to store. The following illustrations shows a Pascal string.



C strings are probably the type of strings you will deal with most commonly. The similarities between the C-style string and normal numeric arrays in C becomes much more clear when you see that C strings are declared as `char*`, where `char` is typically an 8-bit unsigned integer. Unlike LabVIEW and Pascal strings, C strings do not contain any information that directly gives the length of the string. Instead, C strings use a special character called the NULL character to indicate the end of the string. NULL is defined to have a value of zero in the ASCII character set. Notice that NULL is the number zero and not the character “0.” Thus, in C, a string that contains n characters requires $n + 1$ bytes of memory to store: n bytes for the characters in the string and one additional byte for the NULL termination character. The following illustration shows how a C-style string is stored in memory.



The advantage of C-style strings is that they are limited in size only by available memory. However, if you are acquiring data from an instrument that returns numeric data as a binary string, as is common with serial or GPIB instruments, values of zero in the string are possible. If you treat the string as a C-style string, your program incorrectly assumes that the end of the string has been reached, when in fact your instrument is returning a numeric value of zero. For binary data that might contain NULL values, consider using an array of 8-bit unsigned integers.

Observe the following guidelines when passing string data to a shared library:

- Never resize a string, concatenate a string, or perform operations that might increase the length of string data passed from LabVIEW if you are using the C or Pascal string pointers.
- If you must return data as a string, allocate a string of the appropriate length in LabVIEW and pass this string into the shared library to act as a buffer. You can specify the string size and allocate the correct amount of memory with the **Minimum size** control on the **Parameters** tab of the Call Library Function dialog box. If you pass in a string that is smaller than the **Minimum size**, LabVIEW enlarges the size of the string to the minimum.
- If you pass a LabVIEW string handle from the Call Library Function Node to the shared library, you can use the LabVIEW Manager functions, such as `DSSetHandleSize`, to resize the LabVIEW string handle. Consider alignment on different platforms before resizing the string handle.

Related Information

Call Library Function Dialog Box

LabVIEW Manager Functions

DSSetHandleSize Function

Determining When to Resize Array and String Handles in Shared Libraries

When a shared library function increases the size of a string or array that you passed by handle, you need to adjust the size of the handle that references the data. You can adjust the handle size using the memory manager routine `DSSetHandleSize`. Because this routine requires you to calculate the size of the new memory space, you must account for platform-dependent requirements about how you align and pad memory. A piece of data is aligned if it falls on a boundary equal to a multiple of its size. For example, a 2-byte value is at a memory address that is a multiple of 2 and a 4-byte value is at a memory address that is a multiple of 4. On Intel processors the maximum alignment is 4.

The following examples highlight alignment differences on various platforms for a three-dimensional array of clusters, with each cluster containing a double-precision, floating-point number and a 1-byte Boolean:

- (Windows) The first twelve bytes describe the size of each dimension of the array (four bytes per dimension). Cluster elements follow, with each cluster element consisting of an 8-byte double-precision floating-point number followed by a 1-byte Boolean. LabVIEW ignores alignment on Windows.
- (64-bit LabVIEW running on Windows) The first twelve bytes describe the size of each dimension of the array (four bytes per dimension) followed by four bytes of padding. Cluster elements follow, with each cluster element consisting of an 8-byte double-precision floating-point number followed by a 1-byte Boolean and seven bytes of padding.
- (OS X and Linux) The first twelve bytes describe the size of each dimension of the array (four bytes per dimension). Cluster elements follow, with each cluster element consisting of an 8-byte double-precision floating-point number followed by a 1-byte Boolean and three bytes of padding.



NOTE

Although you must use `DSSetHandleSize` to resize a string or an array of arbitrary data type, you can use the `NumericArrayResize` manager routine to resize arrays of numeric data. `NumericArrayResize` automatically accounts for platform-dependent alignment requirements.

Related Information

[DSSetHandleSize Function](#)

[NumericArrayResize Function](#)

[How LabVIEW Stores Data in Memory](#)

[How LabVIEW Stores Clusters in Memory](#)

Array and String Tip

When you are not passing LabVIEW handles and your shared library function must create an array, resize an array, or resize a string of data, you should break the shared library function into the following two shared library functions:

- Determine the number of elements that the array requires or the length of the string to be returned. Have the first shared library function return the desired size to LabVIEW.
- In LabVIEW, initialize an array or string with default values and pass this array to the second shared library function in your shared library, which actually places the data into the array. If you are working with string-based instrument control, it might be easier to pass an array of 8-bit integers than C strings because of the possibility of having NULL values in the string. When you are passing a LabVIEW array handle or LabVIEW string handle from the Call Library Function Node to your shared library, you can use the LabVIEW Manager functions to resize or create an array or string.

Related Information

[Completing the .c File](#)

Importing Shared Libraries

This book describes the Import Shared Library wizard you can use to create or update a LabVIEW project library of wrapper VIs for functions in a Windows .dll file, an OS X .framework file, or a Linux .so file.



(Windows) To view related topics, click the **Locate** button, shown at left, in the toolbar at the top of this window. The LabVIEW Help highlights this topic in the **Contents** tab so you can navigate the related topics.

Advantages of the Import Shared Library Wizard Compared to the Call Library Function Node

Although you can use either the Call Library Function Node or the **Import Shared Library** wizard to call shared library functions, the **Import Shared Library** wizard can accelerate the process of integrating the shared library functions and reduce the potential for errors when calling these libraries.

If you provide the name of a shared library file and a header .h file, the wizard parses the header file and generates a wrapper VI for each function in the shared library, which the wizard saves in a LabVIEW project library .lvlib. The VIs that the wizard generates each contain a Call Library Function Node that accesses the selected function from the shared library.

The **Import Shared Library** wizard has the following advantages compared to using the Call Library Function Node directly:

- The wizard automatically configures the Call Library Function Nodes generated in the wrapper VIs by the wizard. Optionally, you also can customize the configuration for this Call Library Function Node.
- You can use the wizard to configure calls to multiple functions or entire libraries at once. With the Call Library Function Node, you must configure each node to call a function in a shared library individually.
- Because the wizard generates the wrapper VIs in the shared library you specify, you can easily locate and update those VIs. For example, if you want to add or remove some of the functions you already imported from the shared library, you can run the **Import Shared Library** wizard again to update the previously generated wrapper VIs. You can run the wizard multiple times on the same shared library file.
- The wizard converts the data types in the shared library to LabVIEW data types. For example, the wizard automatically generates cluster data types for any structure data types in the shared library functions. With the Call Library Function Node, you must manually convert the structure data type.

If you want to individually configure each call to a shared library or if you want to customize configurations for calling shared libraries, use the Call Library Function Node to call libraries directly. Otherwise, use the **Import Shared Library** wizard to automate this process.



TIP

If you want to import a shared library file for an instrument driver, you can download the LabVIEW Instrument Driver Import Wizard from ni.com.

Related Information

[Using the Import Shared Library Wizard](#)

[Importing Functions from a Shared Library File](#)

[Example: Importing Functions from a Shared Library File \(Windows\)](#)

[Troubleshooting the Import Shared Library Wizard](#)

Call Library Function Node

Supported Data Types for the Import Shared Library Wizard

The LabVIEW Import Shared Library wizard converts the following shared library data types to the indicated LabVIEW data types.

- numeric and numeric pointer
- string and string pointer
- void and void pointer

- Boolean and Boolean pointer
- cluster and cluster pointer

For return values in the shared library functions, the Input/Output type can only be Output. The Configure VIs and Controls page of the **Import Shared Library** wizard displays an exclamation point  icon to indicate that LabVIEW cannot process the specified data type. For data types that LabVIEW does not support, [create a custom control and link to it](#). Refer to the [Example: Importing Functions from a Shared Library File](#) topic for more details.

Numeric

C Data Type	Icon	I/O Type	LabVIEW Control Type	Pass Type	LabVIEW Representation	Unsigned Representation (unsigned int)
__int8		Input	Numeric	Value	I8	U8
__int16		Input	Numeric	Value	I16	U16
__int32		Input	Numeric	Value	I32	U32
__int64		Input	Numeric	Value	I64	U64
char		Input	Numeric	Value	I8	U8
short		Input	Numeric	Value	I16	U16
int		Input	Numeric	Value	I32	U32
long		Input	Numeric	Value	I32	U32
float		Input	Numeric	Value	SGL	
double		Input	Numeric	Value	DBL	

Numeric Pointer

C Type	Icon	I/O Type	I/O Type (const int*)	LabVIEW Control Type	Pass Type	LabVIEW Representation	Unsigned Representation (unsigned int*)
*__int8		Input/Output	Input	Numeric	Pointer	I8	U8
*__int16		Input/Output	Input	Numeric	Pointer	I16	U16
*__int32		Input/Output	Input	Numeric	Pointer	I32	U32

C Type	Icon	I/O Type	I/O Type (const int*)	LabVIEW Control Type	Pass Type	LabVIEW Representation	Unsigned Representation (unsigned int*)
__int64 *		Input/Output	Input	Numeric	Pointer	I64	U64
short *		Input/Output	Input	Numeric	Pointer	I16	U16
int *		Input/Output	Input	Numeric	Pointer	I32	U32
long *		Input/Output	Input	Numeric	Pointer	I32	U32
float *		Input/Output	Input	Numeric	Pointer	SGL	
double *		Input/Output	Input	Numeric	Pointer	DBL	

String and String Pointer

C Type	Icon	I/O Type	LabVIEW Control Type
char*		Input/Output	String
Char []		Input/Output	String
const char *		Input	String
const char []		Input	String

Void and Void Pointer

C Type	Icon	I/O Type	LabVIEW Control Type	Pass Type	LabVIEW Representation
void		void means no control is needed and can appear only as the return control			
void*		Input/Output	Numeric	Value	U32
const void *		Input	Numeric	Value	U32

Boolean and Boolean Pointer

C Type	Icon	I/O Type	LabVIEW Control Type	Pass Type
bool		Input	Boolean	Value
bool *		Input/Output	Boolean	Pointer
const bool *		Input	Boolean	Pointer

Cluster and Cluster Pointer

C Type	Icon	I/O Type	LabVIEW Control Type	Pass Type
struct		Input	Cluster	Value
Struct *		Input/Output	Cluster	Pointer
const struct *		Input	Cluster	Pointer

Using the Import Shared Library Wizard

Use the Import Shared Library wizard to [create or update a LabVIEW project library of wrapper VIs](#) for functions in a Windows .dll file, an OS X .framework file, or a Linux .so file.

The **Import Shared Library** wizard supports most C and C++ header files. The wrapper VIs the wizard creates use the Call Library Function Node, which does not support the C++ this pointer or calling methods in C++ classes.



NOTE

If you want to import a shared library file for an instrument driver, you can download the LabVIEW Instrument Driver Import Wizard from ni.com.

The **Import Shared Library** wizard parses the header file, lists the functions in the shared library, [converts data types in the shared library to LabVIEW data types](#), and generates a wrapper VI for each function. The wizard saves the VIs in a LabVIEW project library that you can edit and creates an HTML report about the generated library that you can launch when you complete the wizard.

In the wizard, you can specify include paths and preprocessor definitions, configure the individual VIs that wrap each function, and configure memory allocation and error handling. The wizard also creates [custom controls](#) for structure elements in the original functions and adds the controls to the project library. You can use the custom controls to modify all the VIs in the library that contain the corresponding data type.

You can run the wizard multiple times on the same shared library file. If you select the **Update VIs for a shared library** option on the **Specify Create or Update Mode** page, the

wizard overwrites the previous version of the project library file and the existing VIs within that file. If you choose not to re-import generated VIs within the project library file, the VIs remain unchanged in the directory. The wizard retains the most recent settings for each individual function in a particular shared library. For example, if you have a shared library that contains three functions, you might update only the second function. The next time you run the wizard on that shared library file, it retains the original settings for functions one and three and the new settings for function two.

Select **Tools»Import»Shared Library** to launch the **Import Shared Library** wizard. Follow the prompts to create wrapper VIs for shared library files. You must provide the name of a shared library file and a header .h file for the wizard to parse.

Refer to the [Importing Functions from a Shared Library File](#) topic for step-by-step instructions for importing a shared library and creating wrapper VIs.

LabVIEW Manager Functions in Shared Libraries

Using LabVIEW Manager Functions in Shared Libraries

LabVIEW has a suite of functions that you can call from shared libraries. This suite of functions performs user-specified routines using the appropriate instructions for a given platform. These routines, which manage the functions of a specific operating system, are grouped into the following categories:

- Memory manager
- File manager
- Support manager

Portability of LabVIEW Manager Functions

External code written using the LabVIEW managers is portable, that is, you can compile it without modification on any platform that supports LabVIEW. This portability has the following advantages:

- The LabVIEW application is built on top of the manager functions. Except for the manager functions, the LabVIEW source code is identical across platforms.
- The analysis VIs rely mainly on manager functions. Therefore, their source code is the same for all platforms.



NOTE

When you call the LabVIEW manager functions from a shared library, use `#include extcode.h` in any files that use manager functions and link to `labviewv.lib`. Set the structure alignment of the compiler to 1 byte.

To achieve platform independence, data types should not depend on the peculiarities of various compilers. For example, the C language does not define the size of an integer. Without an explicit definition of the size of each data type, you have difficulty creating code

that works identically across multiple compilers. LabVIEW managers use data types that explicitly indicate the size of the data type. For example, if a routine requires a 4-byte integer as a parameter, you define the parameter as an `int32`. The managers define data types in terms of the fundamental data types for each compiler. Thus, on one compiler, the managers might define an `int32` as an `int`, while on another compiler, the managers might define an `int32` as a `long int`. When you write external code, use the manager data types instead of the host computer data types so your code is more portable and has fewer errors.

Pointers as Parameters

Some manager functions have a parameter that is a pointer. These parameter type descriptions are identified by a trailing asterisk, such as the **ph** parameter of the `DSCopyHandle` allocating and releasing function, or are type defined as such, such as the **name** parameter of the `FNamePtr` function. In most cases, the manager function writes a value to pre-allocated memory. In some cases, such as `FStrFitsPath` or `GetALong`, the function reads a value from the memory location, so you do not have to pre-allocate memory for a return value.

The following functions have parameters that return a value for which you must pre-allocate memory.

<code>DSMemStats</code>	<code>FNamePtr</code>
<code>DSCopyHandle</code>	<code>FNewRefNum</code>
<code>DateToSecs</code>	<code>FPathToArr</code>
<code>DSMemStats</code>	<code>FPathToAZString</code>
<code>FCreate</code>	<code>FPathToDSSString</code>
<code>FCreateAlways</code>	<code>FPathToPath</code>
<code>FFlattenPath</code>	<code>FRefNumToFD</code>
<code>FGetAccessRights</code>	<code>FStringToPath</code>
<code>FGetEOF</code>	<code>FTextToPath</code>
<code>FGetInfo</code>	<code>FUnflattenPath</code>
<code>FGetPathType</code>	<code>GetAlong</code>
<code>FMOpen</code>	<code>SetAlong</code>
<code>FMRead</code>	<code>RandomGen</code>
<code>FMTell</code>	<code>SecsToDate</code>
<code>FMWrite</code>	<code>NumericArrayResize</code>

You must allocate space for this return value. The following examples illustrate incorrect and correct ways to call one of these functions from within a generic function `foo`.

Incorrect example:

```
foo(Path path) {

PStr p; /* an uninitialized pointer */ File *fd; /* an
uninitialized pointer */

MgErr err;

err = FNamePtr(path, p);

err = FMOpen(fd, path, openReadOnly denyWriteOnly);

}
```

In the incorrect example, `p` is a pointer to a Pascal string, but the pointer is not initialized to point to any allocated buffer. `FNamePtr` expects its caller to pass a pointer to an allocated space and writes the name of the file referred to by `path` into that space. Even if the pointer does not point to a valid place, `FNamePtr` writes its results there, with unpredictable consequences. Similarly, `FMOpen` writes its results to the space to which `fd` points, which is not a valid place because `fd` is uninitialized.

Correct example:

```
foo(Path path) {

Str255 buf; /* allocated buffer of 256 chars */

File fd;

MgErr err;

err = FNamePtr(path, buf);

err = FMOpen(&fd, path, openReadOnly, denyWriteOnly);

}
```

In the correct example, `buf` contains space for the maximum-sized Pascal string, whose address is passed to `FNamePtr`. `fd` is a local variable (allocated space) for a file descriptor.



RELATED INFORMATION

[Memory Manager](#) on page 1672

[File Manager](#) on page 1676

[Support Manager](#) on page 1680

[LabVIEW Manager Data Types](#) on page 1663

LabVIEW Manager Data Types

[LabVIEW Manager](#) data types include the following data types:

- Scalar
- Char
- Dynamic
- Memory-related
- Constants

Scalar

Scalar data types include Boolean and numeric.

Boolean

External code modules work with two kinds of Boolean scalars—those existing in LabVIEW block diagrams and those passing to and from manager routines. The manager routines use a conventional Boolean form where 0 is FALSE and 1 is TRUE. This conventional Boolean form is called a `Bool32` and is stored as a 32-bit value.

LabVIEW block diagrams store Boolean scalars as 8-bit values. The value is 0 if FALSE and 1 if TRUE. This Boolean form is called an `LVBoolean`.

The following table describes the two forms of Boolean scalars.

Name	Description
<code>Bool32</code>	32-bit integer, 0 if FALSE, 1 if TRUE
<code>LVBoolean</code>	8-bit integer, 0 if FALSE, 1 if TRUE

Numeric

The manager data types support 8-, 16-, 32-, and 64-bit signed and unsigned integers.

LabVIEW supports the following single, double, and extended floating-point data types.

Type	Description
Single	32-bit
Double	64-bit
Extended	Up to 80-bit

LabVIEW supports complex numbers containing two floating-point numbers, with different complex numeric types for each of the floating-point data types.

LabVIEW supports the following basic data types for numbers:

- Signed integers
 - `int8` 8-bit integer
 - `int16` 16-bit integer
 - `int32` 32-bit integer
 - `int64` 64-bit integer
- Unsigned integers
 - `uInt8` 8-bit integer
 - `uInt16` 16-bit integer
 - `uInt32` 32-bit integer
 - `uInt64` 64-bit integer
 - `size_t` 32-bit integer
- Floating-point numbers
 - `float32` 32-bit floating-point number
 - `float64` 64-bit floating-point number
 - `floatExt` extended-precision floating-point number

The following table explains how various platforms store extended-precision numbers.

Platform	Storage Format
Windows and Linux	80-bit structure with two <code>int32</code> components, <code>mh</code> and <code>ml</code> , and an <code>int16</code> component, <code>e</code>

Mac Intel

64-bit double-precision floating-point number

Complex Numbers

The complex data types are structures with two floating-point components, `re` and `im`. As with floating-point numbers, complex numbers can have 32-bit, 64-bit, and extended-precision components. The following table contains the code for the type definitions for each of these complex data types.

Complex Number Type	Code
32-bit	<pre>typedef struct { float32 re, im; } cplx64;</pre>
64-bit	<pre>typedef struct { float64 re, im; } cplx128;</pre>
Extended-precision	<pre>typedef struct { floatExtre, im; } cplxExt;</pre>

char

The `char` data type is defined by C to be an 8-bit signed integer. LabVIEW defines an unsigned `char` data type, with the following type definition:

```
typedef uInt8 uChar;
```

Dynamic

LabVIEW defines a number of data types you must allocate and deallocate dynamically. Arrays, strings, and paths have data types you must allocate using [memory manager](#) and [file manager](#) routines.

Arrays

LabVIEW supports arrays of any of the basic data types described in this section. You can construct more complicated data types using clusters, which can in turn contain scalars, arrays, and other clusters.

The first four bytes of a LabVIEW array indicate the number of elements in the array. The elements of the array follow the length field.

Strings

LabVIEW supports C- and Pascal-style strings, lists of strings, and LStr, a special string data type you use for string parameters to external code modules. The [support manager](#) contains routines for manipulating strings and converting them among the different types of strings.

C-Style Strings (CStr)

A C-style string CStr is a series of zero or more unsigned characters, terminated by a zero. C strings have no effective length limit. Most manager routines use C strings, unless you specify otherwise. The following code is the type definition for a C string:

```
typedef uChar *CStr;
```

Pascal-Style Strings (PStr)

A Pascal-style string PStr is a series of unsigned characters. The value of the first character indicates the length of the string. A PStr can have a range of 0 to 255 characters. The following code is the type definition for a Pascal string:

typedef uChar	Str255[256], Str31[32],
*StringPtr,	
**StringHandle;	
typedef uChar	*PStr;

LabVIEW Strings (LStr)

The first four bytes of a LabVIEW string LStr indicate the length of the string. The specified number of characters follow the length of the string. LStr is the string data type used by LabVIEW block diagrams. The following code is the type definition for an LStr string:

```
typedef struct {
```

```

int32 cnt;

/* number of bytes that follow */

uChar str[1];

/* cnt bytes */

} LStr, *LStrPtr, **LStrHandle;

```

Concatenated Pascal String (CPStr)

Many algorithms require manipulation of lists of strings. Arrays of strings are usually the most convenient representation for lists. However, arrays of strings can place a burden on the memory manager because of the large number of dynamic objects it must manage. To make working with lists more efficient, LabVIEW supports the concatenated Pascal string CPStr data type, which is a list of Pascal-style strings concatenated into a single block of memory. Using the CPStr data structure, you can use support manager routines to create and manipulate lists.

The following code is the type definition for a CPStr string:

```

typedef struct {

int32 cnt;

/* number of pascal strings that follow */

uChar str[1];

/* cnt concatenated pascal strings */

} CPStr, *CPStrPtr, **CPStrHandle;

```

Paths

A path (pathname) indicates the location of a file or directory in a file system. LabVIEW has a separate data type for a path, represented as Path, which the file manager defines in a platform-independent manner. The actual data type for a path is private to the file manager and subject to change. You can create and manipulate Path data types using file.

Memory-Related

LabVIEW uses pointers and handles to reference dynamically allocated memory. The memory-related data types have the following type definitions:

```

typedef uChar *UPtr;

```



```
typedef uChar **UHandle;
```

Constants

The manager data types define the following constant for use with external code modules:

```
NULL 0(uInt32)
```

The following constants define the possible values of the Bool32 data type:

```
FALSE 0 (int32)
```

```
TRUE 1 (int32)
```

The following constants define the possible values of the LVBoolean data type:

```
LVFALSE 0 (uInt8)
```

```
LVTRUE 1 (uInt8)
```

LabVIEW Manager Function Errors

If you receive errors from LabVIEW Manager functions, most error names correspond to LabVIEW error codes. The error names correspond to the following LabVIEW error codes:

Error Name	LabVIEW Error Code Number
mgArgErr	1
mFullErr	2
fEOF	4
fIsOpen	5
fIOErr	6
fNotFound	7
fNoPerm	8
fDiskFull	9
fDupPath	10
ftMFOpen	11
fNotEnabled	12

Error Name	LabVIEW Error Code Number
rfNotFound	13
rAddFailed	14
rNotFound	15
iNotFound	16
iMemoryErr	17
dPenNotExist	18
cfgBadType	19
cfgTokenNotFound	20
cfgParseError	21
cfgAllocError	22
ecLVSBFormatError	23
ecLVSBSubrError	24
ecLVSBNoCodeError	25
wNullWindow	26
wDestroyMixup	27
menuNullMenu	28
pAbortJob	29
pBadPrintRecord	30
pDriverError	31
pWindowsError	32
pMemoryError	33
pDialogError	34
pMiscError	35
dvInvalidRefnum	36
dvDeviceNotFound	37
dvParamErr	38
dvUnitErr	39
dvOpenErr	40

Error Name	LabVIEW Error Code Number
dvAbortErr	41
bogusError	42
cancelError	43
OMObjLowErr	44
OMObjHiErr	45
OMObjNotInHeapErr	46
OMOHeapNotKnownErr	47
OMBadDPIDErr	48
OMNoDPinTabErr	49
OMMsgOutOfRangeErr	50
OMMethodNullErr	51
OMUnknownMsgErr	52
mgNotSupported	53
ncBadAddressErr	54
ncInProgress	55
ncTimeOutErr	56
ncBusyErr	57
ncNotSupportedErr	58
ncNetErr	59
ncAddrInUseErr	60
ncSysOutOfMem	61
ncSysConnAbortedErr	62
ncConnRefusedErr	63
ncNotConnectedErr	64
ncAlreadyConnectedErr	65
ncConnClosedErr	66
amInitErr	67
occBadOccurrenceErr	68

Error Name	LabVIEW Error Code Number
occWaitOnUnBoundHdlrErr	69
occFunnyQ0verFlowErr	70
fDataLogTypeConflict	71
ecLVSBCannotBeCalledFromThread	72
amUnrecognizedType	73
mCorruptErr	74
ecLVSBErrorMakingTempDLL	75
ecLVSB0ldCIN	76
fmtTypeMismatch	81
fmtUnknownConversion	82
fmtTooFew	83
fmtTooMany	84
fmtScanError	85
lvOLEConvertErr	87
rtMenuErr	88
pwdTampered	89
LvVariantAttrNotFound	90
LvVariantTypeMismatch	91
axEventDataNotAvailable	92
axEventStoreNotPresent	93
axOccurrenceNotFound	94
axEventQueueNotCreated	95
axEventInfoNotAvailable	96
oleNullRefnumPassed	97
iviInvalidDowncast	102
iviInvalidClassSesn	103
ncSockNotMulticast	108
ncSockNotSinglecast	109

Error Name	LabVIEW Error Code Number
ncBadMulticastAddr	110
ncMcastSockReadOnly	111
ncMcastSockWriteOnly	112
ncDatagramMsgSzErr	113
dataCorruptErr	116
requireFullPathErr	117
folderNotExistErr	118
ncBtInvalidModeErr	119
ncBtSetModeErr	120
mgBtInvalidGUIDStrErr	121
rVersInFuture	122

Memory Manager

Most applications need routines for allocating and deallocating memory on request. The LabVIEW memory manager is the set of platform-independent memory manager functions you can use to dynamically allocate, manipulate, and deallocate memory. The memory manager supports dynamic allocation of both nonrelocatable and relocatable blocks, using pointers and handles.

If you need to perform dynamic memory allocation or manipulation from external code modules, use the memory manager. If your external code operates on data types other than scalars, you should understand how LabVIEW manages memory and know which utilities manipulate data.

The following code shows how the memory manager defines generic handle and pointer data types:

```
typedef uChar *UPtr;

typedef uChar **UHandle;
```

All data passed to or from a shared library is allocated using the memory manager. You should only use file manager functions, not the memory manager routines, to manipulate paths. Thus, your shared library should use the memory routines when working with parameters passed from the block diagram.

Memory Allocation

Applications use the following types of memory allocation:

- Static
- Dynamic



NOTE

You can allocate memory for variables in C code using `malloc`. However, do not assign the resulting pointer to any variable you pass back to the LabVIEW block diagram. Use the memory manager functions if you want to create or resize memory associated with a variable passed from the LabVIEW block diagram.

Static Memory Allocation

With static memory allocation, the compiler determines memory requirements when you create an application. When you launch the application, LabVIEW creates memory for the known global memory requirements of the application. The memory LabVIEW creates remains allocated while the application runs. Static memory allocation is simple to work with because the compiler handles all the details.

However, static memory allocation cannot address the memory management requirements of most real-world applications because you cannot determine most memory requirements until run time. Also, statically declared memory might result in larger memory requirements because the memory is allocated for the duration of the application.

Dynamic Memory Allocation

With dynamic memory allocation, you reserve memory when you need it and free memory when you are no longer using it. Dynamic allocation requires more work than static memory allocation because you have to determine memory requirements and allocate and deallocate memory as necessary.

You can use the LabVIEW memory manager to dynamically allocate memory in the following ways:

- Using pointers to allocate memory
- Using handles to allocate memory

Using Pointers for Dynamic Memory Allocation

The more conventional method uses pointers to allocate memory. With pointers, you request a block of memory of a certain size. The routine returns the address of the block of memory to your shared library. When you no longer need the block of memory, you call a routine to free the block of memory. You can use the block of memory to store data. You reference the data stored in the block of memory by using the address the manager routine

returned when you created the pointer. You can make copies of the pointer and use them in multiple places in your application to refer to the same data.

Pointers in the LabVIEW memory manager are nonrelocatable, which means the manager never moves the memory block to which a pointer refers while that memory is allocated for a pointer. Because other references to the memory block do not become out of date, not moving the memory block allocated to a pointer avoids problems that occur when you need to change the amount of memory allocated to a pointer. If you need more memory, sufficient memory might not exist to expand the memory space of the pointer without moving the memory block to a new location. If an application had multiple references to the pointer, moving the memory block to a new location causes problems because each pointer refers to the old memory address of the data. Using invalid pointers can cause severe problems.

Using Handles for Dynamic Memory Allocation

A second form of memory allocation uses handles. As with pointers, when you allocate memory using handles, you request a block of memory of a certain size. The memory manager allocates the memory and adds the address of the memory block to a list of master pointers. The memory manager returns a handle that is a pointer to the master pointer. If you reallocate a handle and it moves to another address, the memory manager updates the master pointer to refer to the new address. If you look up the correct address using the handle, you access the correct data.

Use handles to perform most memory allocation in LabVIEW. Pointers are available because in some cases they are more convenient and simpler to use.

Using Pointers and Handles

Create a handle using `DSNewHandle`, which allows you to specify the size of the memory block, or `DSNewAlignedHandle`, which allows you to specify the size, alignment, and alignment offset of the memory block. Create a pointer using `DSNewPtr`. Create a handle or pointer and set it to all zeros using `DSNewHClr`, `DSNewAlignedHClr`, and `DSNewPClr`.

When you are finished with the handle or pointer, release it using `DSDisposeHandle` or `DSDisposePtr`.

If you need to resize an existing handle, use the `DSSetHandleSize` function, which determines the size of an existing handle. If you need to resize and realign an existing handle, use the `DSSetAlignedHandleSize` function, which determines the size, alignment, and alignment offset of an existing handle. You also can resize and realign an existing handle and set it to all zeros using `DSSetAlignedHSzClr`. Because pointers are not relocatable, you cannot resize them.

A handle is a pointer to a pointer. In other words, a handle is the address of an address. The second pointer, or address, is a master pointer, which means it is maintained by the memory manager. Languages that support pointers provide operators for accessing data by its address. With a handle, you use this operator twice—once to get to the master pointer, and a second time to get to the actual data.

Additional routines make it easy to copy and concatenate handles and pointers to other handles, check the validity of handles and pointers, and copy or move blocks of memory from one place to another.

Simple Example of Using Pointers and Handles

This simple example demonstrates how to work with pointers and handles in C.

The following code shows how to work with a pointer to an `int32`:

```
int32 *myInt32P;

myInt32P = (int32 *)DSNewPtr(sizeof(int32));

*myInt32P = 5;

x = *myInt32P + 7;

...

DSDisposePtr(myInt32P);
```

The first line declares the variable `myInt32P` as a pointer to, or the address of, a 32-bit signed integer. The first line does not actually allocate memory for the `int32`. The first line creates memory for an address and associates the name `myInt32P` with that address. The P at the end of the variable name is a convention used in this example to indicate the variable is a pointer.

The second line creates a block of memory in the data space large enough to hold a single 32-bit signed integer and sets `myInt32P` to refer to this memory block.

The third line places the value 5 in the memory location to which `myInt32P` refers. The `*` operator refers to the value in the address location.

The fourth line sets `x` equal to the value at address `myInt32P` plus 7.

The last line frees the pointer.

The following code is the same example using handles instead of pointers:

```
int32 **myInt32H;

myInt32H =(int32**)DSNewHandle(sizeof(int32));

**myInt32H = 5;

x = **myInt32H + 7;
```


...

```
DSDisposeHandle(myInt32H);
```

The first line declares the variable `myInt32H` as a handle to a 32-bit signed integer. Strictly speaking, the first line declares `myInt32H` as a pointer to a pointer to an `int32`. As with the previous example, the first line does not allocate memory for the `int32`. The first line creates memory for an address and associates the name `myInt32H` with that address. The H at the end of the variable name is a convention used in this example to indicate the variable is a handle.

The second line creates a block of memory in the data space large enough to hold a single `int32`. `DSNewHandle` places the address of the memory block as an entry in the master pointer list and returns the address of the master pointer entry. Finally, the second line sets `myInt32H` to refer to the master pointer.

The third line places the value 5 in the memory location to which `myInt32H` refers. Because `myInt32H` is a handle, you use the `*` operator twice to get to the data.

The fourth line sets `x` equal to the value referenced by `myInt32H` plus 7.

The last line frees the handle.

This example shows only the simplest aspects of how to work with pointers and handles in C. Other examples show different aspects of using pointers and handles. Refer to a C manual for a list of other operators you can use with pointers and for more information about how to work with pointers.

File Manager

The file manager supports routines for opening and creating files, reading data from and writing data to files, and closing files. In addition, you can manipulate the end-of-file mark of a file and position the current read or write mark to an arbitrary position in the file. You also can move, copy, and rename files, determine and set file characteristics, and delete files.

The file manager contains a number of routines for directories, with which you can create and delete directories. You also can determine and set directory characteristics and obtain a list of a directory's contents.

LabVIEW supports concurrent access to the same file, so you can have a file open for both reading and writing simultaneously. When you open a file, you can indicate whether you want the file to be read from and written to concurrently. You also can lock a range of the file, if you need to make sure a range is nonvolatile at a given time.

The file manager also provides many routines for manipulating paths, or path names, in a platform-independent manner. The file manager supports the creation of path descriptions, which are either relative to a specific location or absolute, that is, the full path. With file manager routines you can create and compare paths, determine

characteristics of paths, and convert a path between platform-specific descriptions and the platform-independent form.

Applications that manipulate files can use the functions in the file manager. The file manager routines support basic file operations such as creating, opening, and closing files, writing data to files, and reading data from files. In addition, you can use file manager routines to create directories, determine characteristics of files and directories, and copy files. File manager routines use a LabVIEW data type for file path names, called `Path s`, that indicates a file or directory path independent of the platform. You can translate a `Path` to and from the conventional format a host platform uses for describing a file pathname.

Identifying Files and Directories

When you perform operations on files and directories, you need to identify the target of the operation. The platforms LabVIEW supports use a hierarchical file system, meaning files are stored in directories, possibly nested several levels deep. These hierarchical file systems support the connection of multiple discrete storage media, called volumes. For example, DOS-based systems support multiple drives connected to the system. For most of these hierarchical file systems, you must include the volume name to specify the location of a file. On other systems, such as Linux, you do not need to specify the volume name because the physical implementation of the file system is hidden from the user.

How you identify a target depends upon whether the target is an open or closed file. If a target is a closed file or a directory, specify the target using the path of the target. The path describes the volume containing the target, the directories between the top level and the target, and the name of the target. If the target is an open file, use a file descriptor to specify that LabVIEW should perform an operation on the open file. The file descriptor is an identifier the file manager associates with the file when you open it. When you close the file, the file manager dissociates the file descriptor from the file.

Path Specifications

LabVIEW uses the following types of file path specifications:

- Conventional
- Empty
- LabVIEW specifications

Conventional

All platforms have a method for describing the paths for files and directories. These path specifications are similar, but they are usually incompatible from one platform to another. You usually specify a path as a series of names separated by separator characters. Typically, the first name is the top level of the hierarchical specification of the path. The last name is the file or directory the path identifies.

A path can be one of the following types:

- Relative path
- Absolute path

A relative path describes the location of a file or directory relative to an arbitrary location in the file system. An absolute path describes the location of a file or directory starting from the top level of the file system.

A path does not necessarily go from the top of the hierarchy down to the target. You can often use a platform-specific tag in place of a name that indicates the path should go up a level from the current location.

(OS X 64-bit and Linux) You specify the path of a file or directory as a series of names separated by the slash (/) character. If the path is an absolute path, you begin the specification with a slash. Indicate the path should move up a level using two periods in a row (..). Thus, the following path specifies a file `readme` relative to the top level of the file system, where `username` is the user directory:

```
/usr/home/ username /myapps/readme
```

The following paths are two relative paths to the same file:

<code>username /myapps/readme</code>	relative to <code>/usr/home</code>
<code>../myapps/readme</code>	relative to a directory inside of the <code>username</code> directory

(Windows) You separate names in a path with a backslash (\) character. If the path is an absolute path, you begin the specification with a drive designation, followed by a colon (:), followed by the backslash. Indicate the path should move up a level using two periods in a row (..). Thus, the following path specifies a file `readme` relative to the top level of the file system, on a drive named `C`:

```
c:\home\ username \myapps\readme
```

The following paths are two relative paths to the same file:

<code>username \myapps\readme</code>	relative to the home directory
<code>..\myapps\readme</code>	relative to a directory inside of the <code>username</code> directory

(OS X 32-bit) You separate names in a path with the colon (:) character. If the path is an absolute path, you begin the specification with the name of the volume containing the file. If an absolute path consists of only one name, that is, it specifies a volume, the path must end with a colon. If the path is a relative path, it begins with a colon. The colon is optional for a relative path consisting of only one name. Indicate the path should move up a level using two colons in a row (::). Thus, the following path specifies a file `readme` relative to the top level of the file system, on a drive named `hard drive`:

```
hard drive:home: username :myapps:readme
```

The following paths are two relative paths to the same file:

: username : myapps : readme	relative to the home directory
: : myapps : readme	relative to a directory inside of the username directory

Empty

You can define a path with no names, called an empty path. An empty path is either absolute or relative. An absolute path contains the complete address of a file's location within the file hierarchy. An empty absolute path refers to the highest point in the file hierarchy. A relative path contains the address of a file's location relative to the position of the user within the file hierarchy. An empty relative path refers to the current position of the user within the file hierarchy.

(OS X 64-bit and Linux) A slash (/) represents the empty absolute path. The slash specifies the root of the file hierarchy. A period (.) represents the empty relative path.

(Windows) You represent the empty absolute path as an empty string. The empty absolute path specifies the set of all volumes on the system. A period (.) represents the empty relative path.

(OS X 32-bit) The empty absolute path is represented as an empty string. The absolute path specifies the set of all volumes on the system. A colon (:) represents the empty relative path.

LabVIEW

In LabVIEW, you specify a path using a special LabVIEW data type, represented as `Path`. The exact structure of the `Path` data type is private to the file manager. You create and manipulate the `Path` data type using file manager routines.

A `Path` is a dynamic data structure. Just as you use memory manager routines to allocate and deallocate handles and pointers, you use file manager routines to create and deallocate a `Path`. Just as with handles, declaring a `Path` variable does not actually create a `Path`. Before you can use the `Path` to manipulate a file, you must dynamically allocate the `Path` using file manager routines. When you are finished using the `Path` variable, you should release the `Path` using file manager routines.

In addition to providing routines for the creation and elimination of a `Path`, the file manager provides routines for comparing, duplicating, determining `Path` characteristics, and converting `Path` to and from other formats, such as the platform-specific format for the system on which LabVIEW is running.

File Descriptors

When you open a file, LabVIEW returns a file descriptor associated with the file. A file descriptor is a data type LabVIEW uses to identify open files. All operations performed on an open file use the file descriptor to identify the file.

A file descriptor is valid only while the file is open. If you close the file, the file descriptor is no longer associated with the file. If you open the file again, the new file descriptor is most likely different from the previous file descriptor.

File Refnums

In the file manager, LabVIEW accesses open files using file descriptors. However, on the front panel and block diagram, LabVIEW accesses open files using file refnums. A file refnum contains a file descriptor for use by the file manager and additional information used by LabVIEW.

LabVIEW specifies file refnums using the LVRefNum data type, the exact structure of which is private to the file manager. To pass references to open files into a shared library, convert file refnums to file descriptors with the FRefNumToFD LabVIEW Manager function. To pass references to open files out of a shared library, convert file descriptors to file refnums with the FNewRefNum LabVIEW Manager function.

Support Manager

The support manager contains a collection of constants, macros, basic data types, and functions, such for bit or byte manipulation of data, string manipulation, mathematical operations, sorting, searching, and determining the current time and date.

The string functions contain much of the functionality of the string libraries supplied with standard C compilers, such as string concatenation and formatting. You can use variations of many of these functions with LabVIEW strings, Pascal strings, and C strings. The following table describes the different string types.

String Type	Description
LabVIEW	4-byte length field followed by data, generally stored in a handle
Pascal	1-byte length field followed by data
C	data terminated by a null character

With the utility functions, you can sort and search on arbitrary data types, using quicksort and binary search algorithms. You also can convert numeric error codes to the associated text description.

The support manager also contains transcendental functions for many complex and extended floating-point operations.

Certain routines specify time as a data structure. The following example code illustrates specifying time as a data structure:

```
typedef struct {
    int32sec; /* 0:59 */
    int32min; /* 0:59 */
    int32hour; /* 0:23 */
    int32mday; /* day of the month, 1:31 */
    int32mon; /* month of the year, 1:12 */
    int32year; /* year, 1600:3000 */
    int32yday; /* day of year (julian date), 1:366 */
    int32isdst; /* 1 if daylight savings time */
} DateRec;
```

Signal Processing and Analysis

June 2012, 371365H-01

Use this book to learn about signal processing and analysis concepts.

© 2005–2012 National Instruments. All rights reserved.

Signal Generation in LabVIEW

The Signal Generation VIs create many common signals required for network analysis and simulation. You also can use the Signal Generation VIs in conjunction with National Instruments hardware to generate analog output signals.

The names of most of the Signal Generation VIs contain the word *wave* or *pattern*. These names indicate whether a VI can keep track of the phase of the signal it generates each time the VI is called.



NOTE

You can use the Signal Generation VIs only in the LabVIEW Full and Professional Development Systems.

When you use a VI that requires the normalized frequency as an input, you must convert the frequency units to the normalized units of cycles per sample. You must use normalized units of cycles per sample with the following Signal Generation VIs :

- Sine Wave VI
- Square Wave VI
- Sawtooth Wave VI
- Triangle Wave VI
- Arbitrary Wave VI
- Chirp Pattern VI

IIR Filter Characteristics in LabVIEW

IIR filters in LabVIEW have the following characteristics:

- The IIR Filter VI interprets values at negative indexes in the following impulse equations as zero the first time you call the VI.

$$x_0 = 1$$

$$x_i = 0$$
 for all $i \neq 0$.
- A transient response, or delay, proportional to the filter order occurs before the filter reaches a steady state.
- The number of elements in the filtered sequence equals the number of elements in the input sequence.
- The filter retains the internal filter state values when the filtering process finishes.

Curve Fitting in LabVIEW

Some Fitting VIs return only the coefficients for the curve that best describe the input data while other Fitting VIs return the fitted curve. Using the VIs that return only coefficients allows you to further manipulate the data. The VIs that return the fitted curve also return the coefficients and the mean squared error (MSE). MSE is a relative measure of the residuals between the expected curve values and the actual observed values. Because the input data represents a discrete system, the VIs use the following equation to calculate MSE:

$$\text{MSE} = \frac{1}{n} \sum_{i=0}^{n-1} (f_i - y_i)^2$$

where f is the sequence representing the fitted values, y is the sequence representing the observed values, and n is the number of observed sample points.

The Fitting VIs automatically set up and solve the Jacobian system and return the set of coefficients that best describes the data set.

Related information Fitting VIs

Spline Interpolation

Spline interpolation avoids the disadvantages of polynomial interpolation because functions require more data points and the line interpolant is similar to the original function.

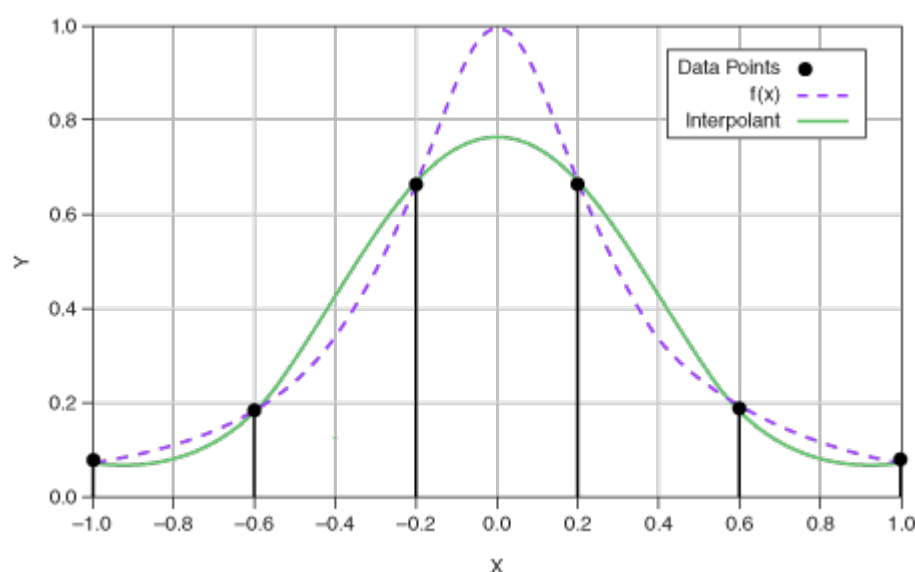
Cubic spline interpolation is a common type of spline interpolation that uses a third-order polynomial for each interval between two adjacent points. Third-order polynomials must meet the following conditions:

- The first and second derivatives at interior points are continuous.
- The polynomials pass through all data points.
- For one-dimensional interpolation, the second derivatives for the beginning and end points are zero. For spline interpolant, the initial boundary and final boundary inputs specify the first derivatives at the beginning and end points. For spline interpolation, a line with the first derivatives behaves the same as the spline interpolant. For spline interpolation, a line with a natural spline behaves the same as one-dimensional interpolation.

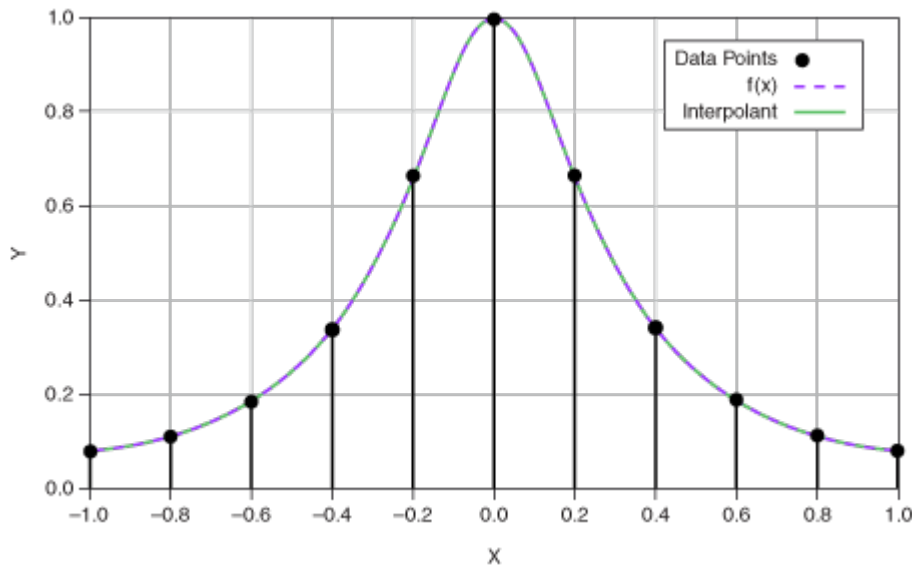
For example, the equation

$$y = f(x) = \frac{1}{1+12x^2}, \quad x \in [-1, 1]$$

with natural cubic spline interpolation is shown in the following figure.



When you have more data points of $f(x)$, the interpolant of the cubic spline interpolation is similar to $f(x)$, as shown in the following figure.



Related information

Interpolate 1D VI

Spline Interpolant VI

Spline Interpolation VI

Basic Linear Algebra Subroutines

Introduction to Basic Linear Algebra Subroutines

Basic Linear Algebra Subroutines (BLAS) are routines that provide standard functions for basic vector and matrix operations. You can classify these routines in the following three levels:

- BLAS level 1, which calculates vector operations.
- BLAS level 2, which calculates matrix-vector operations.
- BLAS level 3, which calculates matrix-matrix operations.

For example, `dgemm` is a BLAS level 3 function that calculates the general real matrix-matrix product using the following expression:

$$\alpha * op(A) * op(B) + \beta * op(C)$$

where α and β are scalars, A, B, and C are matrices, and opA is A or A^T .

Refer to BLAS (Basic Linear Algebra Subprograms) for more information on the BLAS functions.

Related information [BLAS \(Basic Linear Algebra Subprograms\)](#)

Choosing Between BLAS VIs and Linear Algebra VIs

While LabVIEW provides basic Linear Algebra VIs, you can use the Basic Linear Algebra Subroutines VIs for advanced linear algebra algorithms. Use the Basic Linear Algebra Subroutines VIs to combine multiple linear algebra operations, reduce data copying in LabVIEW, and optimize execution. Therefore, if you have limited memory, use the Basic Linear Algebra Subroutines VIs.



NOTE

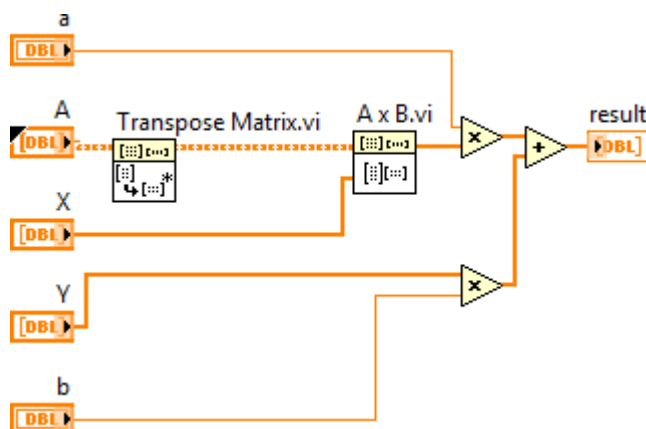
Note You can use the Basic Linear Algebra Subroutines VIs only in the LabVIEW Full and Professional Development Systems.

For example, the following expression combines linear algebra functions:

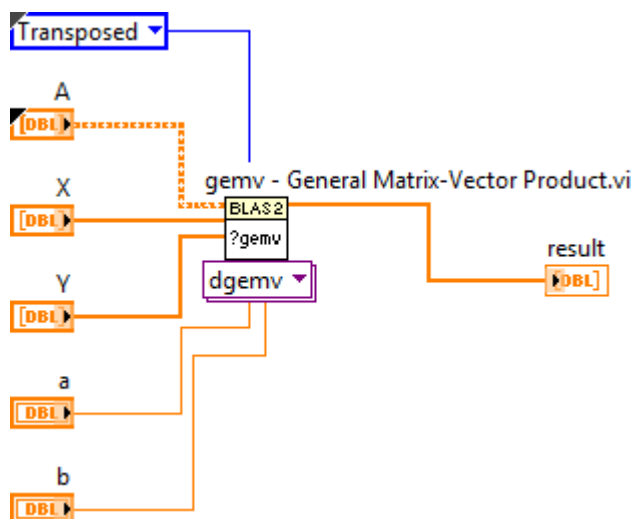
$$\alpha A^T X + bY$$

where α is scalar, A is a matrix, and X and Y are vectors.

With linear algebra VIs, you need five different VIs and functions to perform this calculation, as shown in the following block diagram.



To simplify the code on a block diagram, you can replace multiple VIs and functions with the Basic Linear Algebra Subroutines VIs. For the current example, the `gemv` - General Matrix-Vector Product VI simplifies the code, as shown in the following block diagram.



Related information Linear Algebra VIs Basic Linear Algebra Subroutines VIs [gemv - General Matrix-Vector Product VI](#)

Solving ODEs in LabVIEW

Use the ODE Solver VI to specify the ODE solver to use to evaluate an ordinary differential equation. You can use the ODE Solver VI only in the LabVIEW Full and Professional Development Systems.

LabVIEW includes the following ODE solvers:

- **Runge-Kutta 1 (Euler)** —A fixed step-size, single-step explicit Runge-Kutta ODE solver of first order.
- **Runge-Kutta 2** —A fixed step-size, single-step explicit Runge-Kutta ODE solver of second order.
- **Runge-Kutta 3** —A fixed step-size, single-step explicit Runge-Kutta ODE solver of third order.
- **Runge-Kutta 4** —A fixed step-size, single-step explicit Runge-Kutta ODE solver of fourth order.
- **Runge-Kutta 23** —A variable step-size, single-step explicit Runge-Kutta ODE solver of third order.
- **Runge-Kutta 45** —A variable step-size, single-step explicit Runge-Kutta ODE solver of fifth order, which uses the Dormand-Prince coefficients.
- **BDF** —A variable step-size, variable order (orders 1 through 5) implementation of the multi-step backwards difference formula (BDF), also known as Gear's Method. This method is adequate for moderately stiff problems.
- **Adams-Moulton** —A variable step-size, multi-step variable order (orders 1 through 12) implementation of the Adams-Moulton predictor-corrector pair in predict-evaluate-correct-evaluate (PECE) mode.
- **Rosenbrock** —A variable step-size, single-step explicit solver. This method is adequate for some stiff problems.

Solving PDEs in LabVIEW

Use the Partial Differential Equations VIs to solve partial differential equations (PDE), including the Helmholtz, heat, and wave equations. The following table provides details about these equations.



NOTE

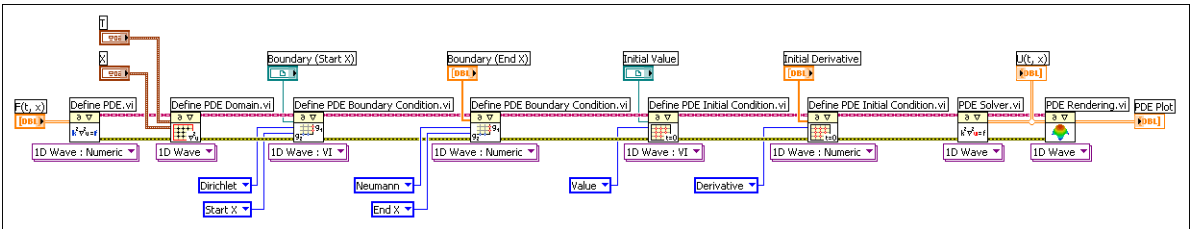
The Partial Differential Equations VIs are available only in the LabVIEW Full and Professional Development Systems.

Type of Equation	Dimension and Domain	Boundary Condition	Method
Helmholtz equation	The equation is two-dimensional and defined in a rectangular or polygonal domain.	The polygonal domain supports only the Dirichlet condition. The rectangular domain supports the Dirichlet and Neumann conditions.	The finite element method solves the equation defined on the polygonal domain. The finite difference method solves the equation defined on the rectangular domain.
Heat equation	The equation is one-dimensional or two-dimensional and defined on the rectangular domain.	The equation supports the Dirichlet and Neumann conditions.	The finite difference method solves this equation.
Wave equation	The equation is one-dimensional or two-dimensional and defined on the rectangular domain.	The equation supports the Dirichlet and Neumann conditions.	The finite difference method solves this equation.

The following block diagram examples show a common order for solving PDEs with the VIs. While you can add the VIs to the block diagram in the order they appear in the two examples, you do not need to follow that order when you use the PDE VIs. The following list provides an example of an order you can use when you want to solve PDEs with the VIs.

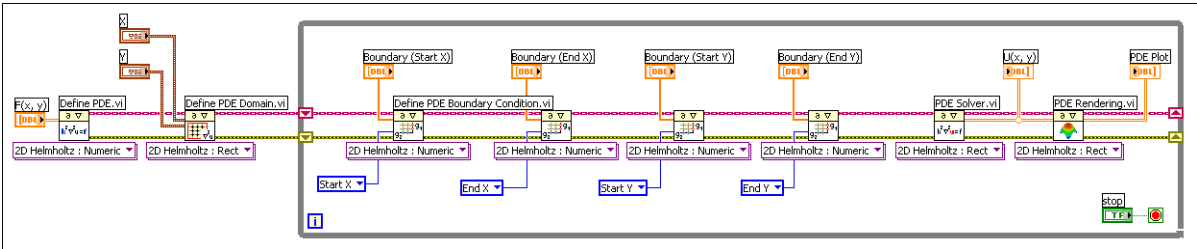
- Define PDE
- Define PDE Domain
- Define PDE Boundary Condition
- Define PDE Initial Condition
- PDE Solver
- PDE Rendering

The following block diagram illustrates an example of how to solve a 1D wave equation.



NOTE
The polymorphic instance you choose must match the type of equation you need to solve. The type of instance also needs to be consistent from step to step. Otherwise, LabVIEW returns a type conflict error.

The following block diagram illustrates an example of how to solve a Helmholtz equation in a loop. The coefficient, the right side of the equation, and the domain of the equation do not change during the execution. While you do not need to repeatedly define the equation and domain in the loop, the boundary values vary over time. Therefore, you only need to place the definition of the boundary condition in the loop. In the following block diagram example, a shift register of a While Loop implements the PDE class to transfer data from one iteration to the next.



Related information Partial Differential Equations VIs Define PDE VI Define PDE Domain VI Define PDE Boundary Condition VI Define PDE Initial Condition VI PDE Solver VI PDE Rendering VI

Point-By-Point Analysis

Point-by-Point Analysis and Array-Based Analysis in

The approach used for most point-by-point analysis operations in LabVIEW remains the same as array-based analysis. You can use filters, integration, mean value algorithms, and so on, in the same situations and for the same reasons that you can use these operations in array-based data analysis. In contrast, the computation of zeros in polynomial functions is not relevant to point-by-point analysis, and point-by-point versions of these array-based VIs are not necessary.

The following table compares array-based LabVIEW analysis to point-by-point analysis.

Data Analysis Technology	
Array-Based Analysis (traditional paradigm) • Prepare a buffer unit of data. • Analyze data. • Produce a buffer of analyzed data. • Generate report.	Point-By-Point Analysis (newer paradigm) • Receive continuous stream of data. • Filter and analyze data continuously. • Generate real-time events and reports continuously.

The following table presents other comparisons between array-based and point-by-point analysis.

Characteristic	Array-Based Analysis	Data Acquisition and Analysis with Point By Point VIs
Compatibility	Limited compatibility with real-time systems	Compatible with real-time systems; backward compatible with array-based systems
Data typing	Array-oriented	Scalar-oriented
Interruptions	Interruptions critical	Interruptions tolerated
Operation	You observe, offline	You control, online
Performance and programming	Compensate for startup data loss (4-5 seconds) with complex “state machines”	Startup data loss does not occur; initialize the data acquisition system once and run continuously
Point of view	Reflection of a process, like a mirror	Direct, natural flow of a process
Programming	Specify a buffer	No explicit buffers
Results	Output a report	Output a report and an event in real time
Run-time behavior	Delayed processing	Real time
Run-time behavior	Stop	Continue
Run-time behavior	Wait	Now
Work style	Asynchronous	Synchronous

When you perform point-by-point analysis, keep in mind the following concepts:

- **Initialization** —You must initialize the point-by-point analysis application to prevent interference from settings you made in previous sessions of data analysis.
- **Reentrant Execution** —You must enable LabVIEW reentrant execution for point-by-point analysis. Reentrant execution allocates fixed memory to a single analysis

process, guaranteeing that two processes that use the same analysis function never interfere with each other.

**NOTE**

If you create custom VIs to use in a point-by-point application, be sure to enable reentrant execution. Reentrant execution is enabled by default in almost all Point By Point VIs.

- **Deterministic Performance** —Point-by-point analysis is the natural companion to many deterministic systems, because it efficiently integrates with the flow of a real-time data signal.

Related information [Reentrant Execution](#)

Performing Analysis without Buffers of Data

Analysis functions yield solutions that characterize the behavior of a data set. In array-based data acquisition and analysis, you might analyze a large set of data by dividing the data into 10 smaller buffers. Analyzing those 10 sets of data yields 10 solutions. You can further resolve those 10 solutions into one solution that characterizes the behavior of the entire data set.

In point-by-point analysis, you analyze an entire data set in real-time. A sample unit of a specific length replaces a buffer. The point-by-point sample unit can have a length that matches the length of a significant event in the data set that you are analyzing. For example, the application in a case study acquires a few thousand samples per second to detect defective train wheels. The input data for the train wheel application comes from the signal generated by a train that is moving at 60 km to 70 km per hour. The sample length corresponds to the minimum distance between wheels.

A typical point-by-point analysis application analyzes a long series of sample units, but you are likely to have interest in only a few of those sample units. To identify those crucial samples of interest, the point-by-point application focuses on transitions, such as the end of the relevant signal.

The train wheel detection application in the case study uses the end of a signal to identify crucial samples of interest. The instant the application identifies the transition point, it captures the maximum amplitude reading of the current sample unit. This particular amplitude reading corresponds to the complete signal for the wheel on the train whose signal has just ended. You can use this real-time amplitude reading to generate an event or a report about that wheel and that train.

Related concepts [Case Study of Point-By-Point Analysis](#)

Case Study of Point-by-Point Analysis

Case Study of Point-By-Point Analysis Application Built in LabVIEW

This case study shows a complete point-by-point analysis application built in LabVIEW with Point By Point VIs.



NOTE

You can use the Point By Point VIs only in the LabVIEW Full and Professional Development Systems.

Refer to the Train Wheel PtByPt VI in the `labview\examples\ptbypt\PtByPt_No_HW.llb` for an example of using the Point By Point VIs.

The Train Wheel PtByPt VI is a real-time data acquisition application that detects defective train wheels and demonstrates the simplicity and flexibility of point-by-point data analysis.

When acquiring data with real-time performance, point-by-point analysis helps you analyze data in real time. Point-by-point analysis occurs continuously and instantaneously. While you acquire data, you filter and analyze it, point by point, to extract the information you need and to make an appropriate response. This case study demonstrates the effectiveness of the point-by-point approach for generation of both events and reports in real time.

Related information Point By Point VIs

Point-By-Point Analysis of Train Wheels

In this example, the maintenance staff of a train yard must detect defective wheels on a train. The current method of detection consists of a railroad worker striking a wheel with a hammer and listening for a different resonance that identifies a flaw. Automated surveillance must replace manual testing, because manual surveillance is too slow, too prone to error, and too crude to detect subtle defects. An automated solution also adds the power of dynamic testing, because the train wheels can be in service during the test, instead of standing still.

The automated solution to detect potentially defective train wheels needs to have the following characteristics:

- Detect even subtle signs of defects quickly and accurately.
- Gather data when a train travels during a normal trip.
- Collect and analyze data in real time to simplify programming and to increase speed and accuracy of results.

The Train Wheel PtByPt VI offers a solution for detecting defective train wheels.

**NOTE**

This example focuses on implementing a point-by-point analysis program in LabVIEW. The issues of ideal sampling periods and approaches to signal conditioning are beyond the scope of this example.

Overview of the LabVIEW Point-By-Point Solution

As well as Point By Point VIs, the Train Wheel PtByPt VI requires standard LabVIEW programming objects, such as Case structures, While Loops, numeric controls, and numeric operators.

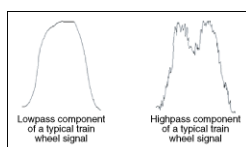
The data the Train Wheel PtByPt VI acquires flows continuously through a While Loop. The process carried out by the Train Wheel PtByPt VI inside the While Loop consists of five analysis stages that occur sequentially. The following list reflects the order in which the five analysis stages occur, briefly describes what occurs in each stage, and corresponds to the labeled portions of the previous block diagram.

1. In the data acquisition stage (DAQ), waveform data flows into the While Loop.
2. In the Filter stage, separation of low- and high-frequency components of the waveform occurs.
3. In the Analysis stage, detection of the train, wheel, and energy level of the waveform for each wheel occurs.
4. In the Events stage, responses to signal transitions of trains and wheels occurs.
5. In the Report stage, the logging of trains, wheels, and trains that might have defective wheels occurs.

Related information While Loop Case Structure

Characteristics of a Train Wheel Waveform

The characteristic waveform that train wheels emit determines how you analyze and filter the waveform signal point-by-point. A train wheel in motion emits a signal that contains low- and high-frequency components. If you mount a strain gauge in a railroad track, you detect a noisy signal similar to a bell curve. The following illustration shows the low- and high-frequency components of this curve.



The low-frequency component of train wheel movement represents the normal noise of operation. Defective and normal wheels generate the same low-frequency component in the signal. The peak of the curve represents the moment when the wheel moves directly

above the strain gauge. The lowest points of the bell curve represent the beginning and end of the wheel, respectively, as the wheel passes over the strain gauge.

The signal for a train wheel also contains a high-frequency component that reflects the quality of the wheel. In operation, a defective train wheel generates more energy than a normal train wheel. In other words, the high-frequency component for a defective wheel has greater amplitude.

Analysis Stages of the Train Wheel

Five Analysis Stages of the Train Wheel

The waveform of all train wheels, including defective ones, falls within predictable ranges. This predictable behavior allows you to choose the appropriate analysis parameters. These parameters apply to five analysis stages.



NOTE

You must adjust parameters for any implementation of the Train Wheel PtByPt VI because the characteristics of each data acquisition system differ.

DAQ Stage of the Train Wheel PtByPt VI

Data moves into the Point By Point VIs through the input data parameter. The point-by-point detection application operates on the continuous stream of waveform data that comes from the wheels of a moving train. For a train moving at 60 km to 70 km per hour, a few hundred to a few thousand samples per second are likely to give you sufficient information to detect a defective wheel.

Filter Stage of the Train Wheel PtByPt VI

The Train Wheel PtByPt VI must filter low- and high-frequency components of the train wheel waveform. Two Butterworth Filter PtByPt VIs perform the following tasks:

- Extract the low-frequency components of the waveform.
- Extract the high-frequency components of the waveform.

In the Train Wheel PtByPt VI, the Butterworth Filter PtByPt VIs use the following parameters:

- `order` specifies the amount of the waveform data that the VI filters at a given time and is the filter resolution. 2 is acceptable for the Train Wheel PtByPt VI.
- `fl` specifies the low cut-off frequency, which is the minimum signal strength that identifies the departure of a train wheel from the strain gauge. 0.01 is acceptable for the Train Wheel PtByPt VI.

- fh specifies the high cut-off frequency, which is the minimum signal strength that identifies the end of high-frequency waveform information. 0.25 is acceptable for the Train Wheel PtByPt VI.

Related information Butterworth Filter PtByPt VI

Analysis Stage of the Train Wheel PtByPt VI

The point-by-point detection application must analyze the low- and high-frequency components separately. The Array Max & Min PtByPt VI extracts waveform data that reveals the level of energy in the waveform for each wheel, the end of each train, and the end of each wheel.

Three separate Array Max & Min PtByPt VIs perform the following discrete tasks:

- Identify the maximum high-frequency value for each wheel.
- Identify the end of each train.
- Identify the end of each wheel.



NOTE

The name Array Max & Min PtByPt VI contains the word array only to match the name of the array-based form of this VI. You do not need to allocate arrays for the Array Max & Min PtByPt VI.

In the Train Wheel PtByPt VI, the Array Max & Min PtByPt VIs use the following parameters and functions:

- sample length specifies the size of the portion of the waveform that the Train Wheel PtByPt VI analyzes. To calculate the ideal sample length, consider the speed of the train, the minimum distance between wheels, and the number of samples you receive per second. 100 is acceptable for the Train Wheel PtByPt VI. The Train Wheel PtByPt VI uses sample length to calculate values for all three Array Max & Min PtByPt VIs.
- The Multiply function sets a longer portion of the waveform to analyze. When this longer portion fails to display signal activity for train wheels, the Array Max & Min PtByPt VIs identify the end of the train. 4 is acceptable for the Train Wheel PtByPt VI.
- threshold provides a comparison point to identify when no train wheel signals exist in the signal that you are acquiring. threshold is wired to the Greater? function. 3 is an acceptable setting for threshold in the Train Wheel PtByPt VI.

Related information Multiply Function Greater? Function Array Max and Min PtByPt VI

Events Stage of the Train Wheel PtByPt VI

After the Analysis stage identifies maximum and minimum values, the Events stage detects when these values cross a threshold setting.

The Train Wheel PtByPt VI logs every wheel and every train that it detects. Two Boolean Crossing PtByPt VIs perform the following tasks:

- Generate an event each time the Array Max & Min PtByPt VIs detect the transition point in the signal that indicates the end of a wheel.
- Generate an event every time the Array Max & Min PtByPt VIs detect the transition point in the signal that indicates the end of a train.

The Boolean Crossing PtByPt VIs respond to transitions. When the amplitude of a single wheel waveform falls below the threshold setting, the end of the wheel has arrived at the strain gauge. For the Train Wheel PtByPt VI, 3 is a good threshold setting to identify the end of a wheel. When the signal strength falls below the threshold setting, the Boolean Crossing PtByPt VIs recognize a transition event and pass that event to a report.

Analysis of the high-frequency signal identifies which wheels, if any, might be defective. When the Train Wheel PtByPt VI encounters a potentially defective wheel, the VI passes the information directly to the report at the moment the end-of-wheel event is detected.

In the Train Wheel PtByPt VI, the Boolean Crossing PtByPt VIs use the following parameters:

- initialize resets the VI for a new session of continuous data acquisition.
- direction specifies the kind of Boolean crossing.

Related information Boolean Crossing PtByPt VI

Report Stage of the Train Wheel PtByPt VI

The Train Wheel PtByPt VI reports on all wheels for all trains that pass through the data acquisition system. The Train Wheel PtByPt VI also reports any potentially defective wheels.

Every time a wheel passes the strain gauge, the Train Wheel PtByPt VI captures its waveform, analyzes it, and reports the event. The following table describes the components of a report on a single train wheel.

Information Source	Meaning of Results
Counter mechanism for waveform events	Stage One: Wheel number four has passed the strain gauge.
Analysis of highpass filter data	Stage Two: Wheel number four has passed the strain gauge and the wheel might be defective.

Information Source	Meaning of Results
Counter mechanism for end-of-train events	Stage Three: Wheel number four in train number eight has passed the strain gauge, and the wheel might be defective.

The Train Wheel PtByPt VI uses point-by-point analysis to generate a report, not to control an industrial process. However, the Train Wheel PtByPt VI acquires data in real time, and you can modify the application to generate real-time control responses, such as stopping the train when the Train Wheel PtByPt VI encounters a potentially defective wheel.

Accessing Scanned I/O Data

This section provides information on using and accessing I/O variables in LabVIEW.

Using I/O Variables (Real-Time, Windows)

An I/O variable is a type of *shared variable* that transfers I/O data. When accessed locally, I/O variables use the *NI Scan Engine* for single-point access to I/O data. When accessed remotely, I/O variables use the *NI Publish-Subscribe Protocol (NI-PSP)* for single-point or buffered access to I/O data. An I/O variable can communicate directly with a physical I/O channel or can transfer custom I/O data between an FPGA VI and a real-time (RT) VI running on the same target.



CAUTION

The NI Scan Engine does not buffer data. To avoid losing data when accessing I/O variables locally, you must read each data point before the next point arrives and avoid writing data faster than the *NI Scan Engine period*.

The following table summarizes the use cases for I/O variables:

Use	I/O Variable Type
Transferring scaled I/O data between the I/O hardware and a LabVIEW VI without using the LabVIEW FPGA Module	I/O Variables for Scaled I/O Data
Transferring custom FPGA-processed I/O data between a LabVIEW FPGA VI and a LabVIEW Real-Time VI running on the same hardware device	I/O Variables for Custom FPGA I/O Data



NOTE

Refer to the specific hardware device documentation for information about NI Scan Engine support.

I/O Variables for Scaled I/O Data

To use I/O variables for scaled I/O data, you first must connect a supported I/O device to a target with the NI Scan Engine installed and then **add** the I/O device to the target in LabVIEW project.

When you add a supported I/O device to a supported target in the Project Explorer window, LabVIEW automatically adds an I/O variable for each I/O channel. If you physically add an I/O device to a target included in the current LabVIEW project, you must add the new I/O device to the project and deploy the corresponding I/O variables before you can use them.

If you physically remove an I/O device from a target that is part of the current LabVIEW project, you must delete the I/O device from the project and redeploy the target that contained the I/O device. Redeploying the target ensures that the configuration on the target reflects the project configuration. You can use the Project & System Comparison dialog box to ensure that the project and target configurations match.

I/O Variables for Custom FPGA I/O Data (FPGA Module or Real-Time Module)

If the project includes an FPGA target that supports the NI Scan Engine, you can create user-defined I/O variables to transfer custom I/O data between FPGA VIs and RT VIs running on the same hardware device.

You can add user-defined I/O variables to the block diagrams of FPGA VIs and RT VIs running on the chassis that hosts the user-defined I/O variables. However, because all I/O variables are unidirectional, you must configure the direction of each user-defined I/O variable as either **FPGA to Host** or **Host to FPGA**. For example, you can acquire analog I/O data and perform an FFT on the data in an FPGA VI, use an **FPGA to Host** I/O variable to transfer the processed data to a control loop in an RT VI, then use a **Host to FPGA** I/O variable to transfer output data from the RT control loop back to the FPGA for output to the physical I/O channel.

You must install the LabVIEW FPGA Module to create user-defined I/O variables. However, if you have only the Real-Time Module, you can access user-defined I/O variables from RT VIs even if you do not install the LabVIEW FPGA Module. For example, a developer with the LabVIEW FPGA Module could create user-defined I/O variables and compile an FPGA bitfile that accesses those variables. Then that developer could distribute the bitfile and the LabVIEW project containing the user-defined I/O variables to another developer without the FPGA Module, who could use the project to run the bitfile on a CompactRIO chassis. In this case, the second developer could not create new user-defined I/O variables but could use the Real-Time Module to access the user-defined I/O variables created by the first developer.

(Windows) If network publishing is enabled on a user-defined I/O variable, you can use the variable in any RT VI or Windows-based VI within the same LabVIEW project. For example, you can use network-published I/O variables to create a user interface VI that runs on Windows.



NOTE

User-defined I/O variables support scanned access only. You cannot use direct access for user-defined I/O variables.

Configuring I/O Variables

You can configure I/O variables interactively at edit-time or run-time or programmatically at run-time. You also can use a combination of these methods. The following table summarizes the advantages and disadvantages of each method:

Method	Use Case	Advantages	Disadvantages
Interactive	Configuring settings for I/O variables when you have access to the Project Explorer window.	Requires less programming.	• Configuration settings are not visible when you view the block diagram. • You can configure settings for an I/O variable only on the target that hosts the variable.

Method	Use Case	Advantages	Disadvantages
Programmatic	- Configuring settings for I/O variables when you do not have access to the Project Explorer window. For example, if you plan to create a stand-alone application that uses an I/O variable and you need to change configuration settings for the I/O variable while running the application, configure the I/O variable programmatically. - Configuring settings for an I/O variable from a computer other than the target that hosts the I/O variable. For example, if you need to configure an I/O variable on a CompactRIO target from a host computer, configure the I/O variable programmatically.	- Configuration settings are visible when you view the block diagram. - You can configure I/O variables remotely.	Requires more programming.

Configuring I/O Variables Interactively

Use the Shared Variable Properties dialog box to configure a single I/O variable or the Multiple Variable Editor window to configure multiple I/O variables simultaneously.

You can update the description, network, and scaling options of an I/O variable even while running a VI that uses the I/O variable. However, you cannot update the name or data type of an I/O variable at run-time.



NOTE

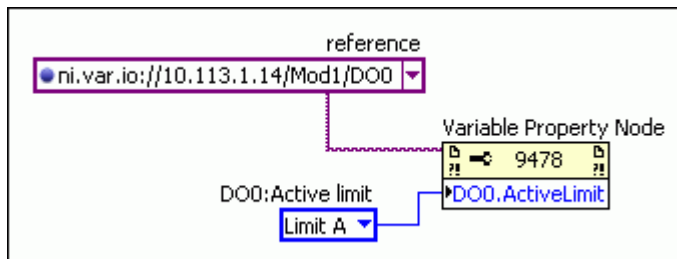
When you add a new I/O variable or change the configuration of an I/O variable, you must deploy the variable to activate the new configuration.

Configuring I/O Variables Programmatically

Complete the following steps to use the Variable properties to configure an I/O variable programmatically:

1. Add the Variable Property Node to the block diagram.
2. Right-click the **reference** input and select **Create»Constant** from the shortcut menu.
3. Click the arrow button on the right side of the shared variable refnum constant and select **Browse**.
4. Use the Browse Variable Object dialog box to select the I/O variable you want to configure.
5. Right-click the shared variable refnum constant and use the **Select Variable Class** shortcut menu to select the class of the property you want to use.
6. Click the white area of the Variable Property Node and select the property you want to use.
7. (Optional) Use the Positioning tool to *resize* the Variable Property Node to display more than one property simultaneously.

The following figure shows an example of programmatically configuring an I/O variable on a remote target:



In this example, the Variable Property Node specifies the current limit for a channel on a CompactRIO module by configuring the I/O variable that corresponds to that channel.



NOTE

If the Variable Property Node cannot connect to the remote I/O variable after five seconds, the configuration call times out. Complete the following steps to change this timeout period:

1. Open the `labview\LabVIEW.ini` file as a text file.
2. On a new line, enter `RemoteIOVConfigurationTimeout=` followed by the new timeout period in number of seconds. For example, if you want to extend the timeout period to 10 seconds, enter `RemoteIOVConfigurationTimeout=10`.
3. Save and close the file.

Deploying I/O Variables

To deploy I/O variables, right-click the project item that contains the I/O variables in the Project Explorer window and select **Deploy** from the shortcut menu.

When you run a VI that contains I/O variable nodes, LabVIEW automatically deploys the I/O variable containers to which those I/O variables belong. This action deploys all of the I/O variables in the containers.



NOTE

If an I/O driver fails to initialize, LabVIEW aborts loading all I/O variables in the project. In this case, the I/O variables might still appear in the project, but if an application attempts to access the variables, a run-time error occurs. Refer to the specific target I/O driver documentation for information about debugging I/O driver errors.

Scaling I/O Variable Values

You can enable linear scaling on an I/O variable or alias on the Scaling page of the Shared Variable Properties dialog box. However, I/O variables do not support nonlinear scaling.



NOTE

Some I/O buses implement hardware scaling. In this case, LabVIEW I/O variable scaling provides an additional scaling layer. LabVIEW does not overwrite the hardware scaling. LabVIEW uses the hardware-scaled value from the I/O bus as the raw value when calculating the software-scaled value.

Creating I/O Aliases

You can create aliases of an I/O variable to name and scale the I/O data differently in different contexts. For example, if you have an I/O variable connected to a thermocouple input, you can create a Celsius alias and a Fahrenheit alias. Then you could scale each alias and use the aliases to display the temperature in both units of measure.

The value of an I/O alias is linked bidirectionally to the value of the parent, so updating the value of the parent updates the value of the I/O alias, and updating the value of the I/O alias updates the value of the parent. By extension, updating any link in a chain of I/O aliases updates all other links in the chain.

Creating a Single I/O Alias

To create an I/O alias, right-click a target that contains I/O variables in the Project Explorer window and select **New»Variable** from the shortcut menu to display the Shared Variable Properties. Enter a name for the I/O alias and select **I/O Alias** from the **Variable Type** pull-

down menu. Then click the **Browse** button and select the existing I/O variable or I/O alias to which you want to bind the new I/O alias.

**NOTE**

When you bind an alias to an I/O variable, ensure that the data type of the alias matches the data type of the underlying I/O variable.

You can use the Project Explorer window to edit the name, description, network-publishing settings, and scaling of an I/O alias. You also can use the Multiple Variable Editor window to edit multiple I/O aliases simultaneously.

**NOTE**

You cannot deploy a library that contains both I/O aliases and other types of shared variables. You must deploy I/O aliases in a [separate library](#).

Creating Multiple I/O Aliases Simultaneously

You can use the Batch Variable Creation dialog box to create multiple I/O aliases simultaneously. Complete the following steps to batch create I/O aliases:

1. Create a single I/O alias to use as a template for the batch creation process.
2. Right-click the variable library that contains the I/O alias and select **Create Variables** from the shortcut menu to display the Batch Variable Creation dialog box.
3. Select **Copy properties from**, click the **Browse** button, and select the existing I/O alias that you want to use as a template.
4. Enter the number of I/O aliases you want to create in the **Number to create** field and click the **OK** button to create the new I/O aliases.

After you complete these steps, LabVIEW automatically opens the Multiple Variable Editor window, which you can use to edit the new variables. For example, you might want to edit the **Alias Path** of each I/O alias to bind to a unique I/O variable.

Forcing I/O Variable Values

You can force the value of an I/O variable either interactively or programmatically. Forcing an I/O variable causes the associated I/O data to assume the value you specify until you unforce the variable, reboot the target, or force the variable to assume a different value. Unforcing an I/O variable returns control of the I/O value to the NI Scan Engine.

**NOTE**

When an I/O variable is forced, each I/O variable access takes slightly longer than when the variable is not forced, which could cause a loop to run late if the loop period is not long enough to accommodate the forcing overhead.

Forcing I/O Variable Values Interactively

Use the NI Distributed System Manager to **force** or unforce the value of an I/O variable during debugging or to manually control an I/O channel. From LabVIEW, select **Tools»Distributed System Manager** to launch the NI Distributed System Manager. You also can launch the Distributed System Manager by right-clicking an RT target in the Project Explorer window and selecting **Utilities»View in System Manager**.



NOTE

You can use the NI Distributed Manager security settings to prevent users from forcing I/O variables from within the NI Distributed System Manager.

Forcing I/O Variable Values Programmatically

Use the Forcing VIs to force and unforce I/O variables programmatically.

Forcing I/O Aliases

Forcing applies to aliases as well as standard I/O variables. When you force an I/O variable, you also force all associated aliases. When you force an alias, you also force the parent I/O variable and all other associated aliases.



NOTE

LabVIEW applies scaling to forced values as if the forced value were the actual value of the I/O channel. So, when you force one link in a chain of aliases, all links in the chain scale appropriately.

Forcing and Unforcing all I/O Variables on a Target Simultaneously

You can use the global forcing state of a target to force and unforce all I/O variables on the target simultaneously. You can enable and disable global forcing on a target either interactively or programmatically.

Use the **Enable Forcing** and **Disable Forcing** buttons in the NI Distributed System Manager to enable and disable global forcing on a target interactively. Use the Enable Variable Forcing and Disable Variable Forcing VIs to enable and disable global forcing on a target programmatically.

Complete the following steps to force all the I/O variables on a target simultaneously:

1. Disable forcing on the target so that you can set the desired forcing values of the I/O variables individually without individually forcing the I/O variables.
2. Set the desired forcing values for all the I/O variables on the target.
3. Enable forcing on the target to force all I/O variables on the target simultaneously.

Accessing I/O Variables Remotely

If network-publishing is enabled on an I/O variable, you can access the I/O variable from a remote computer connected to the same network as the target that hosts the I/O variable. You can access remote I/O variables interactively using the *I/O Variable Probe* view in the NI Distributed System Manager. You also can access remote I/O variables programmatically *in a VI*.

Use the Shared Variable Properties dialog box to enable and disable network publishing on an I/O variable. Enable network-publishing if you need to monitor I/O values on a host computer or access an I/O variable from a remote target. Use the Scan Engine page to set the global network-publishing rate for all I/O variables on a target.



NOTE

If you do not plan to access I/O variables remotely, disable network publishing to minimize CPU overhead. When you disable network publishing on an I/O variable or I/O alias, you cannot access the variable from a VI running on another computer. However, you still can force the variable from the NI Distributed System Manager. To prevent users from forcing an I/O variable, use the System Manager security settings.

Accessing I/O Variables Locally

LabVIEW adds I/O variables to a global scan engine memory map and updates the values of all I/O variables concurrently. However, for I/O variable access on the local target that hosts the I/O variable, you can configure each node to use either scanned access or direct access.

Complete the following steps to select the local access mode of an I/O variable node:

- 1. Right-click an I/O variable node on the block diagram.
- 2. Select **Local Access Mode** from the shortcut menu
- 3. Select the access mode you want to use for the node.



NOTE

LabVIEW adds all I/O variables to the global scan engine memory map regardless of the access mode you use for each variable.

Selecting a Local Access Mode

In general, scanned access is appropriate for groups of I/O channels with similar update rates, and direct access is appropriate for individual I/O channels that update asynchronously to the scan period. The following table summarizes when to use each I/O access method:

Common Use	Access Method
Single-point access to I/O channels that share a common update rate	Scanned

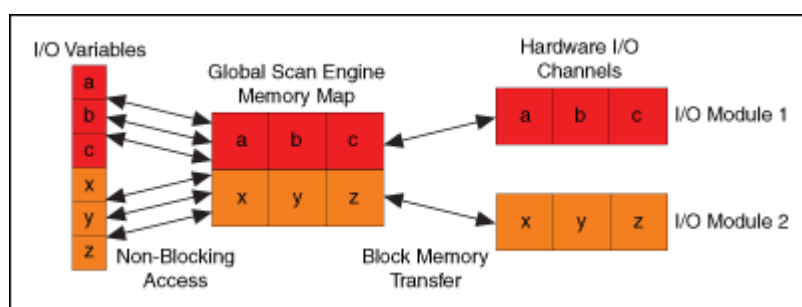
Common Use	Access Method
Single-point access to local I/O channels asynchronous to the NI Scan Engine period, such as a single write for an emergency stop output (not supported for user-defined I/O variables)	Direct

**NOTE**

You can use direct access for I/O channels that update slower than the NI Scan Engine period. However, in such cases, scanned access typically provides the same I/O behavior more efficiently than direct access.

Scanned Access

By default, LabVIEW configures I/O variable nodes to use scanned access. Use scanned access for sets of I/O channels that update at a single rate and for expansion I/O channels. Scanned access uses the scan engine memory map to perform non-blocking I/O reads and writes, as shown in the following figure:



Each time you read from an I/O variable using scanned access, the NI Scan Engine immediately returns the most recent value stored in the memory map without blocking, or waiting for a new value. During each scan, LabVIEW reads the most recent I/O value and writes that value to the NI Scan Engine memory map.

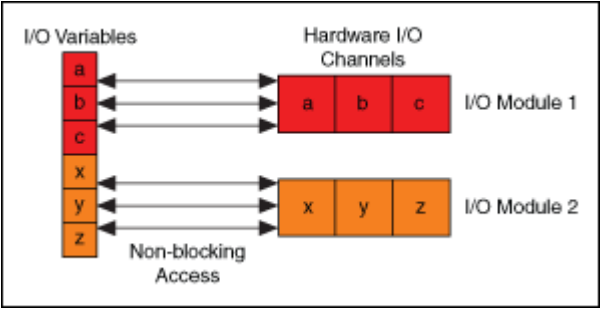
Each time you write to an I/O variable using scanned access, you overwrite the previous value stored in the scan engine memory map. During each scan, LabVIEW pushes the value stored in the memory map to the physical I/O channel. To prevent data loss, you must **synchronize** all I/O variable write operations to the scan period.

Direct Access

Use direct access to read or write single-point local I/O channels asynchronously to the scan period. You can use direct access when the application needs to write a value as soon as possible. Direct access bypasses the scan engine memory map and communicates directly with the I/O device driver to perform non-blocking I/O reads and writes, as shown in the following figure:

**NOTE**

User-defined I/O variables do not support direct access.



NOTE

If you use the NI Scan Engine in conjunction with the RIO Scan Interface, the hardware scan signal governs updates to I/O. If you write a value via direct access under the idle state of the hardware scan signal, the value propagates to the hardware after the idle state of the hardware scan signal ends.



NOTE

The speed of direct access varies by controller, I/O module, and communication protocol. Refer to the specific hardware documentation for more information about I/O access speed.

Interpreting I/O Variable Node Icons

Each I/O variable node on the block diagram includes a glyph to indicate one of the following access methods:

Access Method	Node Appearance
NI-PSP	
Scanned	
Direct	



[Data Communication Methods Home](#)

Accessing Scanned I/O Data (Real-Time, Windows)

Use the NI Scan Engine for single-point access to scanned I/O data.

Using the NI Scan Engine

The NI Scan Engine enables efficient *single-point* access to sets of data channels, such as I/O channels, using a scan that stores data in a global memory map and updates all values at a single rate, known as the scan period. The NI Scan Engine installs on LabVIEW for

Windows by default. You also can install the NI Scan Engine on supported LabVIEW targets, such as RT Series PXI and CompactRIO targets. Refer to the specific hardware device documentation for information about NI Scan Engine support.

(Real-Time Module) By default, the NI Scan Engine runs in a thread above time-critical priority on RT targets, although LabVIEW includes the scan thread in the time-critical category when reporting CPU usage statistics. When running the NI Scan Engine at above time-critical priority, you must synchronize the deterministic sections of the application with the scan period to ensure that the scan thread does not affect the determinism of the application. For applications in which I/O is not the highest-priority task, you also can configure the priority of the scan engine to fall between time-critical and Timed Structure priority. If you do not plan to use I/O variables on a target, do not install the NI Scan Engine on the target. If the NI Scan Engine is already installed on the target, you can use Measurement & Automation Explorer (MAX) to uninstall the NI Scan Engine.



NOTE

(Windows) The only NI Scan Engine priority available on Windows is normal priority. Because of the non-deterministic nature of the Windows operating system, the timing of the NI Scan Engine is not guaranteed when running on Windows.

Configuring NI Scan Engine Settings

Use the Scan Engine page to configure scan engine settings including the scan period, network-publishing rate, and priority level of the NI Scan Engine.

Use the NI Distributed System Manager to monitor and manage scan engine *faults* and *modes*. Select **Tools»Distributed System Manager** to display the NI Distributed System Manager. You also can use the NI Scan Engine VIs to view and configure scan engine settings programmatically.

Deploying and Running the NI Scan Engine

After you configure NI Scan Engine settings in the Project Explorer window, you must deploy a project item that uses the NI Scan Engine, such as My Computer or an RT target, for the settings to take effect.

(Windows) The NI Scan Engine starts running the first time you deploy the target and stops running when you close the project. To automatically deploy the NI Scan Engine and I/O variables when a built application runs, you must enable the **Include target hardware configuration** checkbox on the Advanced page of the Application Properties dialog box before building the application.

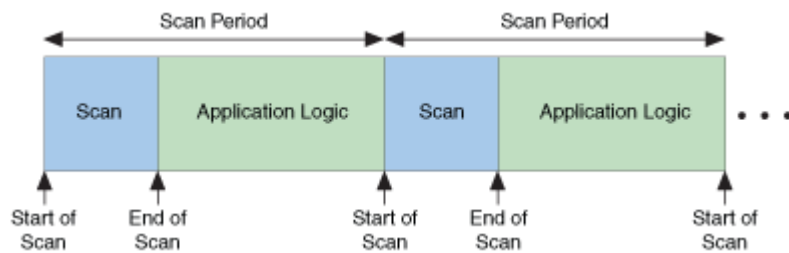
(Real-Time Module) When running on an RT target, the NI Scan Engine starts automatically when you boot the target and runs continuously until you shut down the target.

A single computer can run only a single instance of the NI Scan Engine at a time. For example, if you create a Windows built application that uses the NI Scan Engine, you must close the project from which you built the application before you can run the application on the same computer. Although you can configure multiple instances of the NI Scan Engine in multiple LabVIEW projects, deployment of subsequent projects using the NI Scan

Engine fails if an instance of the NI Scan Engine is running. To run a different instance of the NI Scan Engine, you first must stop the current instance.

Scan Engine Timing

The NI Scan Engine executes at regular intervals determined by the **Scan Period** you specify on the Scan Engine page. Choose a period long enough to accommodate both the scan itself and the application logic, as shown in the following illustration:



NOTE

The length of the scan depends on the number and type of I/O items deployed to the target. To maximize scan engine performance, undeploy any I/O items that you do not plan to use in the application.

Use the Get Scan Engine Period VI to read the scan period programmatically. Use the Set Scan Engine Period VI to set the scan period programmatically.



NOTE

(Real-Time Module) Real-time loops generally need one or two warm-up iterations to begin executing deterministically. Before checking to ensure that an application meets timing requirements, you should allow each time-critical loop to execute warm-up iterations.

Synchronizing to the Scan Engine

(Real-Time Module) Unless you configure the NI Scan Engine to run below time-critical priority, you must synchronize time-critical code to the scan period to avoid collisions that could affect the determinism of the application.

Use the **Synchronize to Scan Engine** timing source to synchronize timed structure execution to the scan engine. If you do not want to use a timed structure, you can use the Synchronize to Scan Engine VI to synchronize to the scan engine. Both synchronization methods trigger execution at the time labeled **End of Scan** in the previous illustration. To use I/O variables as a coherent data set, ensure that the synchronized code finishes executing before the next scan iteration. However, you can skip scan iterations if the code does not depend on a coherent data set.



NOTE

If synchronized code does not finish executing before the next scan iteration, the information reported by the error cluster of an I/O variable might lose synchronization with the I/O value.

**NOTE**

When you click the **Abort** button on a VI that involves synchronization to the NI Scan Engine, the VI does not abort until the current scan iteration completes, so the VI might appear to hang temporarily if the scan period is sufficiently long.

NI Scan Engine Modes

**NOTE**

Only certain targets and devices use NI Scan Engine modes. CompactRIO targets with local I/O modules do not use NI Scan Engine modes. Refer to the I/O hardware documentation for information about hardware-specific mode behavior.

LabVIEW defines four NI Scan Engine modes:

0	Initialization Mode	Occurs only briefly during startup.
1	Configuration Mode	The required mode when configuring scan engine settings on hardware that uses NI Scan Engine modes.
2	Active Mode	The mode in which the scan engine runs and updates values.
3	Fault Mode	The mode triggered when a major or unrecoverable <i>fault</i> occurs.

**NOTE**

Fault mode behavior varies by target. Refer to the specific target hardware documentation for information about fault mode behavior.

Use the NI Distributed System Manager to [view and configure](#) the scan engine mode. The NI Distributed System Manager displays the value of the scan engine mode using the EngineState status item. Use the Get Scan Engine Mode VI to read the scan engine mode programmatically. Use the Set Scan Engine Mode VI to set the scan engine mode programmatically.



[Data Communication Methods Home](#)

NI Scan Engine Faults (Real-Time, Windows)

Targets with the NI Scan Engine installed use faults to address asynchronous error conditions. LabVIEW defines three fault levels: minor, major, and unrecoverable. LabVIEW logs all faults in memory on the target. Use the [Scan Engine Probe](#) view in the NI Distributed System Manager to [view and clear](#) NI Scan Engine faults on a target.

Minor faults are equivalent to warnings and do not trigger fault handling behavior. Examples of minor faults include startup errors, which can occur when the controller is unable to apply its saved configuration on startup.

Major faults are equivalent to errors. For example, if LabVIEW detects that the scan engine has run late, LabVIEW triggers major fault –66460. If LabVIEW detects ten consecutive late scan iterations, LabVIEW triggers major fault –66461 and the NI Scan Engine stops running.



NOTE
Some I/O hardware drivers implement fault handling behavior in response to major faults. Refer to the specific hardware documentation for information about fault handling behavior.

Unrecoverable faults can occur due to a hardware failure or software crash. In the event of an unrecoverable fault, reboot the controller and contact National Instruments.

Viewing and Clearing Faults

Use the NI Distributed System Manager to [view and clear faults](#). You also can use the Get Fault List VI and the Clear Fault VI to view and clear faults programmatically. Clearing a fault removes the fault from memory as if the fault had never occurred.

System Faults

LabVIEW defines a set of common faults. You can log additional faults based on LabVIEW error clusters. LabVIEW groups fault codes by type, as shown in the following table:



NOTE
Refer to the specific hardware documentation for information about fault handling behavior.

Fault Type	Range of Fault Codes
I/O Scan Driver Errors	–66000 through –66099
I/O Variables	–66200 through –66299
NI Scan Engine VIs	–66300 through –66399
NI Scan Engine	–66400 through –66499
User Fault	Any LabVIEW error code

Triggering Faults

You can use the Set Fault VI to trigger minor or major faults based on LabVIEW error codes. To trigger a custom fault, you can [define a custom error code](#) and then pass that code to the Set Fault VI.

LabVIEW reserves a memory pool large enough to log up to 100 unique fault codes. If you exceed 100 unique fault codes, LabVIEW triggers minor fault –66420 and stops logging additional faults. However, even if the maximum number of user faults has been reached, a

major or unrecoverable fault still can trigger hardware drivers to initiate fault handling behavior.

Configuring Faults

You can customize the level and triggering behavior of specific faults with custom fault configurations. You can apply custom fault configurations either interactively or programmatically.

Use the **Fault Configuration** section of the Scan Engine page to apply custom fault configurations interactively. Use the Fault Configuration VIs to apply custom fault configurations programmatically.

Some faults cannot be configured. However, the list of configurable faults depends on which NI Scan Engine I/O software is installed. Use the **Configurable Faults** list in the **Fault Configuration** section of the Scan Engine page to view the list of configurable faults interactively.

Reading and Writing I/O Variable Containers as Arrays (Real-Time, Windows)

By accessing the channels of an I/O variable container as a single array, you can optimize performance and streamline the block diagram. Use the Shared Variable functions to [read and write](#) local I/O variable containers as arrays.

Specifying I/O Variable Containers

An I/O variable container is an item in a LabVIEW project that contains [I/O variables](#). Use one of the following methods to specify an I/O variable container to read or write on the block diagram:

- Use a variable refnum control or constant to specify the I/O variable container using one of the following methods:
 - By [finding](#) the I/O variable container programmatically
 - By building the [URL string](#) programmatically on the block diagram
 - By typing the URL of the container into the constant
 - By using the Browse Variable Object dialog box to find the container interactively
- Find the I/O variable container [programmatically](#).

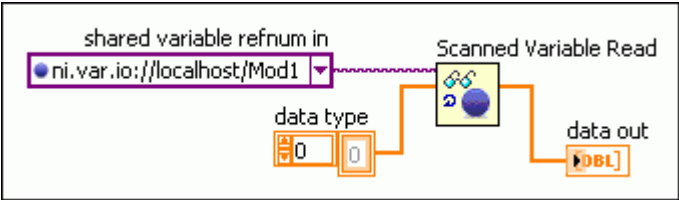
Reading and Writing I/O Variable Containers

You can wire a reference to an I/O variable container to the **shared variable refnum in** input of one of the following functions:

- Read Variable
- Write Variable
- Direct Variable Read
- Direct Variable Write
- Scanned Variable Read
- Scanned Variable Write

When you use one of the read functions in the previous list to read an I/O variable container, the function returns an array that contains the data for each channel in the I/O container. The order of the channels in the array corresponds to the order of the channels that the All Variables property returns. When you use one of the write functions in the previous list to write an I/O variable container, you must wire an array indexed in this same order.

For example, the following block diagram uses the Scanned Variable Read function to read all the I/O channels in the analog input module **Mod1** as an array of doubles:



Unsupported Features

The techniques described in this topic do not support the following features:

Unsupported Feature	Details
Reading and writing I/O variable containers that include multiple data types	You can specify only one data type for an I/O variable container. Therefore, reading or writing an I/O variable container with channels of multiple data types results in an error.
Reading and writing PSP variable containers	This technique applies only to local I/O variable containers. You cannot use this technique to access I/O variable containers remotely or to access network-published shared variable libraries.
Reading timestamps	You cannot enable timestamps when using this technique.



PID and Fuzzy Logic

Use PID and Fuzzy Logic VIs for Proportional-Integral-Derivative (PID) and fuzzy logic control. These VIs appear under the Control & Simulation palette. You can use the PID VIs with input/output (I/O) functions such as data acquisition (DAQ) to implement control of physical processes.

PID Control

The Proportional-Integral-Derivative (PID) algorithm is the most common control algorithm used in industry. A PID controller is a feedback controller you can use to control processes such as heating and cooling systems, fluid level monitoring, flow control, pressure control, and so on. You can use the PID VIs with National Instruments hardware such as a DAQ device, FieldPoint I/O module, or GPIB board, to connect your PC to a system and develop LabVIEW control applications. You also can use the PID VIs with LabVIEW math and logic functions to create block diagrams for real control strategies.

Fuzzy Logic

Fuzzy logic is a method of rule-based decision making for expert systems and process control. Fuzzy logic differs from traditional Boolean logic in that fuzzy logic allows for partial membership in a set. You can use fuzzy logic to control processes represented by subjective, linguistic descriptions. You can wire acquired data to a fuzzy controller to implement real-time decision making or control of a physical system. You also can use outputs of the fuzzy controller with DAQ analog output hardware to implement real-time process control.

Use the Fuzzy System Designer to design a fuzzy system interactively. Use the Fuzzy Logic VIs to design and control fuzzy systems programmatically.



RELATED INFORMATION

[PID Control](#) on page 1747

[Fuzzy Logic](#) on page 1739

Tutorial: Using the Fuzzy System Designer

You can use the PID and Fuzzy Logic VIs to design a [fuzzy system](#) in one of two ways: using the Fuzzy System Designer or using the Fuzzy Logic VIs. This tutorial teaches you to design a fuzzy system interactively using the **Fuzzy System Designer**.



TIP

Refer to the [Modifying a Fuzzy System with the Fuzzy Logic VIs](#) tutorial to learn how to design and modify fuzzy systems using the Fuzzy Logic VIs.

In this tutorial, you use the Fuzzy System Designer to design a fuzzy system similar to the vehicle maneuvering example described in the topics on [fuzzy system design](#). As a reference, you can find the complete fuzzy system you design in this tutorial, `Truck - backward.fs`, in the `labview\examples\control\fuzzy\Car Parking` directory.

Next [1. Creating Input Linguistic Variables](#)



Creating Input Linguistic Variables

The first step in designing a fuzzy system with the Fuzzy System Designer is to create the input and output [linguistic variables](#) for the system. In the vehicle maneuvering example described in the topics on [fuzzy system design](#), the fuzzy system has two linguistic input variables, Vehicle Position x and Vehicle Orientation β , and one linguistic output variable, Steering Angle ϕ .



TIP

The graphs in the [Creating Membership Functions](#) topic illustrate the range and membership functions for each of these linguistic variables.

Complete the following steps to create an input linguistic variable that corresponds to the Vehicle Position x input linguistic variable of the vehicle maneuvering example.

1. Select **Tools»Control Design and Simulation»Fuzzy System Designer** to launch the Fuzzy System Designer and display the **Variables** page.
2. Click the **Add Input Variable** button to the right of the **Input variables** list to display the Edit Variable dialog box.
3. In the **Name** text box, enter `vehicle-position`.
4. Specify the **Range** by entering 0 in the **minimum** field and 10 in the **maximum** field. This specifies that the position of the vehicle relative to the destination ranges from 0.0 to 10.0 meters.
5. Click the **Add Membership Function** button under the **Membership functions** control to enable the control and create a new membership function for the vehicle-position linguistic variable.
6. In the **Name** text box, enter `left`.
7. Select **Trapezoid** from the **Shape** pull-down menu to specify that a trapezoid function determines the degrees of membership for the linguistic variable.
8. Click the **Color** field to select the color to use for the membership function in the **Membership functions graph**.
9. In the **Points** array, enter the following numbers, one for each element of the array: 0, 0, 1, 4. These points specify the values of the linguistic variable corresponding to the base and top points, in order from left to right and base to top, of the membership function.
10. Repeat steps 5 through 9 for each of the following membership functions for Vehicle Position x , using the shape and points shown in the following table:

Membership Function	Shape	Points
Triangle	3, 4, 5	
Triangle	4.5, 5, 5.5	
Triangle	5, 6, 7	
Trapezoid	6, 9, 10, 10	

**NOTE**

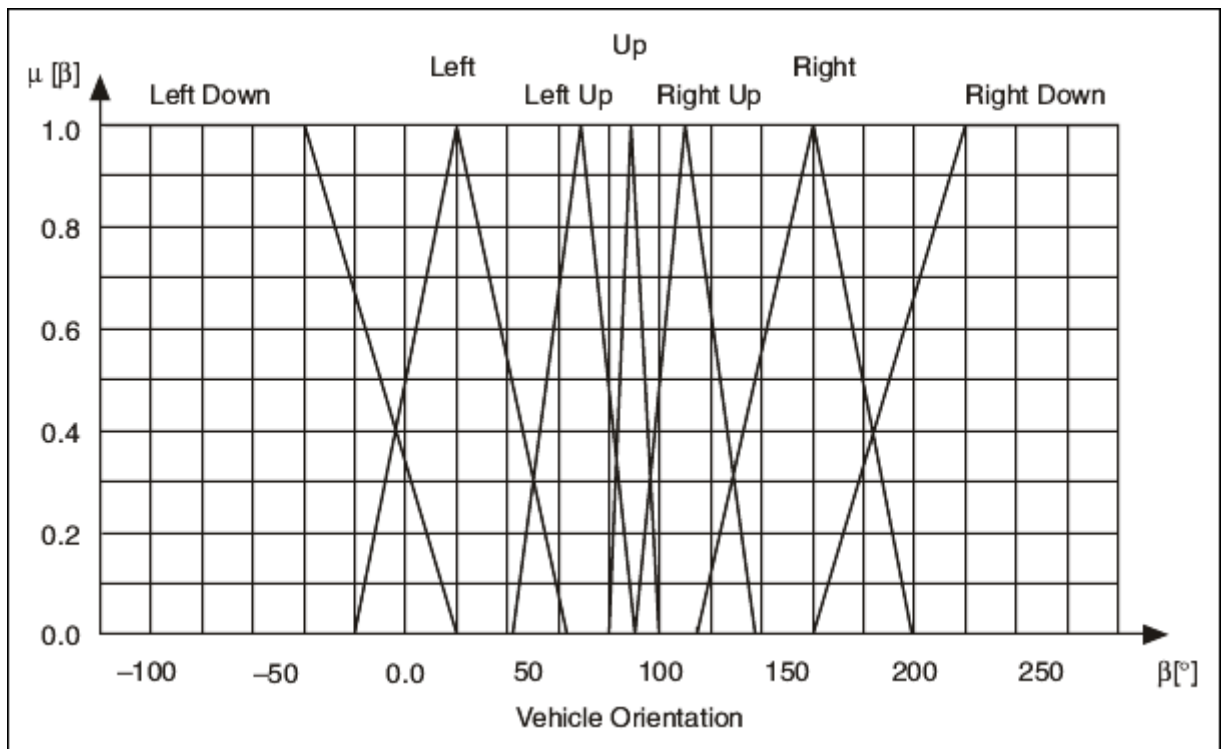
The **Points** array requires that each point for a given membership function be greater than or equal to all previous points. If the array automatically changes your entries to fit this requirement, try entering points from left to right.

You can compare the **Membership functions graph** to the [Vehicle Position](#) graph in the Creating Membership Functions topic to confirm your entries.

11. Click the **OK** button to save your changes and return to the **Variables** page. The vehicle-position input variable appears in the **Input variables** list and the corresponding membership functions appear in the **Input variable membership functions** graph.
12. Repeat steps 2 through 11 to create a vehicle-orientation input linguistic variable that corresponds to the Vehicle Orientation β variable of the vehicle maneuvering example. The following graph shows the range and membership functions for the variable.

**NOTE**

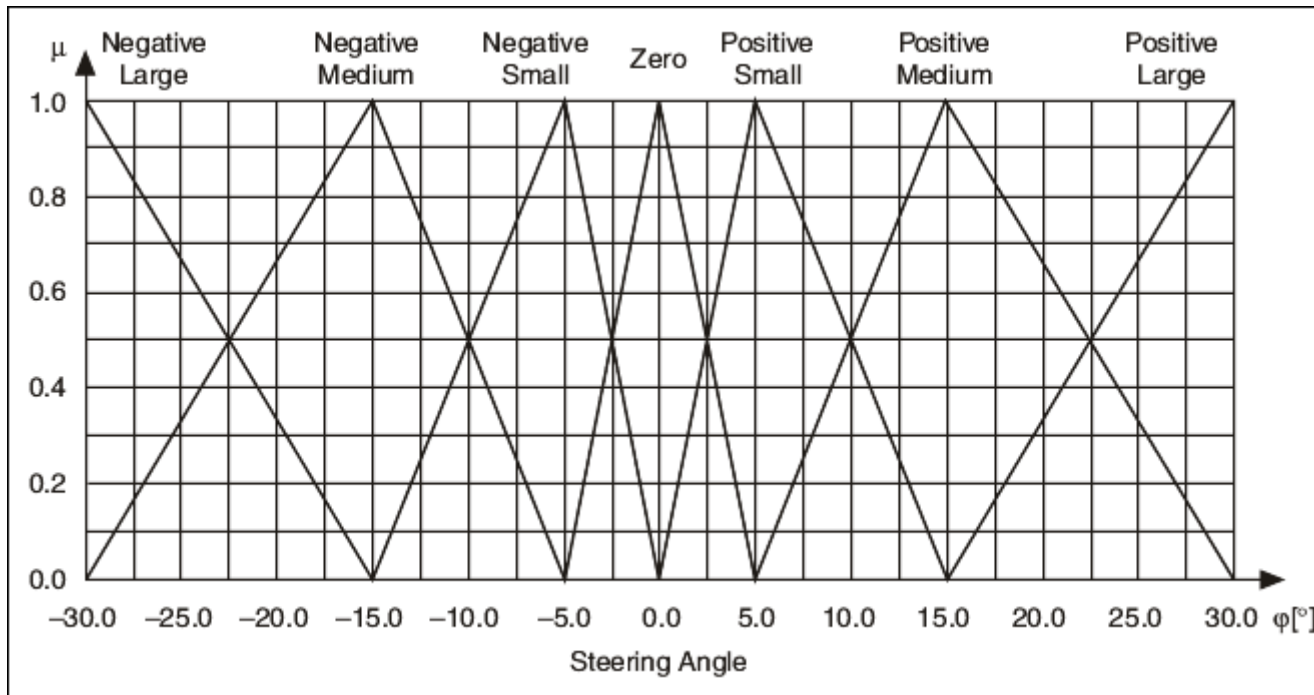
For the purposes of this tutorial, you can use approximate values for the points of the membership functions.



Creating Output Linguistic Variables

You create output linguistic variables similarly to how you create input linguistic variables in the Fuzzy System Designer. Complete the following steps to create an output linguistic variable that corresponds to the Steering Angle ϕ output linguistic variable of the vehicle maneuvering example.

1. Click the **Add Output Variable** button to the right of the **Output variables** list to display the Edit Variable dialog box.
2. Enter steering-angle in the **Name** text box.
3. Enter a **minimum** value of -30 and a **maximum** value of 30. This range specifies that the amount that the steering wheel can turn ranges from -30.0 to 30.0 degrees.
4. Click the **Add Membership Function** button to create a new membership function for the steering-angle output variable.
5. Enter neg-large in the **Name** text box.
6. Select **Triangle** from the **Shape** pull-down menu to specify that a triangle function determines the degrees of membership for the linguistic variable.
7. Select the color you want to use for the membership function in the **Membership functions graph** from the **Color** picker.
8. Enter -30, -30, and -15 in the fields of the **Points** array. These points specify the values of the linguistic variable corresponding to the base and top points, in order from left to right and base to top, of the membership function.
9. Repeat steps 4 through 8 to create each of the membership functions shown in the following graph.



10. Click the **OK** button to save your changes.
11. Select **File»Save As** to save the fuzzy system as `vehicle_maneuvering.fs` in an easily accessible location.

Creating a Rule Base with Pre-Generated Rules

Rules describe, in words, the relationships between input and output *linguistic variables* based on their linguistic terms. The *rule base* of a fuzzy system determines the output values of the fuzzy system based on the input values.

The vehicle-position input linguistic variable you created in [step 1](#) has five membership functions, and the vehicle-orientation input linguistic variable has seven. Therefore, you can construct rules to associate 35 possible combinations of the linguistic terms of these input linguistic variables with the linguistic terms of the steering-angle output linguistic variable you created in [step 2](#).



NOTE

The table in the [Complete Rule Bases](#) topic illustrates this rule base.

Complete the following steps to create the complete rule base for the vehicle maneuvering example. You must have created *both input linguistic variables* and the *output linguistic variable* before creating this rule base.

1. Click the **Rules** tab of the Fuzzy System Designer to display the **Rules** page.
2. Select **Operate»Pre-Generate Rules** to display the Pre-Generate Rules dialog box.
3. Select **AND (Minimum)** from the **Antecedent connective** pull-down menu. This *antecedent connective* specifies that the smallest degree of membership of the antecedents determines the truth value of the aggregated rule antecedent for each rule.

4. Enter a **Degree of support** of 1 to specify the weight that you want to apply to each rule. The final rule weight for each rule is equal to the **Degree of support** multiplied by the truth value of the aggregated rule antecedent.
5. Select **Minimum** from the **Consequent implication** pull-down menu to specify that the fuzzy logic controller truncates the output membership functions at the value of the corresponding rule weights before performing **defuzzification**. Defuzzification is the process of converting the degrees of membership of output linguistic variables into numerical values.
6. Click the **OK** button to return to the **Rules** page. The Fuzzy System Designer populates the **Rules** list with all 35 combinations of linguistic terms of the input variables. Each rule uses the same antecedent connective, degree of support, and implication method that you specified in the Pre-Generate Rules dialog box.

Notice that the consequent of each rule in the **Rules** list is THEN 'steering-angle' IS 'neg-large'. However, the linguistic term of the steering-angle output variable depends on the combination of linguistic terms of the input variables. Therefore, you must modify the pre-generated rules to use the correct consequents.

Modifying Pre-Generated Rules

Complete the following steps to modify the pre-generated rules to use the correct consequents.

1. Select the first rule in the **Rules** list to display an **IF THEN** diagram that shows the **Antecedents** and **Consequents** of the rule.
2. In the **THEN** column, select **neg-small** as the linguistic term for the **steering-angle** output linguistic variable. Notice the consequent of the rule in the **Rules** list changes to THEN 'steering-angle' IS 'neg-small'.
3. Repeat steps 1 and 2 for each rule using the consequents in the following table:

AND		Vehicle Position x [m]				
		Left	Left Center	Center	Right Center	Right
Vehicle Orientation β [°]	Left Down	Negative Small	Negative Medium	Negative Medium	Negative Large	Negative Large
	Left	Positive Small	Negative Small	Negative Medium	Negative Large	Negative Large
	Left Up	Positive Medium	Positive Small	Negative Small	Negative Medium	Negative Large
	Up	Positive Medium	Positive Medium	Zero	Negative Medium	Negative Medium
	Right Up	Positive Large	Positive Medium	Positive Small	Negative Small	Negative Medium
	Right	Positive Large	Positive Large	Positive Medium	Positive Small	Negative Small
	Right Down	Positive Large	Positive Large	Positive Medium	Positive Medium	Negative Small

4. Select **File»Save** to save the fuzzy system.

The fuzzy system now has a [complete linguistic rule base](#) with which to analyze input data and produce output data. However, the rule base returns the output data as linguistic terms. The fuzzy controller must defuzzify the output data before the output can apply to the control structure that the fuzzy system automates. You therefore must specify a defuzzification method for the fuzzy controller to use.

Specifying a Defuzzification Method

After you create the rule base for a fuzzy system, you must specify how a fuzzy controller performs [defuzzification](#) for the system.

In the vehicle maneuvering example, you must supply a continuous output signal to control the steering angle of the vehicle. Therefore, you must use a defuzzification method that calculates the best compromise between any rules that apply at a given time. According to the guidelines for [selecting a defuzzification method](#) the [Center of Maximum \(CoM\)](#), [Center of Area \(CoA\)](#), and [Center of Sums \(CoS\)](#) defuzzification methods all fit this requirement.

Complete the following steps to specify the defuzzification method.

1. On the **Rules** page of the Fuzzy System Designer, select the method you want to use from the **Defuzzification method** pull-down menu.
2. Select **File»Save** to save the fuzzy system.

The fuzzy system is complete. However, before you integrate the fuzzy system into the control structure you want to automate, you must test the functionality of the system.

Testing the Fuzzy System

You can test the relationship between the input and output values of a fuzzy system to validate the rule base of the fuzzy system.

Complete the following steps to test the vehicle maneuvering fuzzy system you created in the previous steps.

1. Click the **Test System** tab of the Fuzzy System Designer to display the **Test System** page.
2. Enter an **Input value** of 5 for the vehicle-position input linguistic variable. Recall from creating the input linguistic variables in [step 1](#) that a value of 5 for vehicle-position corresponds to the center linguistic term.
3. Enter an **Input value** of -30 for the vehicle-orientation input linguistic variable, which corresponds to the left-down linguistic term.
4. In the **Invoked Rules** table at the bottom of the page, notice that the fuzzy system invokes rule 15: IF 'vehicle-position' IS 'center' AND 'vehicle-orientation' IS 'left-down' THEN 'steering-angle' IS 'neg-medium'. Recall from creating the output linguistic variable in [step 2](#) that the neg-medium linguistic term for the steering-angle variable corresponds to the range between -30 and -5. If you configured the fuzzy system correctly, the **Output value** indicator for the steering-angle displays a value within that range. The fuzzy controller calculates the **Output value** based on the weight of the rule, the consequent implication method, and the defuzzification method.
5. (Optional) Use the **Input variable 1** and **Input variable 2** slide controls to sweep a range of values for the input linguistic variables and observe the corresponding changes in the **Input/Output relationship** graph. You can use this graph to verify that the rule base is reasonable and complete. For example, if the **Input/Output relationship** graph displays 0 at some points, the rule base might be incomplete.
6. Fix any errors you find while testing and save the fuzzy system.
7. Click the **Close** button to exit the Fuzzy System Designer.

Controlling the Fuzzy System

After you create a fuzzy system in the Fuzzy System Designer, you can use the FL Fuzzy Controller VI to implement a controller for the fuzzy system. The FuzzyEx Car Backward Parking VI, located in the `labview\examples\control\fuzzy\Car Parking` directory, illustrates how to implement a controller for a fuzzy system.



Open example



Find related examples

This example uses the FL Load Fuzzy System VI to load both the `Truck - forward.fs` and `Truck - backward.fs` files. These files represent the fuzzy systems for moving the vehicle forward and backwards, respectively. Each FL Load Fuzzy System VI passes a .fs file to a FL Fuzzy Controller VI, which implements a fuzzy controller for the fuzzy system.

The fuzzy system you created in this tutorial is similar to the `Truck - backward.fs` file. You can implement a fuzzy controller for the fuzzy system you created in the same fashion that the FuzzyEx Car Backward Parking VI implements a controller for the `Truck - backward.fs` file.

Tutorial: Modifying a Fuzzy System with the Fuzzy Logic VIs

In the [Using the Fuzzy System Designer](#) tutorial, you design a fuzzy system interactively using the Fuzzy System Designer and then implement a fuzzy controller for that system using the FL Fuzzy Controller VI. You also can design a fuzzy system programmatically using the Fuzzy Logic VIs. However, in most cases, it is easier to use the Fuzzy System Designer to design a fuzzy system first and then use the Fuzzy Logic VIs to make programmatic modifications to the system. In this tutorial, you modify a fuzzy system programmatically using the Fuzzy Logic VIs.

This tutorial uses the example of a greenhouse control system to illustrate how to modify a fuzzy system using the Fuzzy Logic VIs. The `greenhouse.fs` file, located in the `labview\examples\control\fuzzy\Dynamic greenhouse controller` directory, represents the fuzzy system that you modify in this chapter.

Next [1. Observing the Fuzzy System](#)



Observing the Fuzzy System

Before you modify a fuzzy system, you must familiarize yourself with the design of the system. Complete the following steps to open and observe the greenhouse fuzzy system in the Fuzzy System Designer.

1. Select **Tools»Control Design and Simulation»Fuzzy System Designer** to launch the Fuzzy System Designer.
2. Select **File»Open** and navigate to `greenhouse.fs` in the `labview\examples\control\fuzzy\Dynamic greenhouse controller` directory.
3. Click the **OK** button to open the greenhouse fuzzy system in the file dialog box. The **Variables** page displays details about the input and output variables of the fuzzy system.

4. Observe the input and output linguistic variables. The greenhouse fuzzy system has two input variables, Temperature and Humidity, and two output variables, Electric Roof and Water Spills.
5. Observe the ranges and membership functions for the linguistic variables. For example, the Temperature input variable represents the temperature, in degrees Celsius, inside the greenhouse, with a range of 0 to 50 and three membership functions: Cold, Normal, and Warm. You can see on the **Input variable membership functions** graph that a trapezoid function represents the shape of the Cold membership function. The left base, left top, right top, and right base points of this membership function are 0, 0, 13, and 21, respectively.
6. Click the **Rules** tab to display the **Rules** page. The **Rules** list displays nine rules for the fuzzy system, which correspond to all possible combinations of the membership functions of the two input linguistic variables. Also notice that the **Defuzzification method** for the fuzzy system is **Center of Area (CoA)**.
7. Select the first rule in the **Rules** list: IF 'Temperature' IS 'Cold' AND 'Humidity' IS 'Dry' THEN 'Electric Roof' IS 'Closed' ALSO 'Water Spills' IS 'Moderate'. Notice the characteristics of this rule. For example, the **Antecedent connective** is **AND (Minimum)**, the **Degree of support** is **1**, and the **Consequent implication** is **Product**.
8. Continue exploring the fuzzy system in the Fuzzy System Designer. When you are familiar with the system, you can click the **Close** button to exit the **Fuzzy System Designer**.

Loading the Fuzzy System

Before modifying the greenhouse fuzzy system, you must load the .fs file. Complete the following steps to load the greenhouse.fs file.

1. Select **File»New VI** to create a new, blank VI.
2. Press the <Ctrl-E> keys to display the block diagram.
3. Add the FL Load Fuzzy System VI to the block diagram.
4. Right-click the **file path** input of the FL Load Fuzzy System VI and select **Create»Constant** from the shortcut menu.
5. Enter the absolute path to the greenhouse . fs file in the path constant.



NOTE

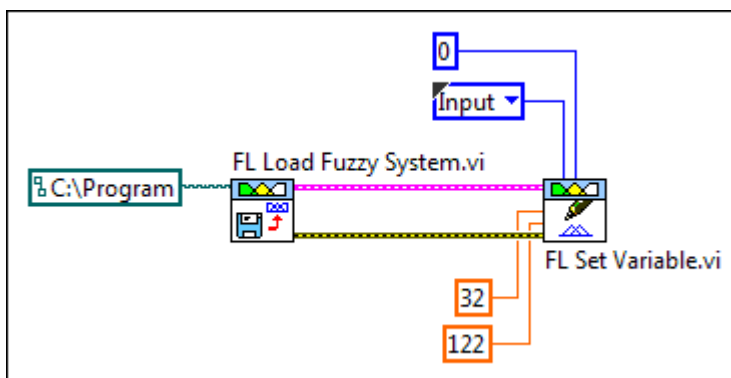
You also can use the File I/O VIs and functions to create a relative path to wire to the **file path** input.

6. Select **File»Save** and save the VI as **Modified Greenhouse Fuzzy System . vi** in an easily accessible location.

Modifying a Linguistic Variable

You can use the Variables VIs to modify the linguistic variables in a fuzzy system. Complete the following steps to modify the range of the Temperature input linguistic variable of the greenhouse fuzzy system to use degrees Fahrenheit instead of degrees Celsius.

1. Add the FL Set Variable VI to the block diagram. This VI modifies the name, range, or membership functions of a linguistic variable in a fuzzy system.
2. Wire the **fuzzy system out** output of the FL Load Fuzzy System VI to the **fuzzy system in** input of the FL Set Variable VI.
3. Right-click the **input/output** input of the FL Set Variable VI and select **Create»Constant** from the shortcut menu.
4. Select **Input** from the **input/output** constant to specify that you want to modify an input linguistic variable.
5. Right-click the **variable index** input of the FL Set Variable VI and select **Create»Constant** from the shortcut menu.
6. Enter 0 in the **variable index** constant to specify that you want to modify the first linguistic variable. The index is zero-based and corresponds to the order in which the linguistic variable was created. Recall that the Temperature variable is the first input linguistic variable in the fuzzy system.
7. Create constants for the **minimum** and **maximum** inputs of the FL Set Variable VI and set them to 32 and 122, respectively. This new range represents the possible temperature, in degrees Fahrenheit, inside the greenhouse. The block diagram now resembles the following image:



8. Save the VI.

This VI sets the new range of the Temperature variable when it runs. However, you still must modify the membership functions of the Temperature variable to complete the Celsius to Fahrenheit conversion.

Modifying Membership Functions

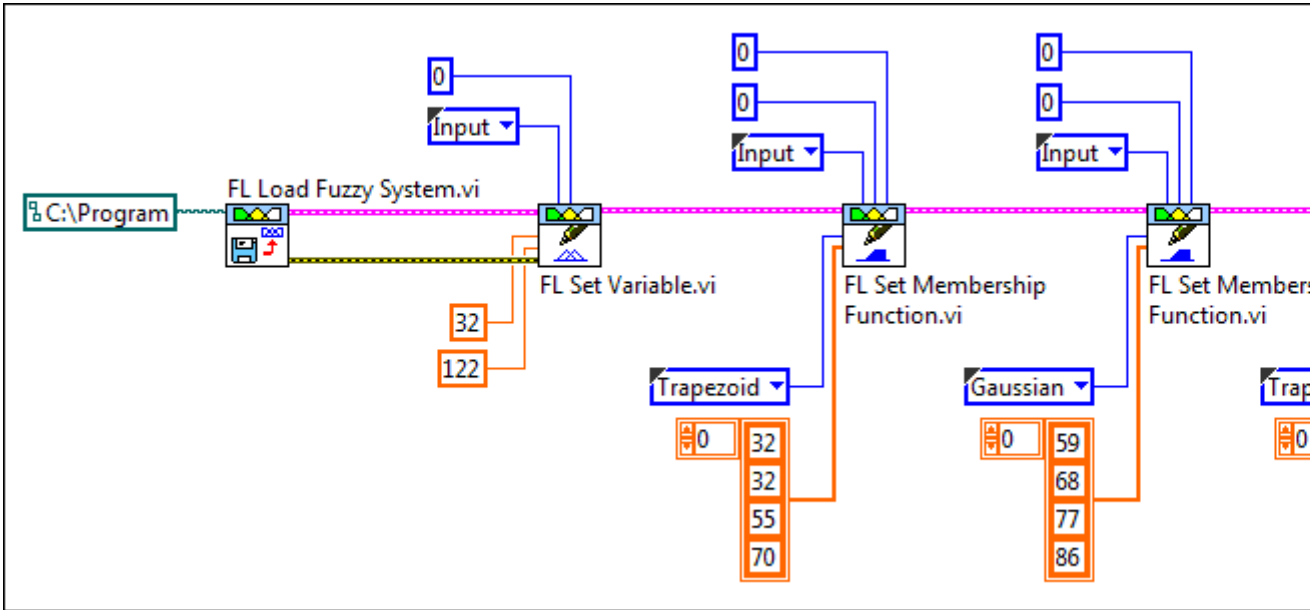
You can use the Membership VIs to modify the membership functions for linguistic variables in a fuzzy system. Complete the following steps to modify the membership functions of the Temperature input linguistic variable to use degrees Fahrenheit instead of degrees Celsius.

1. Add the FL Set Membership Function VI to the block diagram.
2. Wire the **fuzzy system out** output of the FL Set Variable VI to the **fuzzy system in** input of the FL Set Membership Function VI.
3. Create constants for the **membership function index**, **variable index**, and **input/output** inputs of the FL Set Membership Function VI and set them to 0, 0, and

- Input**, respectively. These inputs specify that you want to modify the first membership function, Cold, of the Temperature input linguistic variable.
4. Create a constant for the **shape** input of the FL Set Membership Function VI.
 5. Select **Trapezoid** from the **shape** constant to specify the shape of the function that determines the degrees of membership for the linguistic variable.
 6. Create a constant for the **points** input of the FL Set Membership Function VI. This input specifies the values of the linguistic variable corresponding to the base and top points, in order from left to right and base to top, of the membership function.
 7. In the first four elements of the **points** array constant, enter the values 32, 32, 55, and 70, respectively.
 8. Repeat steps 1 through 5 for the Normal and Warm membership functions. For each membership function, add a new FL Set Membership Function VI to the block diagram and wire the **fuzzy system out** output of the previous VI to the **fuzzy system in** input of the next VI. Use the shapes and points shown in the following table:

Membership Function	Shape	Points
Gaussian	59, 68, 77, 86	
Trapezoid	75, 90, 122, 122	

The block diagram now resembles the following image:



9. Save the VI.

Modifying a Rule

You can use the Rules VIs to modify the rules for a fuzzy system. This section demonstrates how to recreate the first rule of the greenhouse fuzzy system:

IF 'Temperature' IS 'Cold' AND 'Humidity' IS 'Dry' THEN 'Electric Roof' IS 'Closed' ALSO 'Water Spills' IS 'Moderate'

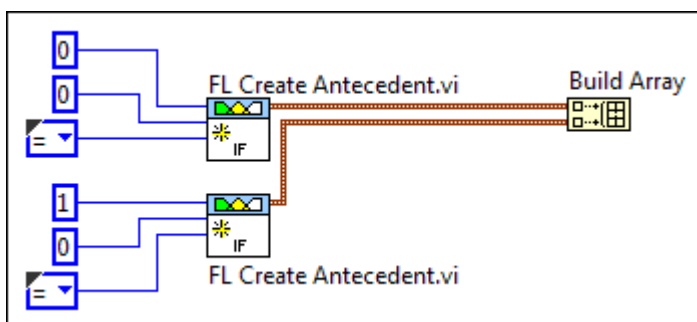
and replace the existing first rule of the fuzzy system with the rule you create.

When you use the Rules VIs to create rules, you must create the rules in parts. First, you create the antecedents. Then, you create the consequents. Finally, you combine the antecedents and consequents for a specific rule. The next three steps of this tutorial walk you through each of these actions.

Creating Antecedents

The first step in creating a rule is to create the antecedents of the rule. Complete the following steps to create the 'Temperature' IS 'Cold' and 'Humidity' IS 'Dry' antecedents.

1. Add the FL Create Antecedent VI to the block diagram. This VI creates an antecedent, or IF portion, of a rule for a fuzzy system.
2. Create constants for the **variable index** and **membership function index** inputs of the FL Create Antecedent VI and set them both to 0. These inputs specify that you want to associate the Temperature input linguistic variable with the Cold membership function.
3. Create a constant for the **condition** input of the FL Create Antecedent VI.
4. Select = from the condition constant to calculate the degree of membership of the linguistic variable within the membership function. FL Create Antecedent VI now creates the 'Temperature' IS 'Cold' antecedent.
5. Repeat steps 1 through 4 to create the 'Humidity' IS 'Dry' antecedent. In the greenhouse fuzzy system, Dry is the first membership function of the second input linguistic variable, so use 1 as the **variable index** and 0 as the **membership function index**.
6. Add a Build Array function to the block diagram.
7. Resize the Build Array function to display two input elements.
8. Wire the **antecedent** output of the first FL Create Antecedent VI to the first **element** input of the Build Array function.
9. Wire the **antecedent** output of the second FL Create Antecedent VI to the second **element** input of the Build Array function. The section of block diagram you just created resembles the following image:



10. Save the VI.

Creating Consequents

You create rule consequents similarly to how you create rule antecedents. Complete the following steps to create the 'Electric Roof' IS 'Closed' and 'Water Spills' IS 'Moderate' consequents.

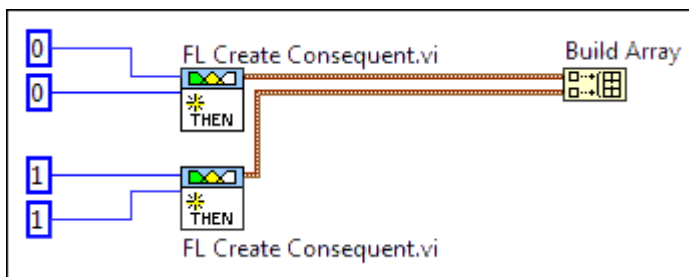
1. Add the FL Create Consequent VI to the block diagram. This VI creates a consequent, or THEN portion, of a rule for a fuzzy system.
2. Create constants for the **variable index** and **membership function index** inputs of the FL Create Consequent VI and set them both to 0. These inputs specify that you want to associate the Electric Roof output linguistic variable with the Closed membership function.



NOTE

Unlike the FL Create Antecedent VI, the FL Create Consequent VI does not have a **condition** input. The FL Create Consequent VI always returns the consequent corresponding to the degree of membership of the output variable within the membership function.

3. Repeat steps 1 and 2 to create the 'Water Spills' IS 'Moderate' consequent. In the greenhouse fuzzy system, Moderate is the second membership function of the second output linguistic variable Water Spills, so set both the **variable index** and **membership function index** to 1.
4. Add a Build Array function to the block diagram.
5. Resize the Build Array function to display two input elements.
6. Wire the **consequent** output of the first FL Create Consequent VI to the first **element** input of the Build Array function.
7. Wire the **consequent** output of the second FL Create Consequent VI to the second **element** input of the Build Array function. The section of block diagram you just created resembles the following image:



8. Save the VI.

Combining the Antecedents and Consequents for a Rule

Complete the following steps to combine the antecedents and consequents you created in the previous steps to form a specific rule.

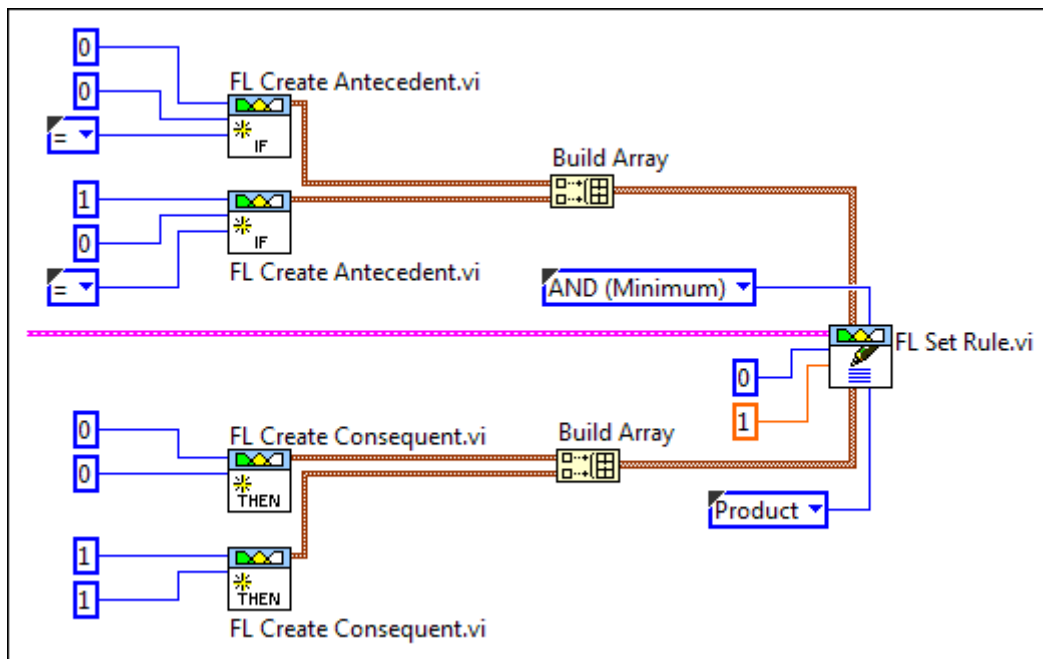
1. Add an FL Set Rule VI to the block diagram. This VI modifies the antecedents, consequents, or relationships of a rule in a fuzzy system.
2. Wire the **fuzzy system out** output of the last FL Set Membership Function VI to the **fuzzy system in** input of the FL Set Rule VI.
3. Create a constant for the **rule index** input of the FL Set Rule VI and set it to 0 to specify that you want to modify the first rule in the fuzzy system.
4. Wire the **appended array** output of the first Build Array function to the **antecedents** input of the FL Set Rule VI.
5. Wire the **appended array** output of the second Build Array function to the **consequents** input of the FL Set Rule VI.
6. Create a constant for the **antecedent connective** input of the FL Set Rule VI to specify how the VI calculates the truth value of the aggregated rule antecedent.
7. Select **AND (Minimum)** from the **antecedent connective** constant. The aggregated rule antecedent now is IF 'Temperature' IS 'Cold' AND 'Humidity' IS 'Dry', where this VI uses the smallest degree of membership of the individual antecedents to calculate the truth value of the aggregated rule antecedent.



NOTE

You cannot specify how this VI calculates the truth value of an aggregated rule consequent. This VI always uses the greatest degree of membership of the individual consequents to calculate the truth value of the aggregated rule consequent.

8. Create a constant for the **consequent implication** input of the FL Set Rule VI. This input specifies the implication method this VI uses to scale the membership functions of the output linguistic variables based on the rule weight.
9. Select **Product** from the **consequent implication** constant to specify that this VI uses the *Product implication method*.
10. Create a constant for the **degree of support** input and set it to 1. This input specifies the weight that you want to apply to the rule. The final rule weight is equal to the **degree of support** multiplied by the truth value of the aggregated rule antecedent. The section of block diagram you just created resembles the following image:



11. Save the VI.

Saving the Fuzzy System

In previous sections of this tutorial, you modified the linguistic variables, membership functions, and rules of the greenhouse fuzzy system. You now can save the fuzzy system to a .fs file. Complete the following steps to save the modified fuzzy system to a .fs file.

1. Add the FL Save Fuzzy System VI to the block diagram.
2. Wire the **fuzzy system out** output of the FL Set Rule VI to the **fuzzy system in** input of the FL Save Fuzzy System VI.
3. Right-click the **file path** input of the FL Save Fuzzy System VI and select **Create»Constant** from the shortcut menu.
4. Enter the absolute path to a modified greenhouse.fs file in the path constant. The FL Save Fuzzy System VI creates the .fs file when it runs.



NOTE

You also can use the File I/O VIs and functions to create a relative path to wire to the **file path** input of the FL Save Fuzzy System VI.

5. Save the Modified Greenhouse Fuzzy System VI.
6. Run the VI. The VI loads the original greenhouse.fs file, modifies the linguistic variable and membership functions you specified, creates new antecedents and consequents for a rule, and saves all the changes to the modified greenhouse.fs file.

You now can use the Fuzzy System Designer to open the modified greenhouse.fs file and observe the modifications you made to the greenhouse fuzzy system.

You also can use the FL Fuzzy Controller VI to implement a fuzzy logic controller for the modified greenhouse fuzzy system. The FuzzyEx Dynamic Fuzzy Controller for a greenhouse example VI in the `labview\examples\control\fuzzy\Dynamic greenhouse controller` directory implements a controller for the greenhouse fuzzy system. This example VI uses the FL Load Fuzzy System VI to load the `greenhouse.fs` file.



Open example



Find related examples

You can load the `modified_greenhouse.fs` file instead and then run the example VI to implement a controller for the modified greenhouse fuzzy system.

Fuzzy System

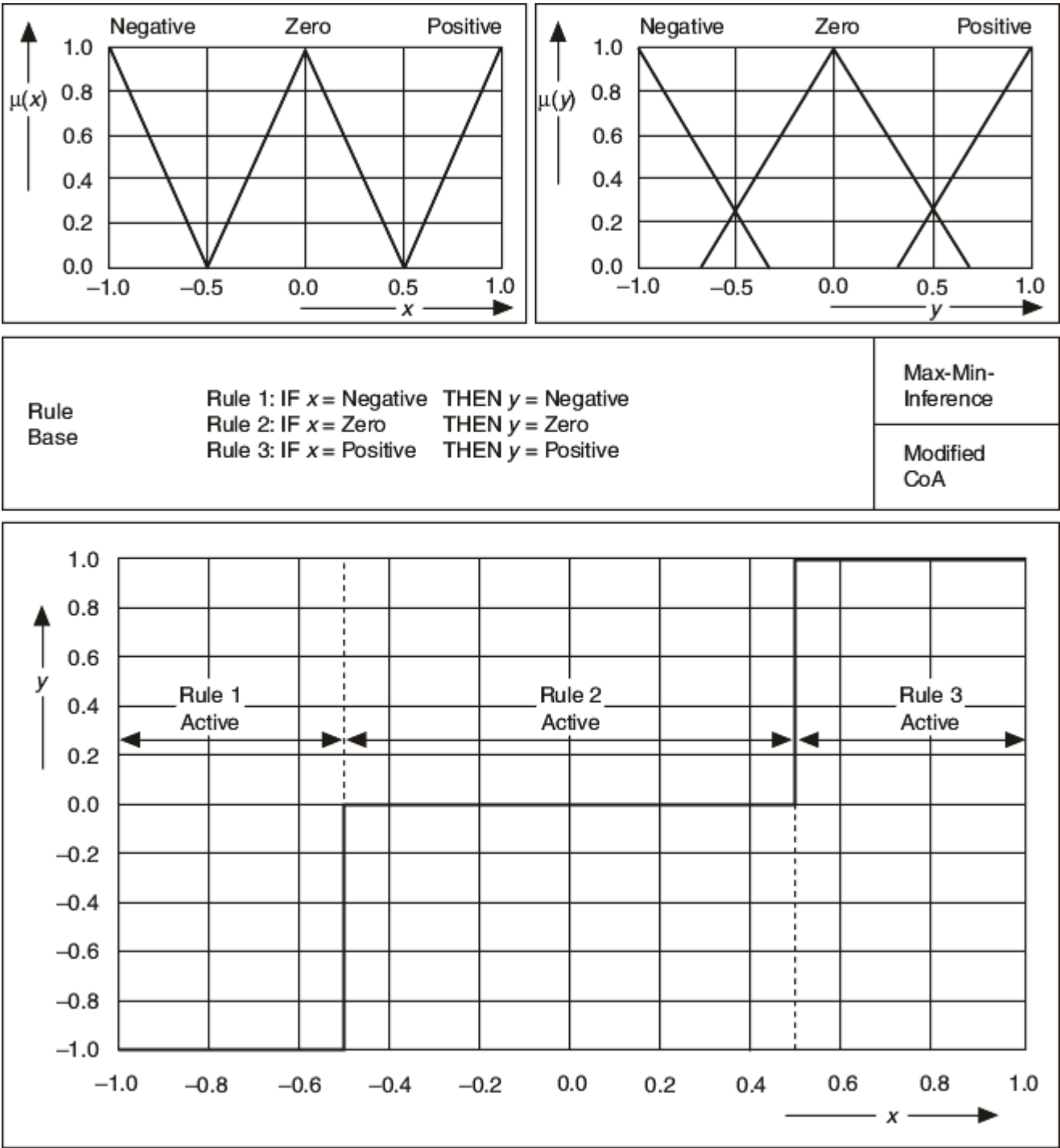
Multi-Loop Control

Most of the PID VIs are polymorphic VIs for use in multiple control-loop applications. For example, you can design a multi-loop PID control application using the PID VI and DAQ functions for input and output. A DAQ analog input function returns an array of data when you configure it for multiple channels. You can wire this array directly into the **process variable** input of the PID VI. The polymorphic type of the PID VI automatically switches from DBL to DBL Array, which calculates and returns an array of output values corresponding to the number of values in the **process variable** array. You also can switch the type of the polymorphic VI manually by right-clicking the VI icon and selecting **Select Type** from the shortcut menu.

When the polymorphic type is DBL Array, other inputs automatically change to array inputs as well. For example, the PID VI inputs **setpoint**, **PID gains**, and **output range** all become array inputs. Each of these inputs can have an array length ranging from 1 to the array length of the **process variable** input. If the array length of any of these inputs is less than the array length of the **process variable** input, the PID VI reuses the last value in the array for other calculations. For example, if you specify only one set of PID gains in the **PID gains** array, the PID VI uses these gains to calculate each **output** value corresponding to each **process variable** input value. Other polymorphic PID and Fuzzy Logic VIs operate in the same manner.

Non-Overlapping Input Terms

The following image shows the controller I/O characteristic that results when non-overlapping antecedent terms describe the input variable.



Only one rule is active for each input situation that leads to the stepped controller characteristic.



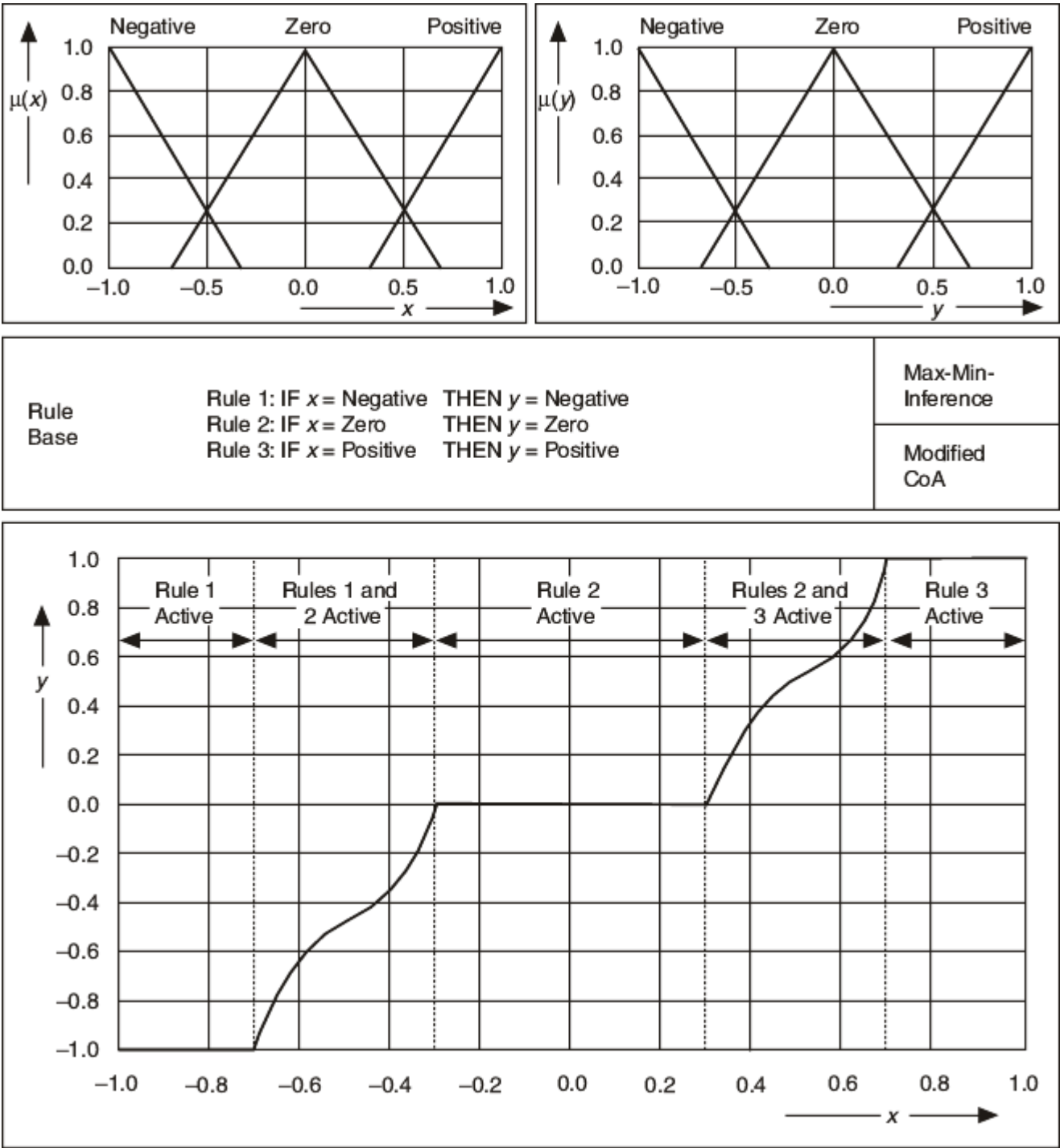
RELATED INFORMATION

- [I/O Characteristics of Fuzzy Controllers](#) on page 1744
- [Specifying an Antecedent Connective](#) on page 1756
- [Creating a Rule Base](#) on page 1788

Partially Overlapping Input Terms

To illustrate how the *I/O characteristics of a fuzzy controller* depend on design parameters such as the *rule base* and *membership function* specifications of the fuzzy system, consider a single-input fuzzy controller. Many of the characteristics of a single-input fuzzy controller apply to fuzzy controllers with two or more inputs.

The following image shows the I/O characteristic of a fuzzy controller for a fuzzy system that has only three *linguistic terms* for the input variable x and the output variable y . The rule base consists of three rules, which indicate that the increasing input values cause the output value to increase.



The resulting controller characteristic shows nonlinear behavior. You obtain different intervals within the controller characteristic because the input linguistic terms partially overlap. The rule base has only one valid rule outside of the overlapping regions. The output therefore has a constant value determined by the output linguistic term of the output linguistic variable, which is independent of the degree of truth for that rule.

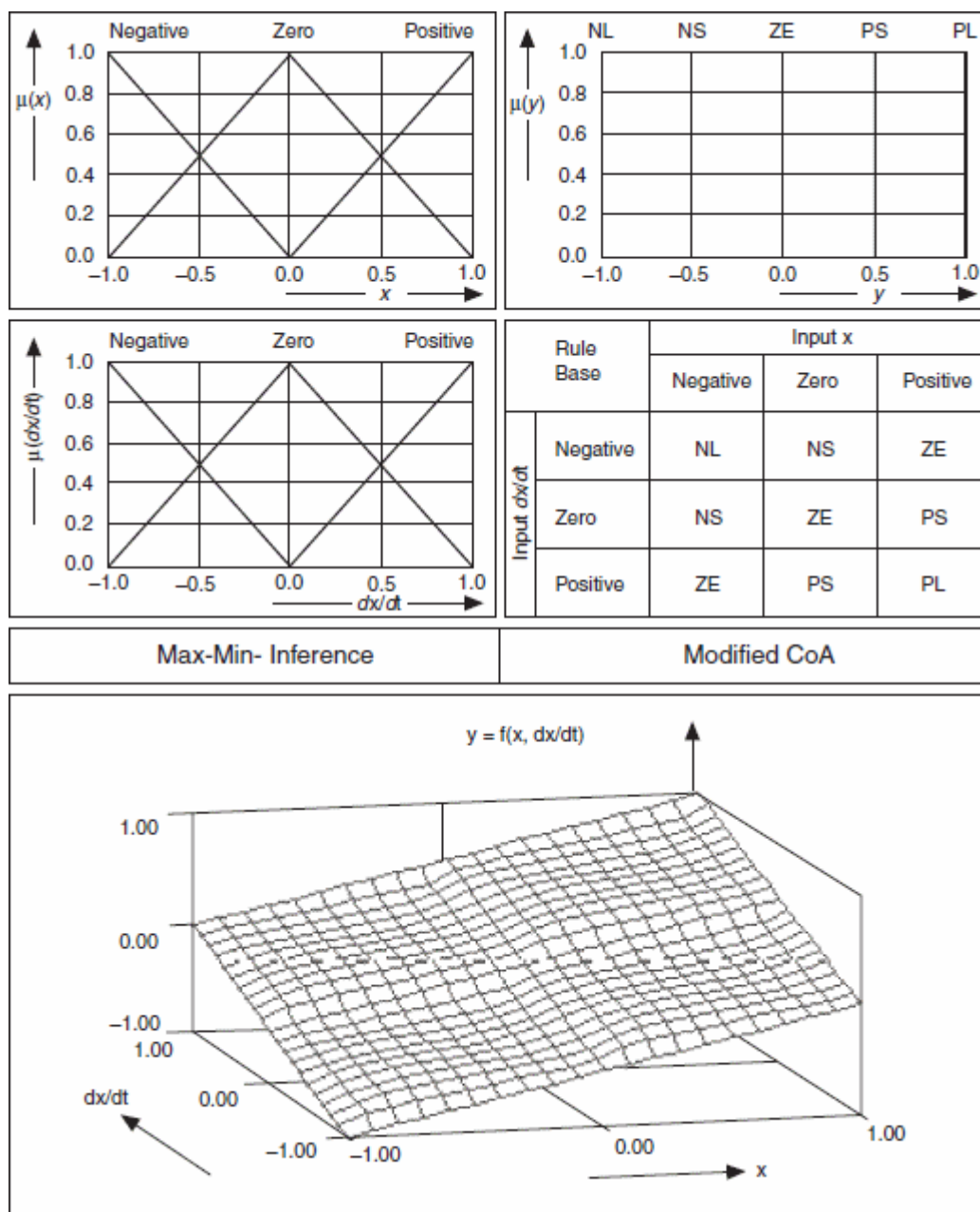
The overlapping sections of the *antecedent terms* lead to the rising intervals of the controller characteristic. Within these parts, two rules are simultaneously active. The different consequent terms, weighted by the degrees of truth of the different active rules, determine the output value. Notice that the overlapping triangular consequent terms cause the rising edges of the controller characteristic to be nonlinear.

Fuzzy Controllers

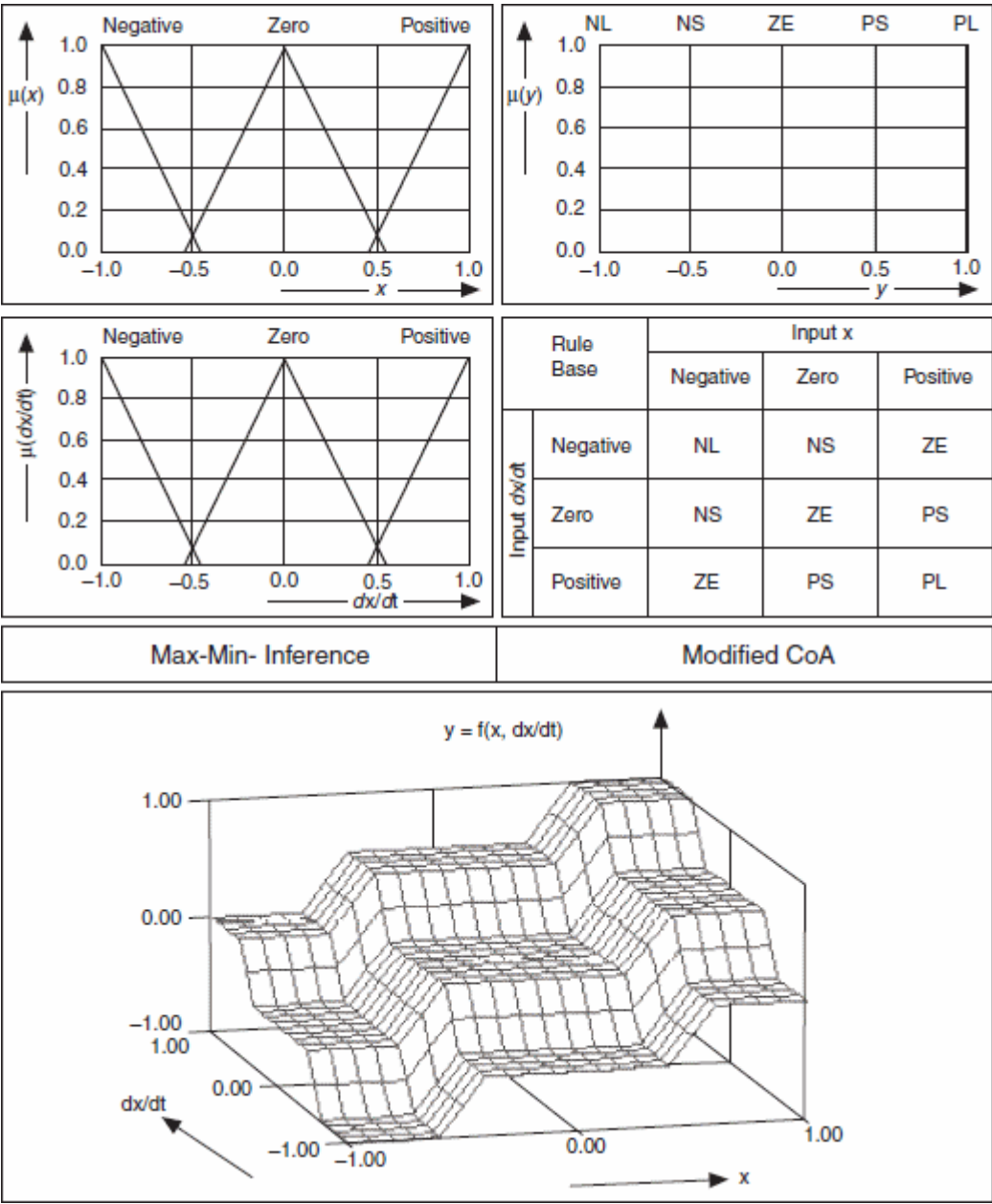
Dual-Input Fuzzy Controllers

In principle, the conclusions about *I/O characteristics* for single-input fuzzy controllers also are valid for controllers with two or more inputs. However, using the AND (Minimum) *antecedent connective* to combine the different input conditions raises an additional nonlinear effect. The fuzzy controller considers the antecedent of the rule with the lowest degree of truth as shown in the *Reproducing a Given I/O Characteristic* example.

The following image shows the I/O characteristic field for a dual-input fuzzy controller.

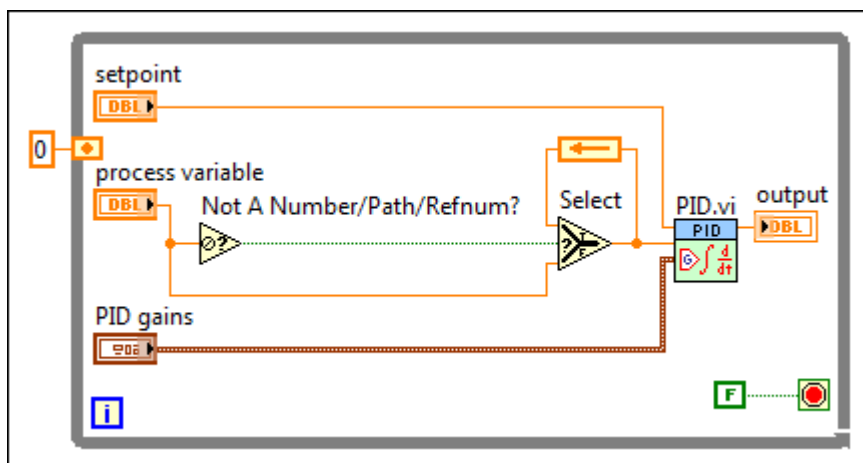


Because the AND (Minimum) antecedent connective is nonlinear, the characteristic field is not exactly linear despite the membership functions that overlap entirely for both input variables. Non-overlapping membership functions yield a stepped characteristic field with constant planes, as shown in the following image.



Fault Protection

If a PID VI receives an invalid input, such as NaN (Not a Number), the VI outputs NaN until you reinitialize the VI. However, you can use the Not A Number/Path/Refnum? function to check for invalid inputs and respond in a way that is appropriate to the application. The following block diagram uses the Not A Number/Path/Refnum? function with a Select function and a Feedback Node to reuse the last valid input when the VI receives an invalid input.



Refer to the General PID with Fault Protection VI in the `labview\examples\control\PID` directory for an example of a single-channel PID implementation that includes fault protection.



Open example



Find related examples

Filtering Control Inputs

You can use the PID Control Input Filter VI to filter unwanted, high-frequency noise from measured values in a control application. For example, if you are measuring process variable values using a DAQ device, you can use this VI to filter noise from input values in the control loop before passing the values to control functions such as the PID VI.

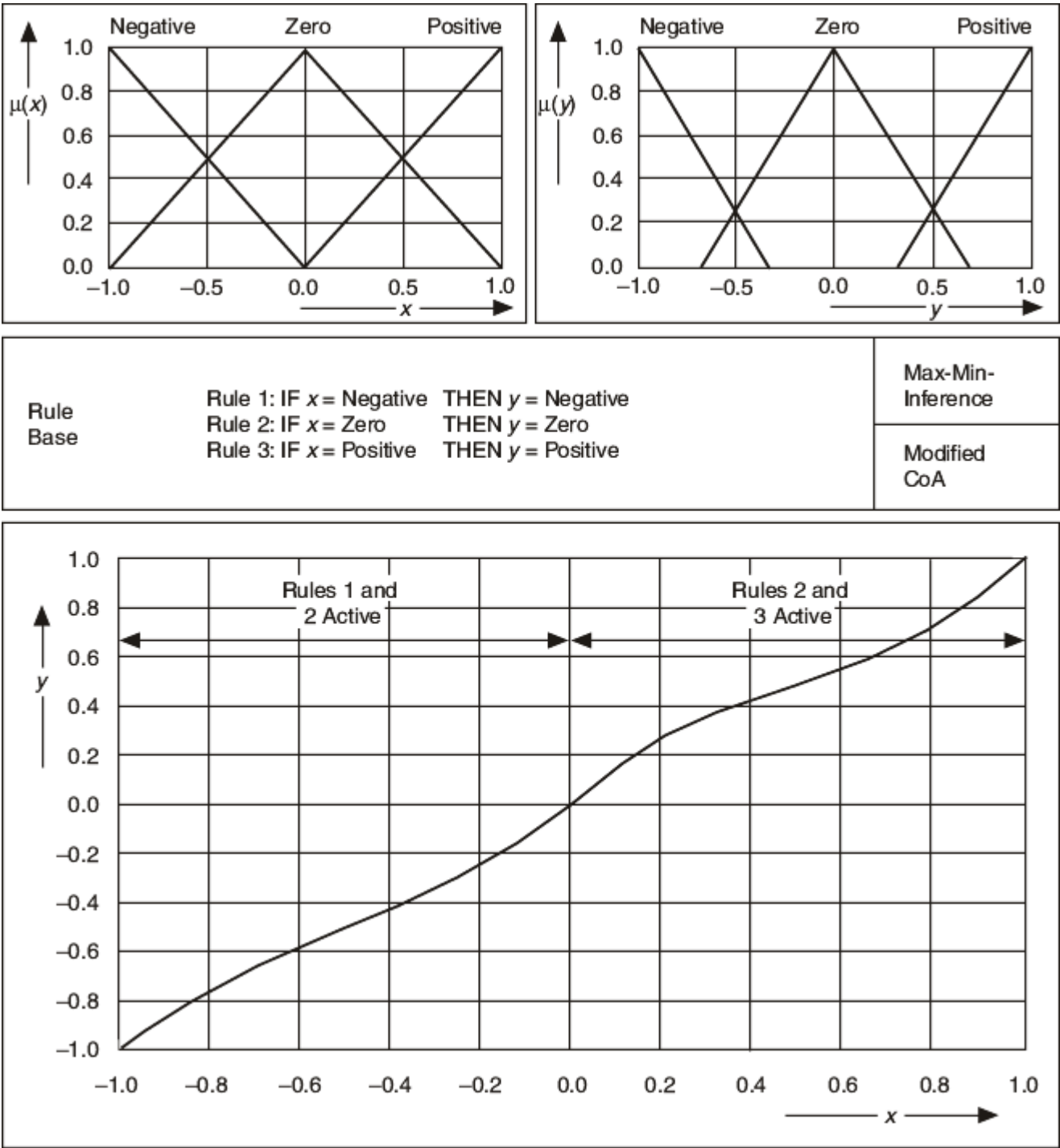
When filtering noise, set the sampling rate of the control system to at least 10 times the fastest time constant of the physical system. This ensures that any frequency components of the measured signal greater than 1/10 of the sampling frequency are a result of noise in the measured signal. Gains in the PID controller can amplify this noise and produce unnecessary wear on actuators and other system components.

The PID Control Input Filter VI uses a lowpass fifth-order finite impulse response (FIR) filter. The cutoff frequency of the lowpass filter is 1/10 of the sampling frequency, regardless of the actual sampling frequency value.

You can use the Filters PtByPt VIs or the LabVIEW Digital Filter Design Toolkit to perform additional filtering tasks. Refer to the National Instruments website for more information about or to purchase the Digital Filter Design Toolkit.

Fully Overlapping Input Terms

The following image shows the resulting controller I/O characteristic for antecedent terms that overlap entirely. The consequent term distribution and the rule base remain unchanged for this case.



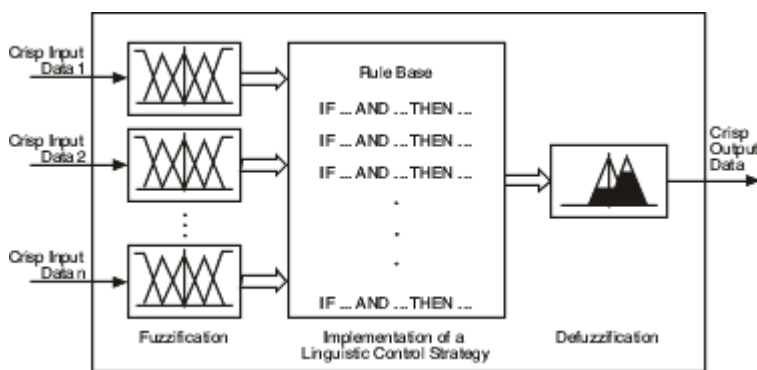
RELATED INFORMATION

- [I/O Characteristics of Fuzzy Controllers](#) on page 1744
- [Specifying an Antecedent Connective](#) on page 1756
- [Creating a Rule Base](#) on page 1788

Fuzzy Controllers

You can use fuzzy controllers to control *fuzzy systems*. Most traditional control algorithms require a mathematical model of the system you want to control. However, many physical systems are difficult or impossible to model mathematically. In addition, many processes are either nonlinear or too complex for you to control with traditional strategies. However, if you can describe a control strategy qualitatively, you can use fuzzy logic to create a fuzzy controller that emulates a heuristic rule-of-thumb strategy.

The following image illustrates the process of a fuzzy controller.



Fuzzification

Fuzzification is the process of associating crisp, or numerical, input values with the *linguistic terms* of the corresponding input *linguistic variables*.

For example, a fuzzy controller might associate the temperature reading from a thermometer with the linguistic terms cold, moderate, and hot for the current temperature linguistic variable. Depending on the *membership functions* for the linguistic terms, the temperature value might correspond to one or more of the linguistic terms.

Implementing a Linguistic Control Strategy

After a fuzzy controller fuzzifies the input values of a fuzzy system, the fuzzy controller uses the corresponding input linguistic terms and the *rule base* to determine the resulting linguistic terms of the output linguistic variables.

For example, suppose the current temperature of a room is 50 degrees, which corresponds to a linguistic term of cold with a degree of membership of 0.4. Also suppose the desired temperature is 70, which corresponds to a linguistic term of moderate with a degree of membership of 0.8. The fuzzy controller invokes the following rule of the fuzzy system: IF current temperature is cold AND desired temperature is moderate, THEN heater setting is low.

Notice that this rule consists of two antecedents, "current temperature is cold" and "desired temperature is moderate". The truth value of each antecedent is equal to the degree of membership of the linguistic variable within the corresponding linguistic term. The fuzzy logic controller uses an antecedent connective to determine how to calculate the

truth value of the aggregated rule antecedent. Suppose the invoked rule in this example uses the AND (Minimum) antecedent connective, which specifies to use the smallest degree of membership of the antecedents as the truth value of the aggregated rule antecedent. Therefore, the truth value of the aggregated rule antecedent is 0.4.

You can specify a degree of support for each rule of a fuzzy system. The weight of a rule is equal to the degree of support multiplied by the truth value of the aggregated rule antecedent. The fuzzy controller uses an implication method to scale the membership functions of an output linguistic variable based on the rule weight before performing defuzzification.

Defuzzification

Defuzzification is the process of converting the degrees of membership of output linguistic variables within their linguistic terms into crisp numerical values. A fuzzy controller can use one of several mathematical methods to perform defuzzification. The most accurate *defuzzification method* for a fuzzy controller varies based on the control application.

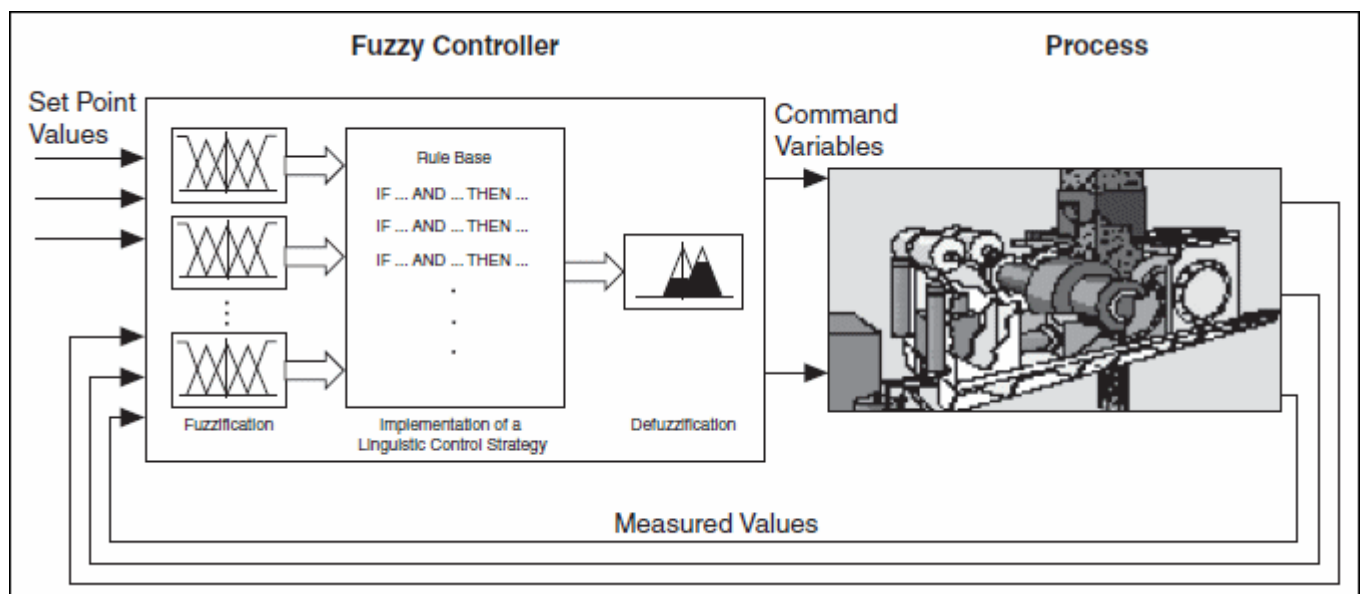
Related Information

[I/O Characteristics of Fuzzy Controllers](#)

[Fuzzy Controllers in Closed-Loop Control Structures](#)

Fuzzy Controllers in Closed-Loop Control Structures

The most common use case for fuzzy controllers is in closed-loop control structures. The most basic structure of closed-loop control applications uses sensor signals as input signals for the system and the outputs as command values to drive the actuators of the process. The following image shows a simple closed-loop control structure with a fuzzy controller.



Pure fuzzy control applications are more the exception than the rule. In most cases the fuzzy controller outputs serve as reference parameters, such as gains, that you provide to a conventional controller instead of directly to driving actuators.

Because you can regard a fuzzy controller as a nonlinear characteristic field controller, it has no internal dynamic aspects. Thus, you must implement any dynamic property by an appropriate preprocessing of the measured input data.

PID and Fuzzy Logic VIs enable you to integrate fuzzy controllers and PID control. For example, you can use fuzzy controllers with conventional PID controllers or create hybrid fuzzy-PI controllers. Use either the Fuzzy System Designer or the Fuzzy Logic VIs to design a fuzzy system. Then use the FL Fuzzy Controller VI to implement a fuzzy controller for the fuzzy system. Finally, you can integrate the fuzzy controller into a control structure you create using the PID VIs.



RELATED INFORMATION

[PID Control](#) on page 1747

[Using Fuzzy Controllers with PID Controllers](#) on page 1760

[Fuzzy-PI Controllers](#) on page 1741

[Fuzzy-PI Controllers](#) on page 1741

[Using Fuzzy Controllers with PID Controllers](#) on page 1760

Fuzzy Logic

Fuzzy logic is a method of rule-based decision making used for expert systems and process control. Fuzzy logic differs from traditional Boolean logic in that fuzzy logic allows for partial membership in a set.

Traditional Boolean logic is two-valued in the sense that a member either belongs to a set or does not. Values of one and zero represent the membership of a member to the set with one representing absolute membership and zero representing no membership. Fuzzy logic allows for partial membership, or a degree of membership, which might be any value along the continuum of zero to one.

You can think of fuzzy logic in terms of fuzzy systems, which are systems of variables associated with fuzzy logic, and fuzzy controllers, which use defined rules to control a fuzzy system based on the current values of input variables. You can use the Fuzzy Logic VIs to design and control fuzzy systems. You also can use the Fuzzy System Designer to design fuzzy systems interactively.

Related Information

Fuzzy Systems

Fuzzy Controllers

Tutorial: Using the Fuzzy System Designer

Tutorial: Modifying a Fuzzy System with the Fuzzy Logic VIs



RELATED INFORMATION

[Fuzzy Systems](#) on page 1740

[Fuzzy Controllers](#) on page 1737

[Fuzzy Systems](#) on page 1740

[Fuzzy Controllers](#) on page 1737

Fuzzy System Design Overview

The topics in this section use an example fuzzy system that automates a vehicle to park itself to illustrate the basics of fuzzy system design.



NOTE

The topics in this section describe the general process of designing a fuzzy system. You also can refer to the tutorials [Using the Fuzzy System Designer](#) and [Modifying a Fuzzy System with the Fuzzy Logic VIs](#) for examples with step-by-step instructions of how to design a fuzzy system with the PID and Fuzzy Logic VIs.

A fuzzy system consists of three main parts: linguistic variables, membership functions, and rules. Therefore, fuzzy system design is a three-step process:

1. [Creating Linguistic Variables](#)
2. [Creating Membership Functions](#)
3. [Creating a Rule Base](#)

Fuzzy Systems

A fuzzy system is a system of variables that are associated using fuzzy logic. A [fuzzy controller](#) uses defined rules to control a fuzzy system based on the current values of input variables. You can use the [Fuzzy System Designer](#) and the [Fuzzy Logic VIs](#) to design and control fuzzy systems.

Fuzzy systems consist of three main parts: linguistic variables, membership functions, and rules.

Linguistic Variables

[Linguistic variables](#) represent, in words, the input variables and output variables of the system you want to control. For a heater, you might have two input linguistic variables, current temperature and desired temperature, and one output linguistic variable, heater setting. Each linguistic variable has a range of expected values. For example, the range of current temperature might be 0 to 100 degrees. The range of desired temperature might be 50 to 80 degrees.

A fuzzy controller requires at least one input linguistic variable and one output linguistic variable.

Linguistic Terms and Membership Functions

Linguistic terms represent, in words, categories for the values of a linguistic variable. The linguistic variables current temperature and desired temperature each might include the linguistic terms cold, moderate, and hot. The linguistic variable heater setting might include the linguistic terms off, low, and high.

Membership functions are numerical functions corresponding to linguistic terms. A membership function represents the degree of membership of linguistic variables within their linguistic terms. The degree of membership is continuous between 0 and 1, where 0 is equal to 0% membership and 1 is equal to 100% membership. For example, the linguistic variable current temperature might have full membership (1) within the linguistic term hot at 100 degrees, no membership (0) within that term at 70 degrees or less, and partial membership at all temperatures between 70 and 100 degrees.

Rules

Rules describe, in words, the relationships between input and output linguistic variables based on their linguistic terms. For example, you might define the following rule:

IF current temperature is cold AND desired temperature is moderate, THEN heater setting is low.

The clauses "current temperature is cold" and "desired temperature is moderate" are the antecedents of this rule. The AND connective specifies how the fuzzy logic controller relates the two antecedents to determine the truth value for the aggregated rule antecedent. The clause "heater setting is low" is the consequent of this rule.

A **rule base** is the set of rules for a fuzzy system. The rule base is equivalent to the control strategy of the controller.

Related Information

[Fuzzy System Design Overview](#)

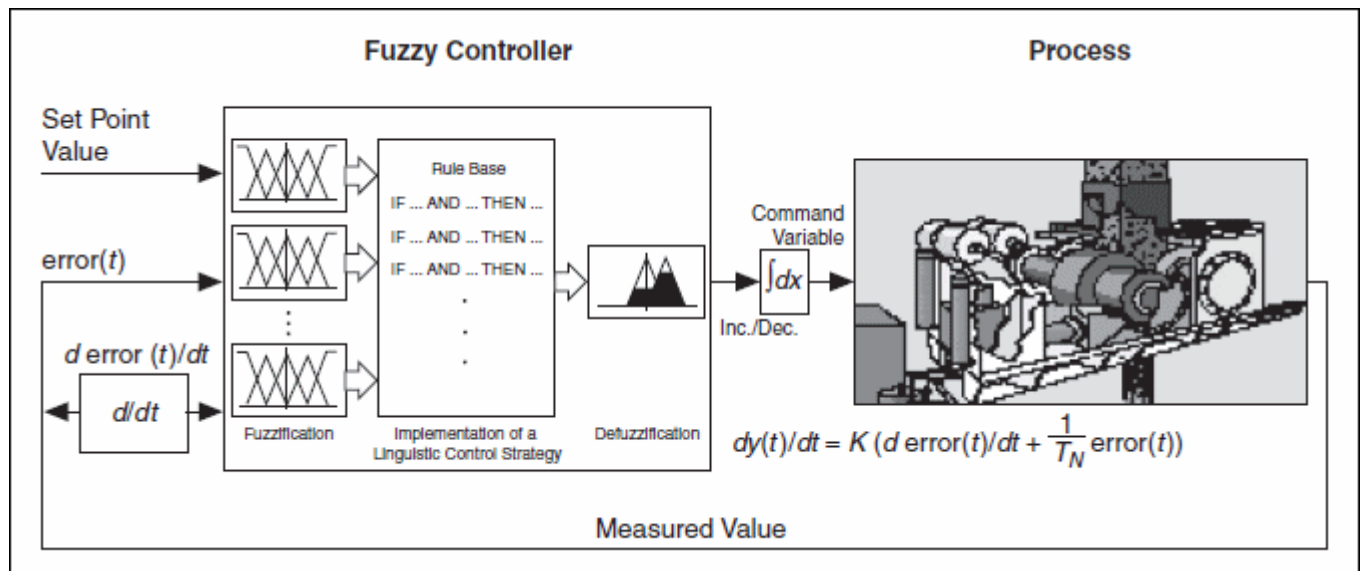
[Defuzzification Methods](#)

Fuzzy-PI Controllers

A PI controller is a controller that produces proportional plus integral control action. The PI controller has only one input and one output. The output value increases when the input value increases.

A fuzzy-PI controller is a generalization of the conventional PI controller that uses an error signal and its derivative as input signals. Fuzzy-PI controllers have two inputs and one output. Multiple inputs allow for greater control diversity for a fuzzy-PI controller over a conventional PI controller.

The fuzzy-PI controller shown in the following image uses the error signal $e(t)$ and its derivative $de(t)/dt$ from the measured data preprocessing step as inputs. If the output signal describes the necessary difference toward the current output value, you need a subsequent integrator device to build up the command variable value.



Advantages of Fuzzy-PI Controllers

The benefit of the fuzzy-PI controller is that it does not have a special operating point. The rules evaluate the difference between the measured value and the set value, which is the error signal. The rules also evaluate the tendency of the error signal to determine whether to increase or decrease the control variable. The absolute value of the command variable has no influence.

Another advantage of a fuzzy-PI controller over a conventional PI controller is that it can implement nonlinear control strategies and that it uses linguistic rules. With fuzzy-PI controllers, you can consider the error tendency by itself when the error becomes small.

Gain Scheduling

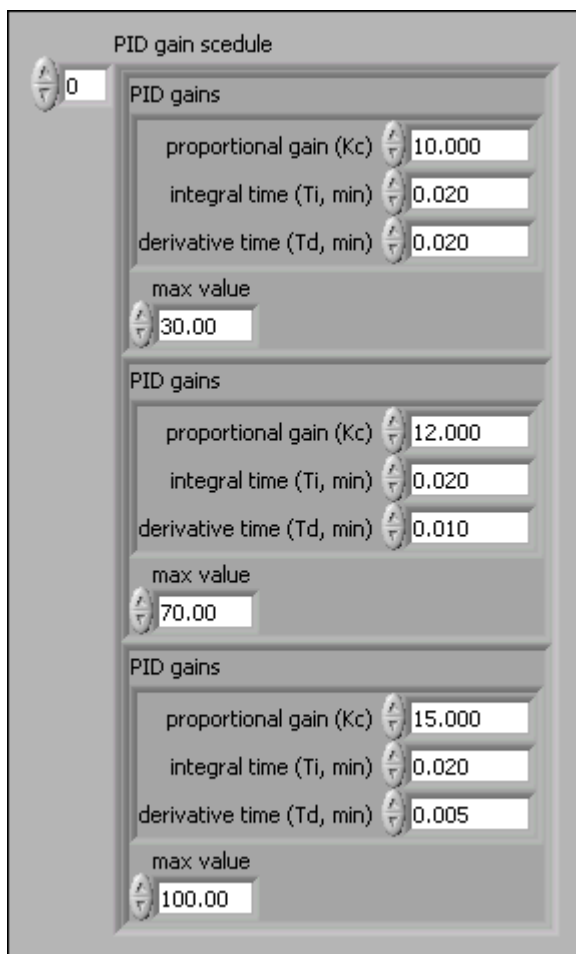
Gain scheduling refers to a system where you change controller parameters based on measured operating conditions. For example, the scheduling variable can be the setpoint, the process variable, a controller output, or an external signal. For historical reasons, the term gain scheduling is used even if other parameters such as derivative time, T_d , or integral time, T_i change. Gain scheduling effectively controls a system whose dynamics change with the operating conditions.

You can use the PID Gain Schedule VI, to apply different sets of PID parameters for different regions of operation of your controller. Because most processes are nonlinear, PID

parameters that produce a desired response at one operating point might not produce a satisfactory response at another operating point. The PID Gain Schedule VI selects and outputs one set of PID gains from a gain schedule based on the current value of its **gain scheduling value** input. For example, to implement a gain schedule based on the value of the process variable, wire the process variable value to the **gain scheduling value** input and wire the **PID gains out** output to the **PID gains** input of the PID VI.

The **PID gain schedule** input of the PID Gain Schedule VI is an array of clusters of **PID gains** and corresponding maximum values. Each set of **PID gains** corresponds to the range of input values from the **max value** of the previous element of the array to the **max value** of the same element of the array. The input range of the **PID gains** of the first element of **PID gain schedule** is all values less than or equal to the corresponding **max value**.

The following front panel shows a VI that uses the setpoint value as the gain scheduling variable with a default range of 0 to 100.



The following table summarizes parameter ranges specified in the previous image:

Range	Parameters
$0 \leq SP \leq 30$	$K_c = 10$ $T_i = 0.02$ $T_d = 0.02$
$30 \leq SP \leq 70$	$K_c = 12$ $T_i = 0.02$ $T_d = 0.01$

Range	Parameters
$70 \leq SP \leq 100$	$K_c = 15 \ T_i = 0.02 \ T_d = 0.005$

I/O Characteristics of Fuzzy Controllers

You can consider a fuzzy controller to be a nonlinear characteristic field controller. The rule base and membership functions of the fuzzy system determine the behavior of the controller. Because the controller has no internal dynamic aspects, the I/O characteristics can describe the transient response of the controller completely.

This section includes examples that show the I/O characteristics of a fuzzy controller under various of conditions. Select an item from the list for information about I/O characteristics of a fuzzy controller with the described conditions.

- Partially Overlapping Input Terms
- Fully Overlapping Input Terms
- Non-Overlapping Input Terms
- Undefined Input Term Intervals
- Singletons as Output Terms with Fully Overlapping Input Terms
- Different Overlapping Degrees of Membership Functions for Output Terms
- Wide and Small Membership Functions for the Output Terms
- Using the Mean of Maximum (MoM) Defuzzification Method
- Reproducing a Given I/O Characteristic
- Dual-Input Fuzzy Controllers

These examples show that you can use a fuzzy controller to perform arbitrary I/O operations. The number of input and output linguistic terms depends on the desired characteristic type and the precision to which you approximate the given I/O characteristic.



RELATED INFORMATION

[Fuzzy Controllers](#) on page 1737

[Creating a Rule Base](#) on page 1788

[Creating Membership Functions](#) on page 1791

[Partially Overlapping Input Terms](#) on page 1731

[Fully Overlapping Input Terms](#) on page 1735

[Non-Overlapping Input Terms](#) on page 1729

[Undefined Input Term Intervals](#) on page 1758

[Singletons as Output Terms with Fully Overlapping Input Terms](#) on page 1753

[Different Overlapping Degrees of Membership Functions for Output Terms](#) on page 1802

[Wide and Small Membership Functions for Output Terms](#) on page 1809

[Using the Mean of Maximum \(MoM\) Defuzzification Method](#) on page 1806

[Reproducing a Given I/O Characteristic](#) on page 1775

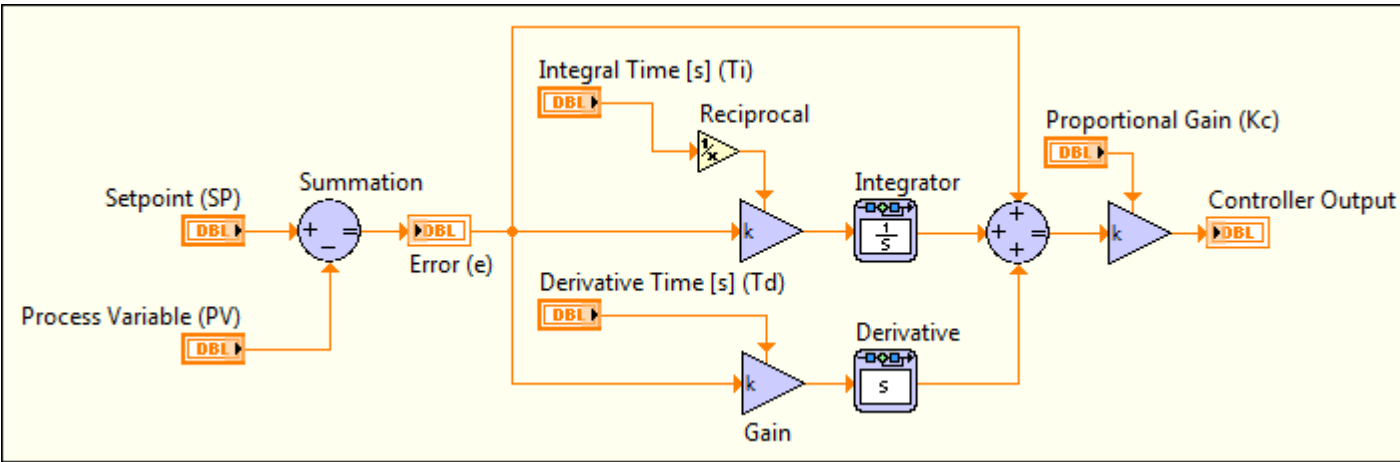
[Dual-Input Fuzzy Controllers](#) on page 1733

PID Algorithms

The following equations describe the basic PID algorithm. This section also provides information about [how the PID VIs implement this algorithm](#), as well as the [advanced PID algorithm](#).

Basic PID Controller

The following simulation diagram represents the basic PID (Proportional-Integral-Derivative) controller:



NOTE
Constructing a simulation diagram like the one in the previous image requires the LabVIEW Control Design and Simulation Module. However, the PID VIs implement PID controllers for you. You do not need the Control Design and Simulation Module to build PID controllers. Refer to the National Instruments website at ni.com for more information about the Control Design and Simulation Module.

Error Calculation (e)

A PID controller compares the setpoint SP to the process variable PV to obtain the error e, as follows:

$$e = SP - PV$$

where	e is the error
SP is the setpoint	
PV is the process variable	

Controller Action (u(t))

The PID controller calculates the controller action, u(t) as follows:

$$u(t) = K_c \left(e + \frac{1}{T_i} \int_0^t e dt + T_d \frac{de}{dt} \right)$$

where	K_c is controller gain
T_i is the integral time in minutes, also called the reset time	
T_d is the derivative time in minutes, also called the rate time	

This equation involves three actions that simultaneously apply to the output of the controller:

Proportional Action (P)

Proportional action P is proportional to the error e . This term defines how quickly the controller must respond to changes in the error. The following formula represents the proportional action:

$$u_P(t) = K_c e$$

Integral Action (I)

Integral action I is proportional to the integral of the error e . This term is generally used to correct steady-state error and to avoid disturbances in the system. The following formula represents the integral action:

$$u_I(t) = \frac{K_c}{T_i} \int_0^t e dt$$

Derivative Action (D)

Derivative action D is proportional to the derivative of the error e . This term works as a brake to the response, avoiding the system to overshoot. The following formula represents the derivative action:

$$u_D(t) = K_c T_d \frac{de}{dt}$$

Related Information

[PID Topologies](#)

[Implementing the PID Algorithm with the PID VIs](#)

[Advanced PID Algorithm](#)

PID Control

Proportional-Integral-Derivative (PID) controllers are common feedback controllers. In PID control, you specify a process variable and a setpoint. The process variable is the system parameter you want to control, such as temperature, pressure, or flow rate, and the setpoint is the desired value for that system parameter. A PID controller determines a controller output value, for example the heater power or valve position, and applies the

controller output value to the system to drive the process variable toward the setpoint value.

Use the PID VIs to develop the following control applications based on PID controllers:

- Proportional (P); proportional-integral (PI); proportional-derivative (PD); and proportional-integral-derivative (PID) algorithms
- Gain-scheduled PID
- PID autotuning
- Error-squared PID
- Lead-Lag compensation
- Setpoint profile generation
- Multi-loop cascade control
- Feedforward control



RELATED INFORMATION

[Implementing the PID Algorithm with the PID VIs](#) on page 1764

[Gain Scheduling](#) on page 1742

[Autotuning PID Controllers \(PID and Fuzzy Logic\)](#) on page 1774

[Advanced PID Algorithm](#) on page 1770

[Lead/Lag](#) on page 1768

[Setpoint Profile Generation](#) on page 1752

[Multi-Loop Control](#) on page 1729

[Lead/Lag](#) on page 1768

[PID Algorithms](#) on page 1745

[Designing a Control Strategy](#) on page 1799

[Implementing PID Controllers with the PID VIs](#) on page 1763

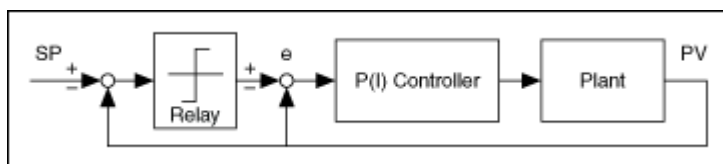
[Autotuning PID Controllers \(PID and Fuzzy Logic\)](#) on page 1774

PID Relay Autotuning Technique

The PID Relay [autotuning technique](#), which is used by the [Autotuning Wizard](#) and the PID Relay instance of the PID Online Autotuning VI, uses the setpoint relay experiment to determining the information needed to tune the controller.

The following image illustrates the autotuning procedure excited by the setpoint relay experiment, which connects a relay and an extra feedback signal with the setpoint. The setpoint relay steps the setpoint up or down, exciting the autotuning procedure and allowing the controller to respond to the change and tune the system for an acceptable response. The PID Autotuning VI and the PID Relay instance of the PID Online Autotuning

polymorphic VI directly implement this process. The existing controller remains in the loop.



For most systems, the nonlinear relay characteristic generates a limiting cycle, from which the autotuning algorithm identifies the relevant information needed for PID tuning. If the existing controller is proportional only, the autotuning algorithm identifies the ultimate gain K_u and ultimate period T_u . If the existing model is PI or PID, the autotuning algorithm identifies the dead time τ and time constant T_p , which are two parameters in the integral-plus-deadtime model.

$$G_P(s) = \frac{e^{-\tau s}}{T_p s}$$

PID Topologies

Standard PID literature provides several ways to define a PID controller. In general, PID controllers are classified using the following forms.

Academic Form

In the Academic form, the proportional parameter P actuates on the integral I and derivative D terms, as represented by the following equation:

$$\frac{U(s)}{Y(s)} = K_c \left(1 + \frac{1}{sT_i} + sT_d \right)$$

The [algorithms](#) implemented by the PID VIs are based on the Academic form with time expressed in minutes.

Parallel Form

In the Parallel form, the three parameters P , I , and D are independent of each other and control each interaction. This form also is referred to as non-interacting PID. The Parallel form is represented by the following equation:

$$\frac{U(s)}{Y(s)} = K_p + \frac{K_i}{s} + sK_d$$

Series Form

In the Series form, the parameters are coupled and in series with each component, as represented by the following equation:

$$\frac{U(s)}{Y(s)} = K'_C \left(1 + \frac{1}{sT_I}\right) (1 + sT_D)$$

Converting Topologies

You can use the PID Structure Conversion VI to convert a PID controller from one topology, or form, to another.

Programmatically Autotuning a Previously Implemented PID Controller

You can use the PID Online Autotuning VI to tune a controller you implement or previously implemented with the PID Advanced VI.



NOTE

You must use the PID Advanced VI to implement the controller. You can implement PID controllers with various PID VIs, but PID Advanced is the only VI that supports putting the controller under manual control, which is necessary for using the PID Online Autotuning VI. You also can use the PID Advanced Autotuning VI to *implement and autotune a PID controller* from a single VI.

Complete the following steps autotune a previously implemented PID controller.

1. If you have not already done so, *implement a PID controller* using the PID Advanced VI.
2. Put the controller in manual mode by setting the **auto?** input of the PID Advanced VI to FALSE.
3. Add the PID Online Autotuning VI to the block diagram within the control loop but to the left of the PID Advanced VI.
4. *Manually select the polymorphic instance* of the PID Online Autotuning VI.
5. Without deleting the controls, constants, or shift registers that contain the values, delete the wires to the **setpoint**, **process variable**, and **PID gains** inputs of the PID Advanced VI.
6. Wire the values to the **Setpoint in**, **Process Variable**, and **PID gains in** inputs of the PID Online Autotuning VI.
7. Create a Boolean control with latched mechanical action and a default value of FALSE, and wire the control to the **Autotune?** input of the PID Online Autotuning VI.
8. Wire the **setpoint out**, **Process Variable Out**, and **PID gains out** outputs of the PID Online Autotuning VI to the **setpoint**, **process variable**, and **PID gains** inputs of the PID Advanced VI.

9. (Optional) If you are specifying a value for the **dt (s)** input, wire the same value to both VIs.
10. Run the VI.
11. Set **Autotune?** to TRUE to begin autotuning.

Refer to the Autotuning PID Online and Inline VI in the `labview\examples\control\PID` for an example of implementing autotuning with the PID Online Autotuning VI and the PID Advanced VI.



Open example



Find related examples

Programmatically Implementing and Autotuning a PID Controller

You can use the PID Advanced Autotuning VI both to implement and autotune a PID controller.



NOTE

The PID Advanced Autotuning VI incorporates the functionality of the PID Advanced VI and the PID Online Autotuning VI. You also can use the PID Online Autotuning VI to *programmatically autotune a previously implemented PID controller*.

Complete the following steps to implement a tuned PID controller with the PID Advanced Autotuning VI.

1. Add the PID Advanced Autotuning VI to the block diagram within a control loop.
2. Wire the inputs you want to use to implement the controller. Minimally, wire the **setpoint**, **process variable**, and **PID gains** inputs.
3. Create a control for the **autotuning parameters** input and select the **controller type** and **control specification** you want to use.
4. Set the **technique** component of the **autotuning parameters** input to the *autotuning technique* you want to use.
5. Create a Boolean control with latched mechanical action and a default value of FALSE for the **Autotune?** input.
6. Run the VI.
7. Set **Autotune?** to TRUE to begin autotuning.

Refer to the Autotuning PID Online VI in the `labview\examples\control\PID` for an example of implementing autotuning with the PID Advanced Autotuning VI.



Open example



Find related examples

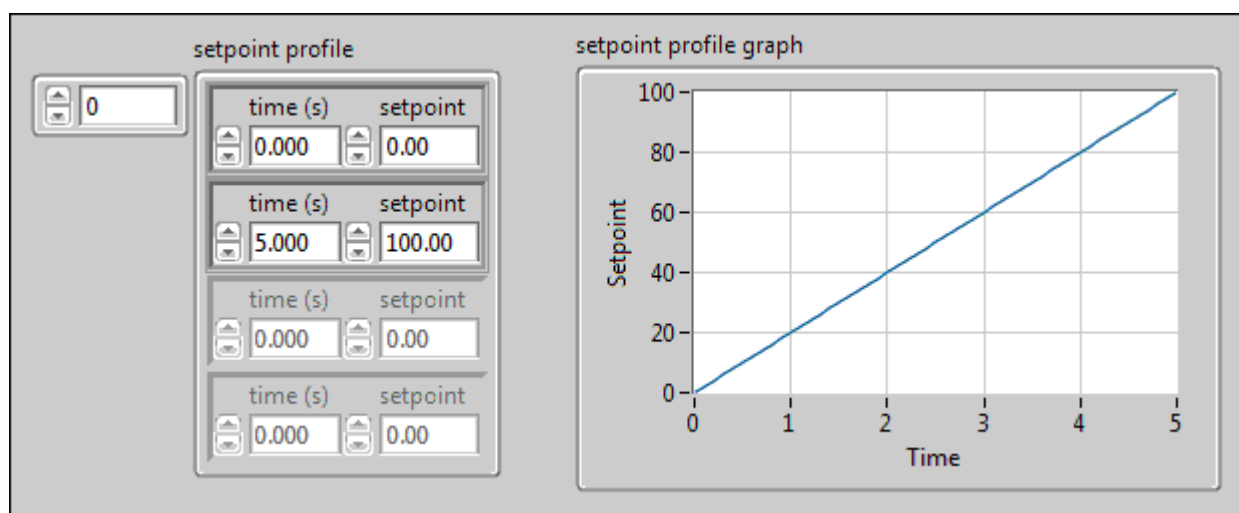
Setpoint Profile Generation

You can use the PID Setpoint Profile VI inside a control loop to generate a profile of setpoint values over time for a ramp and soak type PID application. For example, you might want to ramp the setpoint temperature of an oven control system over time, and then hold, or soak, the setpoint at a certain temperature for another period of time. You can use the PID Setpoint Profile VI to implement any arbitrary combination of ramp, hold, and step functions.

Specify the setpoint profile as an array of pairs of time and setpoint values with the time values in ascending order.

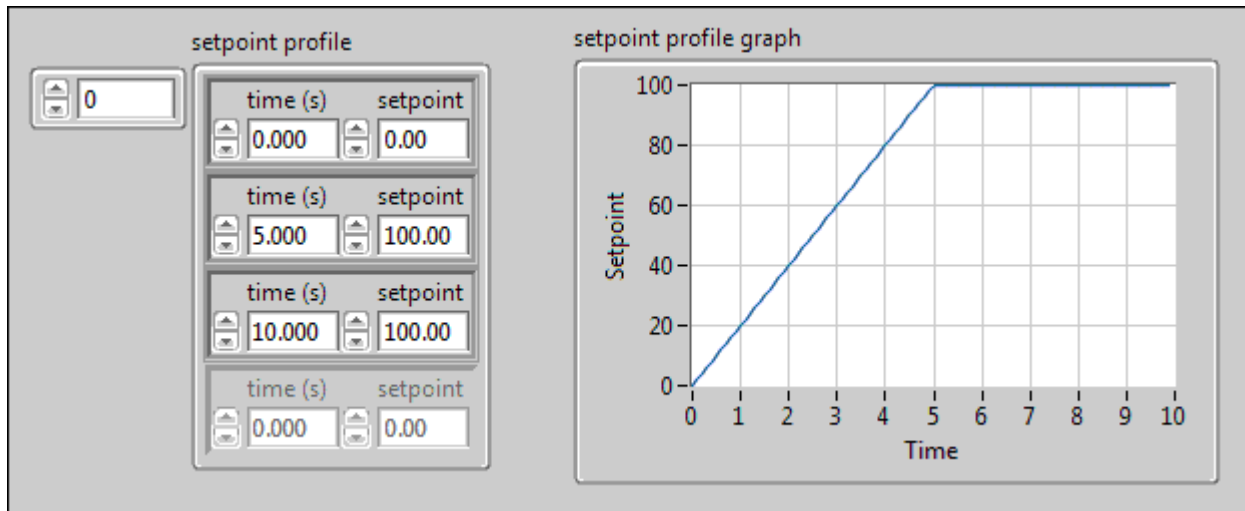
Ramp Setpoint Profile

You can specify a ramp setpoint profile with two setpoint profile array values, as shown in the following image.



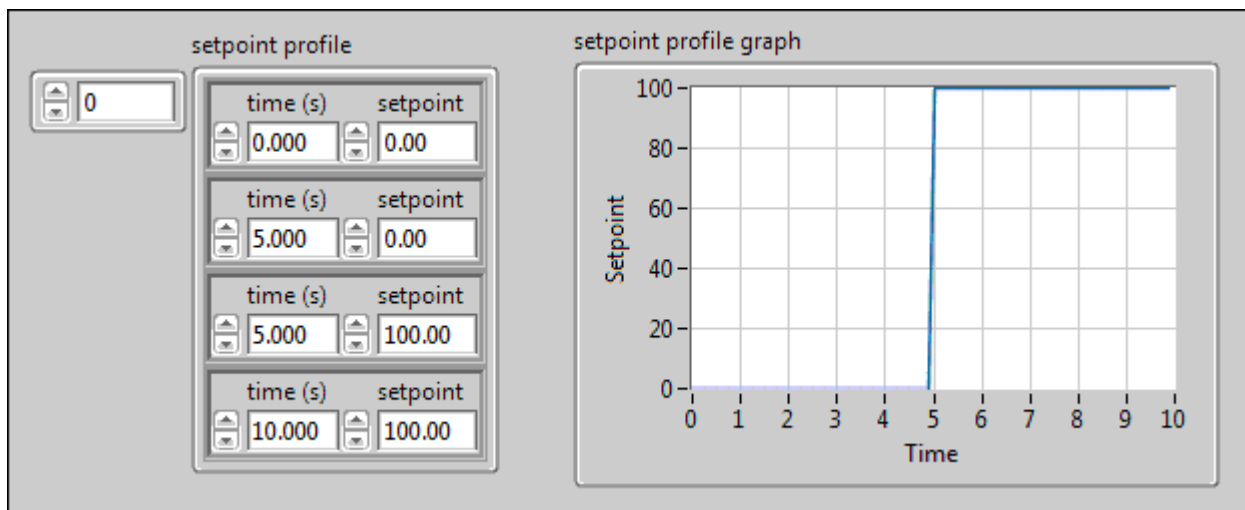
Ramp and Hold Setpoint Profile

A ramp and hold setpoint profile also can have two successive array values with the same setpoint value, as shown in the following image.



Step Setpoint Profile

A step setpoint profile can have two successive array values with the same time value but different setpoint values, as shown in the following image.

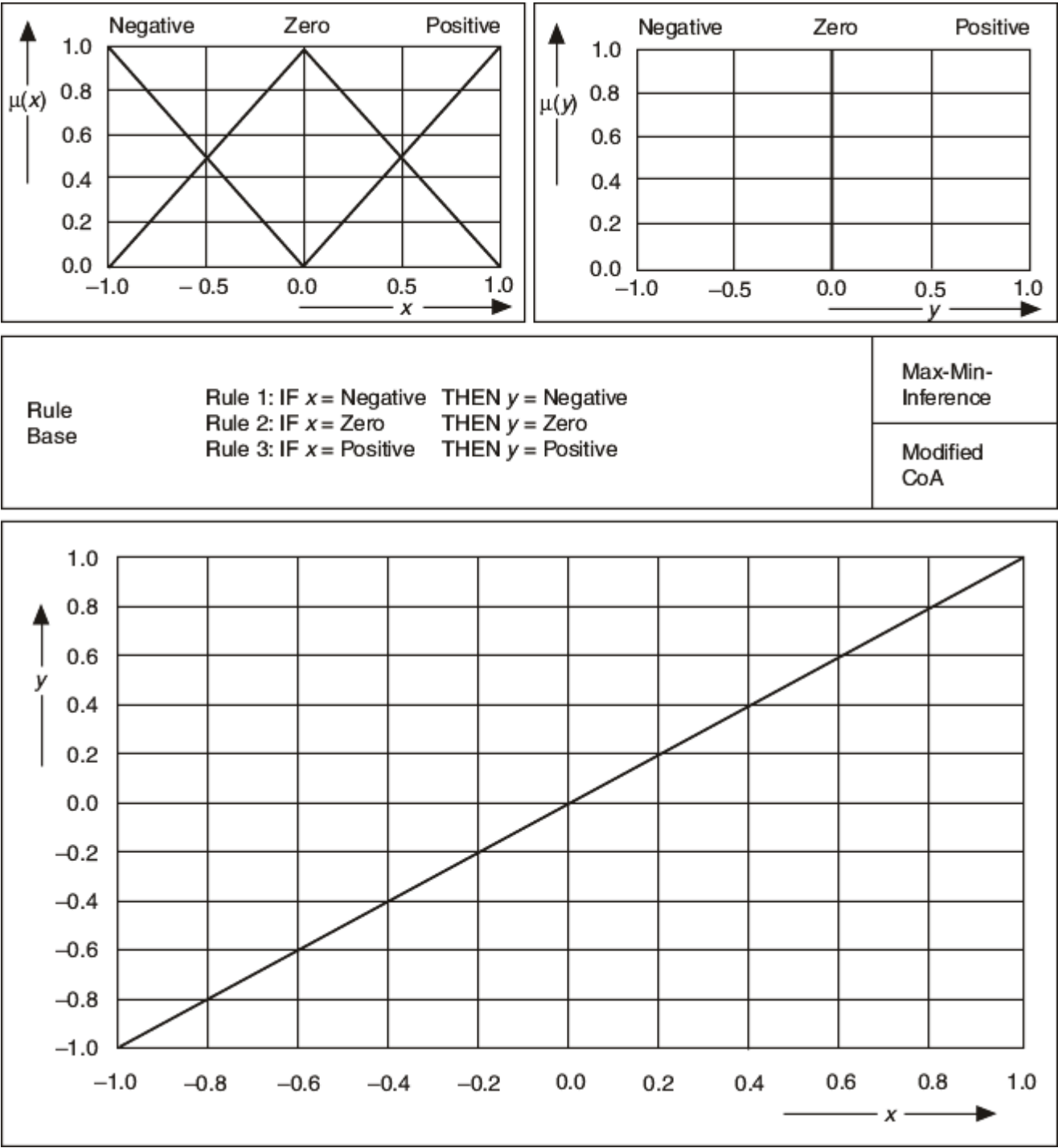


The PID Setpoint Profile VI returns a single **setpoint** value determined from the current elapsed time. The first call to the VI initializes the current time in the setpoint profile to 0. On subsequent calls, the VI determines the current time from the previous time and the **dt** input value. If you reinitialize the current time to 0 by passing a value of TRUE to the **reinitialize?** input, you can repeat the specified setpoint profile.

Singletons as Output Terms with Fully Overlapping Input Terms

The simplest way to obtain a linear *controller characteristic* is to use singletons as consequent terms with entirely overlapping input terms. Singletons are normalized rectangular *membership functions* with an infinitely small width. Using singleton membership functions for consequent terms makes the *Center of Area (CoA)*

defuzzification method identical to the *Center of Maximum (CoM)* method. The following image shows the controller for the CoA method using singleton membership functions.



The controller characteristic remains relatively unchanged when you leave the input terms entirely overlapped to vary the overlapping degree of the membership functions for the consequent terms, especially if all the consequent terms are equal in width. Then only the typical values of the consequent terms are significant.

Therefore, in most closed-loop control applications, you can use singleton membership functions to model the output terms sufficiently rather than using triangular or other membership function types.

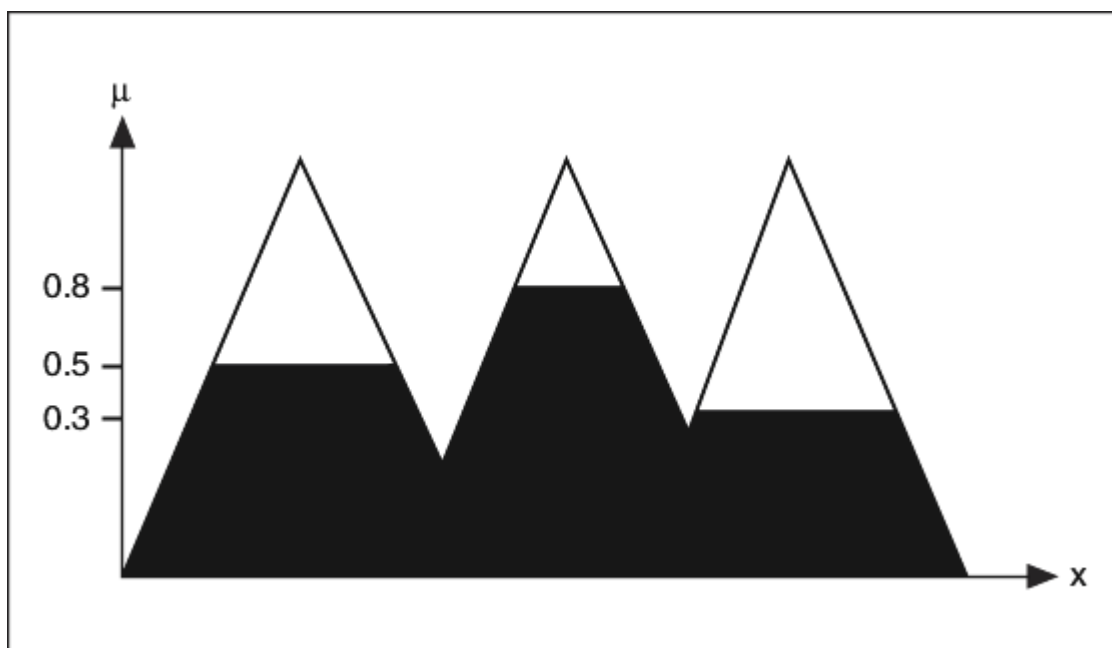
Specifying a Consequent Implication Method

A fuzzy controller uses a consequent implication method to scale the *membership functions* of each output *linguistic variable* based on the corresponding rule weight before performing defuzzification.

With PID and Fuzzy Logic VIs, you can use either the Minimum or Product consequent implication method.

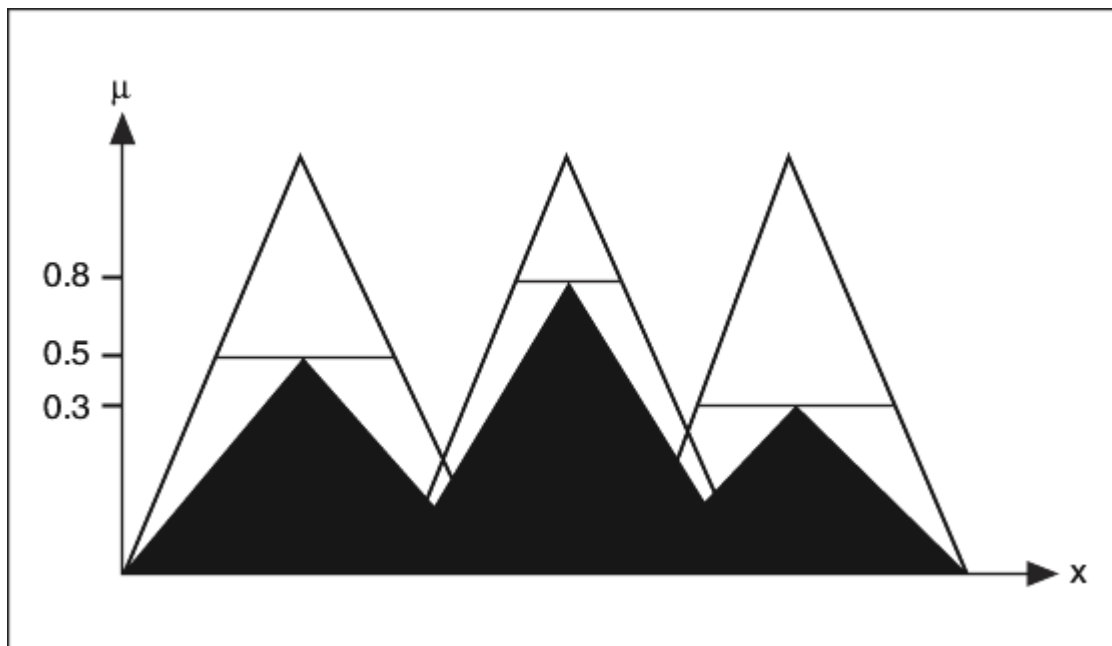
Minimum Consequent Implication Method

In the Minimum implication method, the fuzzy logic controller truncates the output membership functions at the value of the corresponding rule weights. For example, if an output linguistic variable has three membership functions with rule weights 0.5, 0.8, and 0.3, respectively, the scaled membership functions might appear similar to the following image.



Product Consequent Implication Method

In the Product implication method, the fuzzy logic controller scales the output membership functions at the value of the corresponding rule weights. For example, if an output linguistic variable has three membership functions with rule weights 0.5, 0.8, and 0.3, respectively, the scaled membership functions might appear similar to the following image.



Specifying an Antecedent Connective

When *creating a rule base*, if a rule has more than one antecedent, you must specify an antecedent connective to determine how to calculate the truth value of the aggregated rule antecedent.

Because *linguistic variables* can have partial degrees of membership within *linguistic terms*, you cannot use Boolean operators from conventional dual logic as antecedent connectives. PID and Fuzzy Logic VIs use the following antecedent connectives instead.

AND (Minimum)

$$\mu A * B = \min (\mu A, \mu B)$$

AND (Product)

$$\mu A * B = (\mu A, \mu B)$$

OR (Maximum)

$$\mu A + B = \max (\mu A, \mu B)$$

OR (Probabilistic)

$$A + B = (A + B - AB)$$

Notice that these definitions agree with the logical operators used in Boolean logic. A truth table uses conventional operators to yield equivalent results.

The AND (Minimum) antecedent connective specifies to use the smallest degree of membership of the antecedents as the truth value of the aggregated rule antecedent, while the AND (Product) specifies to use the product of the degrees of membership of the antecedents. The OR (Maximum) antecedent connective specifies to use the largest degree of membership of the antecedents. The OR (Probabilistic) antecedent connective specifies to use the probabilistic sum of the degrees of membership of the antecedents.

Antecedent Connectives for a Vehicle Maneuvering Fuzzy System

Assume the following rules are invoked for the input values of a vehicle maneuvering fuzzy system.

(1)	IF Vehicle Position x is Center (degree of membership = 0.8)	AND (Minimum)	Vehicle Orientation β is Left Up (degree of membership = 1.0) = 0.8
THEN Steering Angle ϕ is Negative Small			
(2)	IF Vehicle Position x is Right Center (degree of membership = 0.1)	AND (Minimum)	Vehicle Orientation β is Left Up (degree of membership = 1.0) = 0.1
THEN Steering Angle ϕ is Negative Medium			

Notice that each rule uses the AND (Minimum) antecedent connective. In Rule 1, Vehicle Position x has a degree of membership of 0.8 within the linguistic term Center and Vehicle Orientation β has a degree of membership of 1.0 within the linguistic term Left Up. Because the antecedent connective is AND (Minimum), the fuzzy controller for this fuzzy system uses the smallest degree of membership of the antecedents, or 0.8, as the truth value of the aggregated rule antecedent. Similarly, the smallest degree of membership of the antecedents in Rule 2 is 0.1. Therefore, the fuzzy controller uses 0.1 as the truth value of the aggregated rule antecedent. The truth value of the aggregated rule antecedent is equivalent to the degree of truth of the rule.

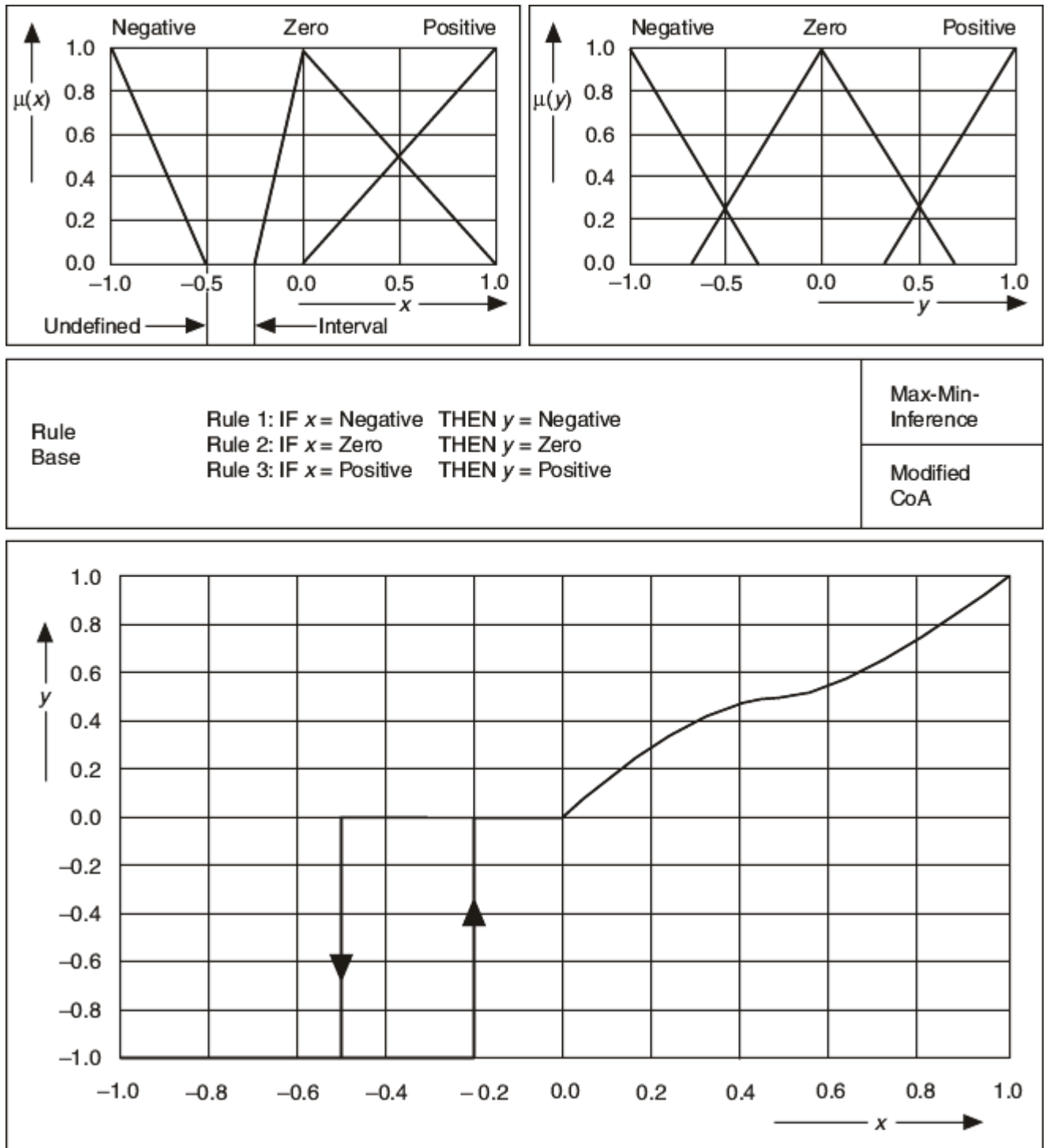
If these two rules are the only rules invoked for a given set of input values, the other linguistic terms for the Steering Angle ϕ output linguistic variable have a truth value of 0. The following list describes the final truth values for each of the linguistic terms.

Negative Large Negative Medium Negative Small Zero Positive Small Positive Medium Positive Large	to a degree of to a degree of to a degree of to a degree of to a degree of to a degree of to a degree of	0.0 0.1 0.8 0.0 0.0 0.0 0.0

If a rule has more than one consequent, the fuzzy logic controller must evaluate the aggregated rule consequent. When you implement a fuzzy controller with PID and Fuzzy Logic VIs, the fuzzy controller always considers only the consequent that has the largest degree of membership.

Undefined Input Term Intervals

If the **rule base** has undefined intervals within input and output linguistic terms, or if the rule base is incomplete, you must specify the output of the fuzzy controller. If no rule is available for a certain situation, the output value remains undefined. One way to avoid this problem is to leave the current output value unchanged until the controller encounters a situation that is covered by the rules. The following image shows the resulting effect on the controller characteristic.



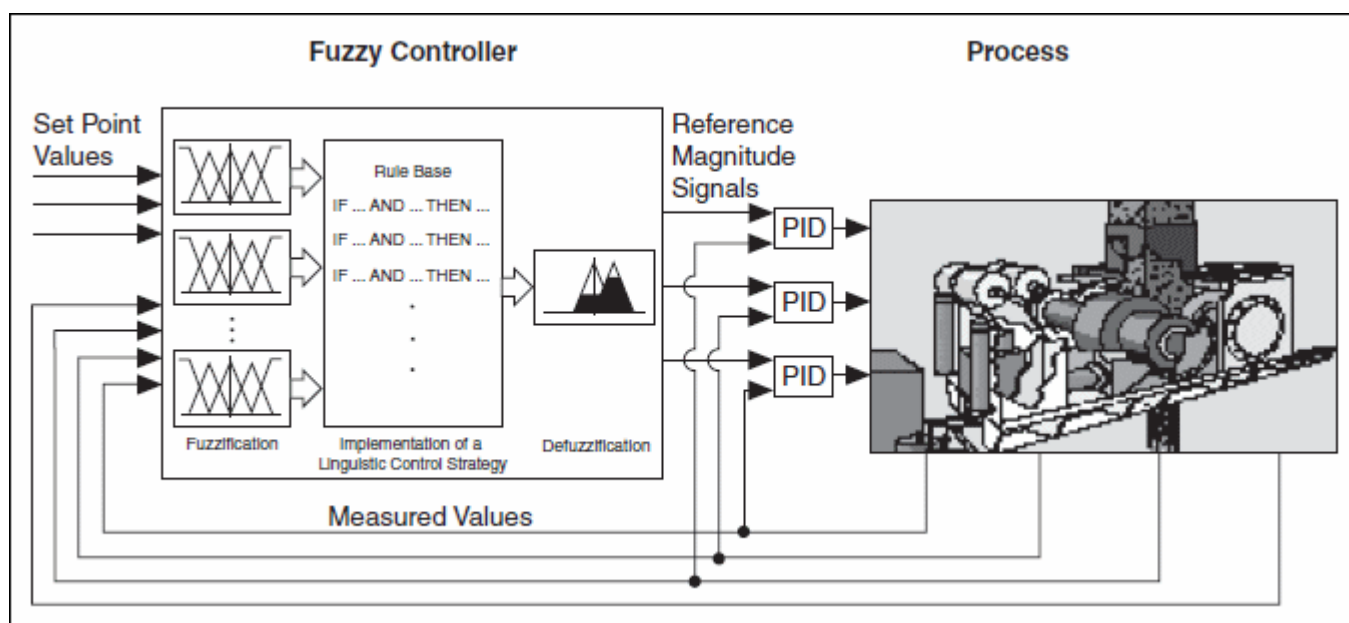
If you use an old output value as a default value, undefined intervals or incomplete rule bases can lead to hysteretic effects on the [controller characteristic](#).

You can use non-overlapping, rectangular-shaped consequent terms to obtain an exact linear [controller characteristic](#) for a single-input controller. In this case, both the area and momentum vary linearly with the degree of truth, and overlapping regions of the output linguistic terms do not cause any distortion.

Using Fuzzy Controllers with PID Controllers

The chemical industries and process technologies often use fuzzy controllers with underlying PID control loops. The fuzzy controllers these industries use are PID fuzzy controllers that control single-process parameters. Usually, people supervise the operating point of the entire process.

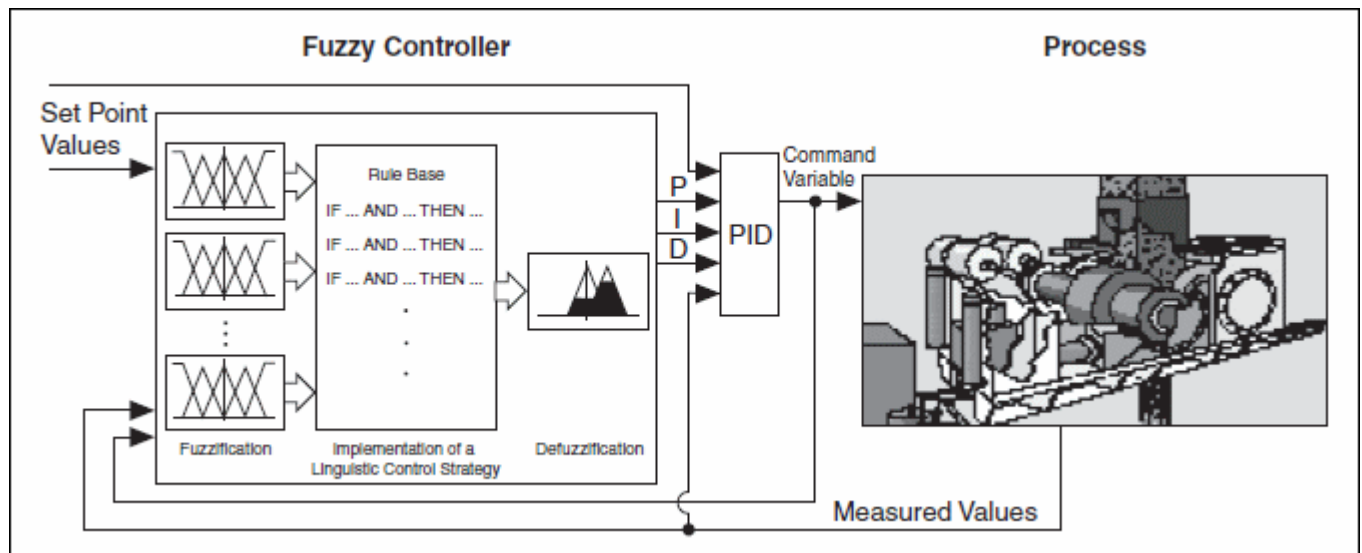
The following image shows the controller structure of the fuzzy controller with underlying PID control loops.



For automatic operation of such multi-variable control problems, you must build a model-based controller. However, for most applications, either the process is too complex to model adequately, or the mathematical modeling task requires too much time.

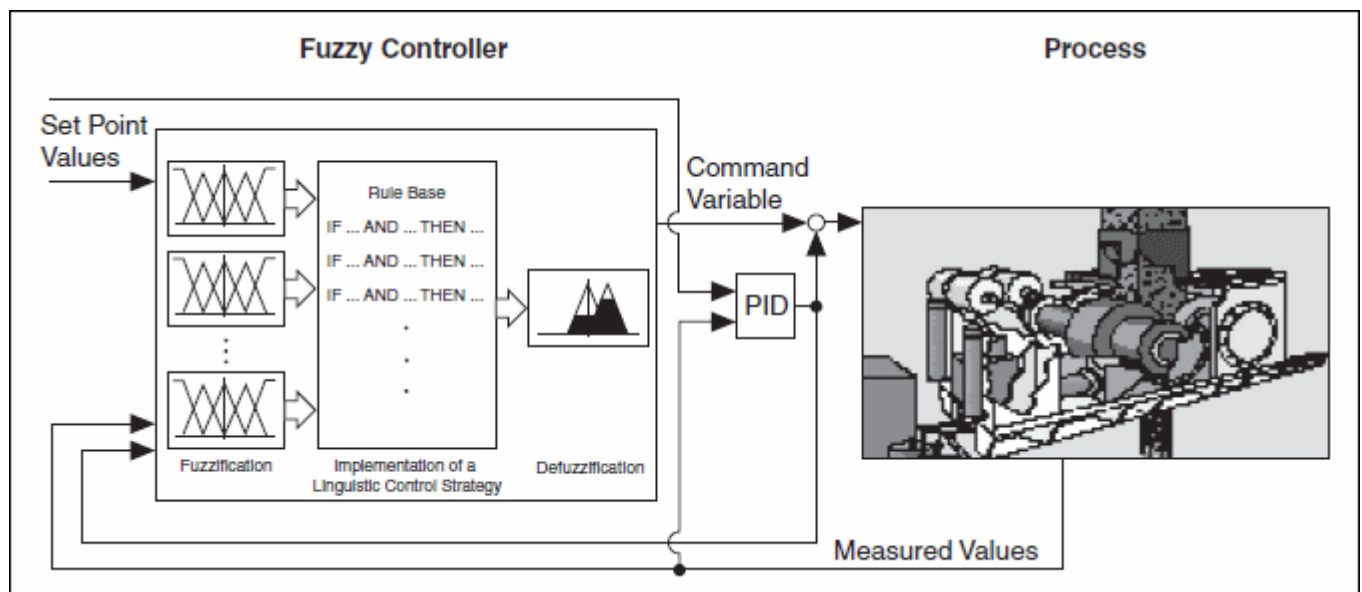
Using a Fuzzy Controller to Tune PID Controller Parameters

The following image shows how to use a fuzzy controller to tune the parameters of a conventional PID controller automatically. This fuzzy controller constantly interprets the process reaction and calculates the optimal P, I, and D gains. You can apply this control structure to processes that change their characteristics over time.



Using a Fuzzy Controller to Correct PID Controller Output

In the following image, both the fuzzy controller and the PID controller work in parallel. The process adds the output signals from both controllers, but the output signal from the fuzzy controller is zero under normal operating conditions. The PID controller output leads the process. The fuzzy controller intervenes only when it detects abnormal operating conditions, such as strong disturbances.



PID Autotuning

Implementing Autotuning with the Autotuning Wizard

To use the Autotuning Wizard to improve your controller performance, you must first create your control application and determine PID parameters that produce stable control of the system. You can develop the control application using either the PID VI, the PID Gain Schedule VI, or the PID Autotuning VI. Because the PID Autotuning VI has inputs and outputs consistent with the other PID VIs, you can replace many PID VIs with it.



NOTE

The PID Autotuning VI cannot directly replace the PID Advanced VI or the more advanced autotuning VIs, such as PID Advanced Autotuning or PID Online Autotuning. These VIs provide advanced functionality that the PID Autotuning VI does not support.

Autotuning-Specific Inputs

The PID Autotuning VI has several inputs and outputs specific to the autotuning procedure, including the following inputs:

- **autotuning parameters** —A cluster of parameters that the PID Autotuning VI uses for the autotuning process. The Autotuning Wizard allows you to specify all of these parameters manually, so wiring this input is optional.
- **autotune?** —Specifies whether to invoke the Autotuning Wizard and begin autotuning. Wire a Boolean control, such as a button, to this input so a user can press the button to open the Autotuning Wizard. The Autotuning Wizard steps the user through the autotuning process.



NOTE

*Set the Boolean control mechanical action to **Latch When Released*** so the Autotuning Wizard does not open repeatedly when the user presses the button.

Autotuning-Specific Outputs

The following outputs are specific to the autotuning procedure:

- **tuning completed?** —Indicates when the autotuning process is complete. You can use this output to determine when to update the **PID gains** values.
- **PID gains out** —Returns the tuned PID gains. Before autotuning completes, **PID gains out** returns the same values as the **PID gains** input. **PID gains out** updates only when autotuning completes and **tuning completed?** returns TRUE.

Implementing PID Controllers with the PID VIs

The PID VIs are all variations of the basic PID VI, and each variation provides additional functionality as described in the following table. You can use these VIs interchangeably because they use consistent inputs and outputs where possible. Refer to the help topic for the VI you want to use for specific information about the VI.

Use Case	VI Name
- Simple systems, or when an efficient algorithm is required. - Implements the basic PID algorithm	PID VI
- Complex systems that require options for manual or automatic operation, setpoint weighting, filtering of derivative action, and so on. - Implements the advanced PID algorithm and provides extra input parameters to expand on the functionality of the PID VI.	PID Advanced VI
Complex systems that require the same advanced options as the PID Advanced VI and the ability to autotune parameters.	PID Advanced Autotuning VI
Systems that contain significant dead time.	PID Autotuning (Temperature) VI
Systems that require lead/lag function.	PID Lead-Lag VI



RELATED INFORMATION

[Implementing the PID Algorithm with the PID VIs](#) on page 1764

[Advanced PID Algorithm](#) on page 1770

[Autotuning PID Controllers \(PID and Fuzzy Logic\)](#) on page 1774

[Lead/Lag](#) on page 1768

[Fault Protection](#) on page 1734

[Multi-Loop Control](#) on page 1729

[Setpoint Profile Generation](#) on page 1752

[Gain Scheduling](#) on page 1742

[Control Output Rate Limiting](#) on page 1788

[Lead/Lag](#) on page 1768

[Converting Between Percentage of Full Scale and Engineering Units](#) on page 1788

[Filtering Control Inputs](#) on page 1735

Implementing the PID Algorithm with the PID VIs

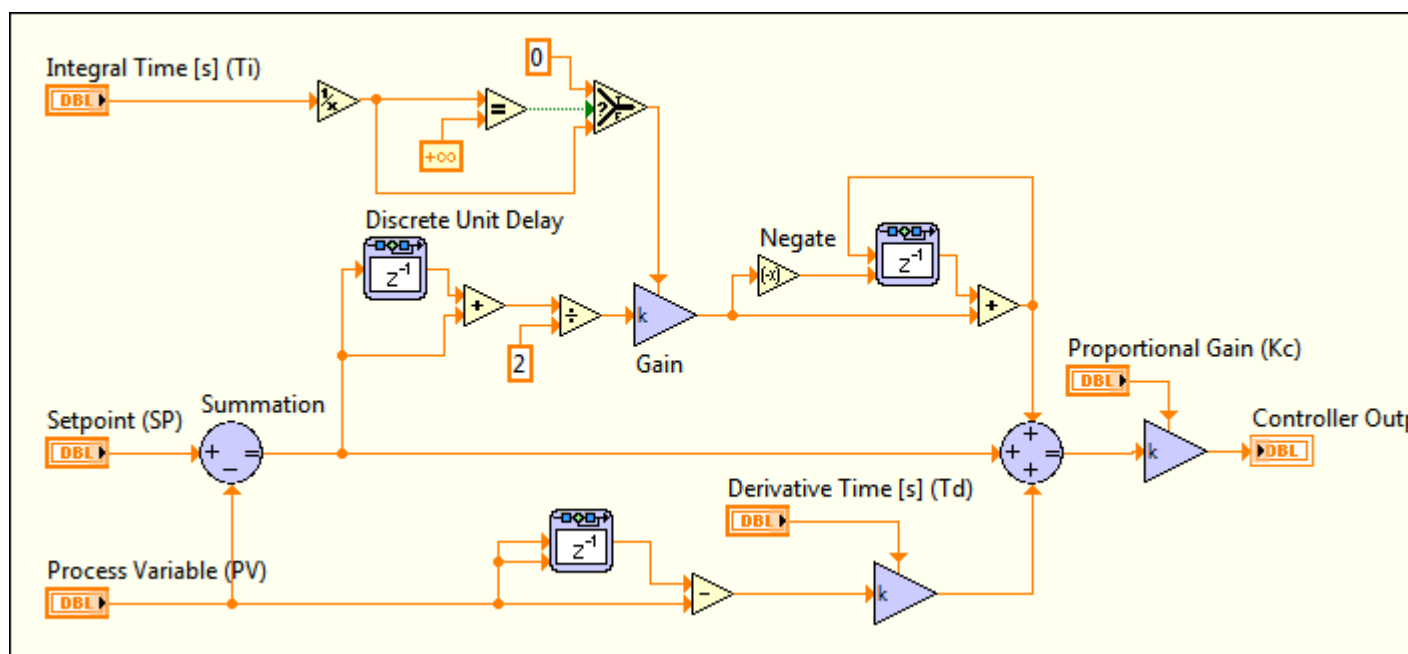
This following sections describe how the basic PID VIs (not including PID Advanced, PID Advanced Autotuning, and other VIs with advanced options) implement the [PID algorithm](#), and the assumptions and transformations necessary to implement the PID controller. To implement a PID controller, LabVIEW requires the algorithm to sample the input signals and discretize the integral and derivative action.



NOTE

The following formulas apply to most VIs on the PID palette. The PID Advanced and PID Advanced Autotuning VIs use [extended formulas](#) with more advanced optional features.

The following simulation diagram represents the PID implementation provided by the basic PID VIs:



NOTE

Constructing a simulation diagram like the one in the previous image requires the LabVIEW Control Design and Simulation Module. However, the PID VIs implement PID controllers for you. You do not need the Control Design and Simulation Module to build PID controllers. Refer to the National Instruments website at ni.com for more information about the Control Design and Simulation Module.

Error Calculation

The following formula represents the current error used in calculating proportional, integral, and derivative action:

$$e(k) = (SP(k) - PV(k))$$

where	k is the index of the sampled signal at time k * t
e(k) is the current error	
SP is the setpoint	
PV is the process variable	

Proportional Action

Proportional action is the controller gain times the error, as shown in the following formula:

$$u_p(k) = (K_c * e(k))$$

where	K _c is the controller gain
e(k) is the current error	

The PID VIs express the proportional component in terms of controller gain. The relationship between controller gain K_c and proportional band PB is K_c = 100 / PB.

Integral Action (Trapezoidal Integration)

There are several options for discretizing integral action, such as forward difference, backward difference, and trapezoidal approximation, which is also known as Tustin or Bilinear transformation. The PID VIs use trapezoidal integration to avoid sharp changes in integral action when there is a sudden change in PV or SP, as represented by the following formula:

$$u_I(k) = u_I(k-1) + \frac{K_c}{T_i} \left(\frac{e(k) + e(k-1)}{2} \right) \Delta T$$

where	ΔT is the sampling time of the controller

Derivative Action

Abrupt changes in SP can generate bumps to the output of the controller as a result of applying derivative action to the error e. These bumps are referred to as derivative kick. To avoid derivative kick, you can apply derivative action to the PV only, and not to the error e. The following formula represents the derivative action implemented by the PID VIs, which avoids derivative kick:

$$u_D(k) = -K_c \frac{T_d}{\Delta_t} (PV(k) - PV(k-1))$$

Controller Output

Controller output is the summation of the proportional, integral, and derivative action, as shown in the following formula:

$$u(k) = u_P(k) + u_I(k) + u_D(k)$$

Output Limiting and the Anti-Windup Algorithm

The actual controller output is limited to the range specified for control output.

if	$u(k) \geq u_{max}$	then	$u(k) = u_{max}$

and

if	$u(k) \leq u_{min}$	then	$u(k) = u_{min}$

The PID VIs use an integral sum correction algorithm that facilitates anti-windup. Windup occurs at the upper limit of the controller output. When the error e decreases, the controller output decreases, moving out of the windup area. The integral sum correction algorithm prevents abrupt controller output changes when you switch from manual to automatic mode or change any other parameters. The integral sum correction works as follows:

if	$ u_P(k) + u_I(k) > limit $	then	$u_I(k) = limit - u_P(k)$

The previous statement demonstrates that the integral sum correction does not take into account the derivative action on value changes.

Default PID Values and Gain Changes

The default ranges for the parameters SP, PV, and output range correspond to percentage values. However, you can use actual engineering units. Adjust corresponding ranges accordingly. The parameters T_i and T_d are specified in minutes.

You can call the PID VIs from inside a While Loop with a fixed cycle time. All the PID VIs are reentrant. Multiple calls from high-level VIs use separate and distinct data. Also, the PID VI has a multi-channel mode where you provide an array that represents an input for each channel.

PID gains can change at any time. When a change is detected, the PID algorithm tries to perform a bumpless transition by adjusting the integral action to keep the output constant with the new parameters. This avoids output changes when you try to modify the PID parameters, and is useful in gain scheduling.

Gain Scheduling

Gain scheduling refers to a system where you change controller parameters based on measured operating conditions. For example, the scheduling variable can be the setpoint, the process variable, a controller output, or an external signal. For historical reasons, the term gain scheduling is used even if other parameters such as derivative time or integral time change. Gain scheduling effectively controls a system with dynamics that change with operating conditions.

With the PID Gain Schedule VI, you can define unlimited sets of PID parameters for gain scheduling. For each schedule, you can use **autotuning** to update the PID parameters.

Internal Model Control Autotuning Method

Internal model control (IMC) is a control technique that uses an internal model to try to estimate the system response. PID and Fuzzy Logic VIs use IMC when it cannot successfully implement the **Ziegler-Nichols method**, such as when a model has no delay. You also can use the PID Autotuning Design VI to directly implement this autotuning method.

This method assumes a first-order model with delay. If your system is not a first-order model, these techniques try to approximate one during tuning.

The following table shows the tuning formula for the internal model control technique, where K is the model gain, τ is the plant time constant, τ_c is the desired time constant, and θ is the plant delay.

Controller	Model	K_c	T_i	T_d
PI (delay-free)	$\frac{K}{\tau s + 1}$	$\frac{1}{K} \frac{\tau}{\tau_c}$	τ	—
PI + Plant (with delay)	$\frac{K_e^{-\theta s}}{\tau s + 1}$	$\frac{1}{K} \frac{\tau}{\tau_c + \theta}$	τ	—
PID	$\frac{K_e^{-\theta s}}{\tau s + 1}$	$\frac{1}{K} \frac{\tau + \frac{\theta}{2}}{\tau_c + \frac{\theta}{2}}$	$\tau + \frac{\theta}{2}$	$\frac{\tau\theta}{2\tau + \theta}$

This method is based on algorithms and techniques described in reference material by Seborg, Dale E., Thomas F. Edgar, and Duncan A. Mellichamp. For more information about this method, refer to the reference material by those authors.

Lead/Lag

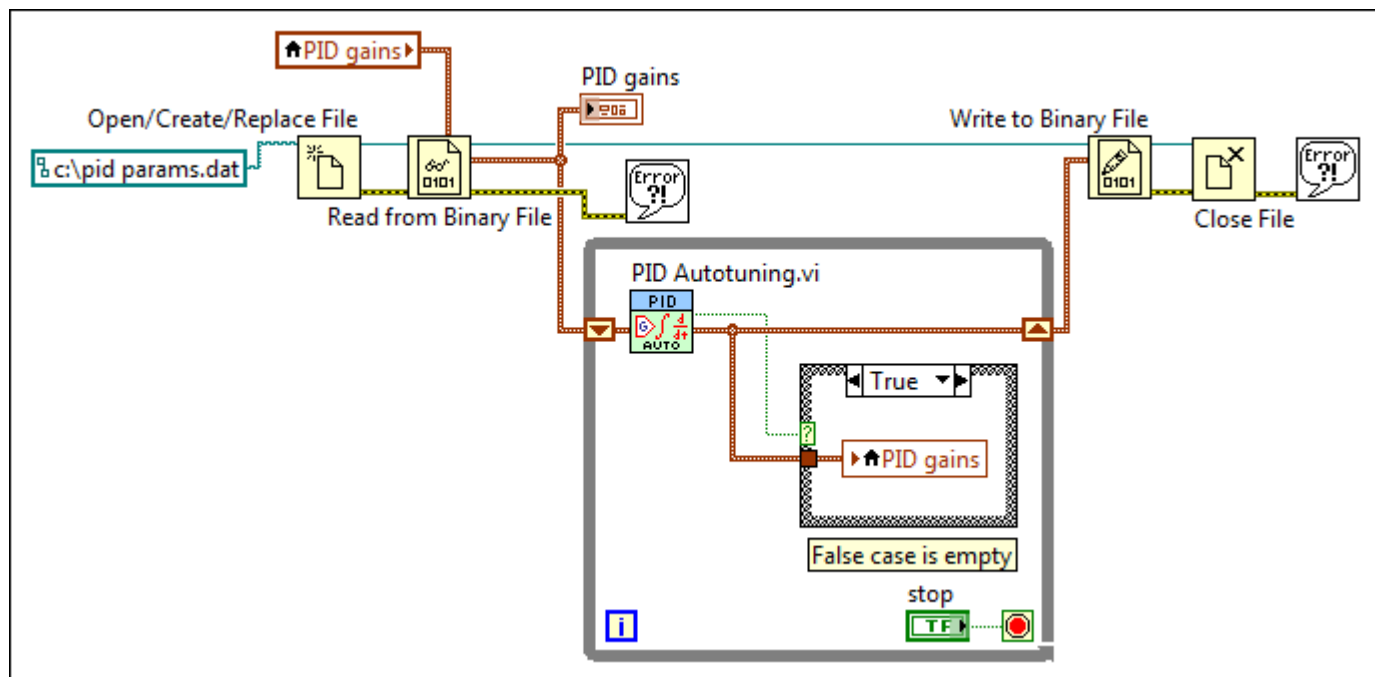
You can use the PID Lead-Lag VI to implement a PID controller with a lead/lag function. The VI uses a positional algorithm that approximates a true exponential lead/lag. Feedforward control schemes often use this kind of algorithm as a dynamic compensator.

The **output** value on the first call to the VI is the same as the **input** value.

Logging Parameter Values After Autotuning

The PID autotuning VIs, including PID Autotuning, PID Advanced Autotuning, and PID Online Autotuning, update the parameters of a PID controller and return updated PID gains values for the next iteration of the control application. To ensure that the correct PID gains values are available each time you run your control application, you must either save the VI you are using to implement the application each time it runs, or use a datalog file to save the PID gains.

The following block diagram shows a VI that uses the File I/O VIs and functions to create a datalog file and save the latest value of the **PID gains out** output of the PID Autotuning VI to the file.



Before the control loop begins, the Open/Create/Replace File function opens a datalog file, and the Read from Binary File function reads the file to obtain the **PID gains** parameters. When the autotuning procedure runs, a local variable updates the **PID gains** control. After the control loop is complete, the Write to Binary File function writes the current **PID gains**

cluster to the datalog file and saves the file. This ensures that each time the control application runs, the PID Autotuning VI uses updated parameters.



RELATED INFORMATION

[Autotuning PID Controllers \(PID and Fuzzy Logic\)](#) on page 1774

[Updating PID Parameters After Autotuning](#) on page 1777

Online Autotuning Techniques

To tune PID parameters of a system, some PID VIs allow you to select the technique to apply to the system. You can choose from two techniques that implement a step response-based measurement and two techniques that implement a frequency response-based measurement.

On the PID Advanced Autotuning VI, the **autotuning parameters** input allows you to configure the technique to use. With the PID Online Autotuning polymorphic VI, select the polymorphic instance that corresponds to a tuning technique you want to use. Other PID VIs control the tuning technique for you.

Step Response Techniques

The following techniques are based on step response measurement:

- **Step Open Loop** —Performs an open-loop step test. This technique applies a step to the input and waits until it reaches steady-state, when the process variable does not change. This technique assumes that you can model any process as a first-order lag and a pure deadtime system.
- **Step Closed Loop** —Performs a closed-loop step test similar to the Step Open Loop technique.

If you do not know the characteristics of the model, such as initial PID gains, you can first perform Step Open Loop tuning on the model to obtain the parameters. Wait until the input reaches steady-state. This process is also known as commissioning. Next, perform Step Closed Loop tuning on the model using the parameters computed by the Step Open Loop tuning process as initial parameters. Both techniques assume a first-order model with delay. If your system is not a first-order model, these techniques try to approximate one during tuning.

Frequency Response Techniques

The following techniques are based on frequency response measurement

- **PID Relay** —Uses the setpoint relay experiment to determine the information needed to tune the controller. The Autotuning Wizard uses this technique.
- **Relay Feedback** —Uses an on-off relay in a feedback loop to determine ultimate gain and ultimate frequency. This technique is a variation on a closed-loop step test, but

is more efficient for systems with a large time constant. This method also incorporates hysteresis to avoid noise in the system.

Related Information

Step Open Loop Autotuning Technique

Step Closed Loop Autotuning Technique

PID Relay Autotuning Technique

Relay Feedback Autotuning Technique



RELATED INFORMATION

[Autotuning PID Controllers \(PID and Fuzzy Logic\)](#) on page 1774

[Step Open Loop Autotuning Technique](#) on page 1777

[Step Closed Loop Autotuning Technique](#) on page 1777

[PID Relay Autotuning Technique](#) on page 1748

[Implementing Autotuning with the Autotuning Wizard](#) on page 1762

[Relay Feedback Autotuning Technique](#) on page 1775

[Step Open Loop Autotuning Technique](#) on page 1777

[Step Closed Loop Autotuning Technique](#) on page 1777

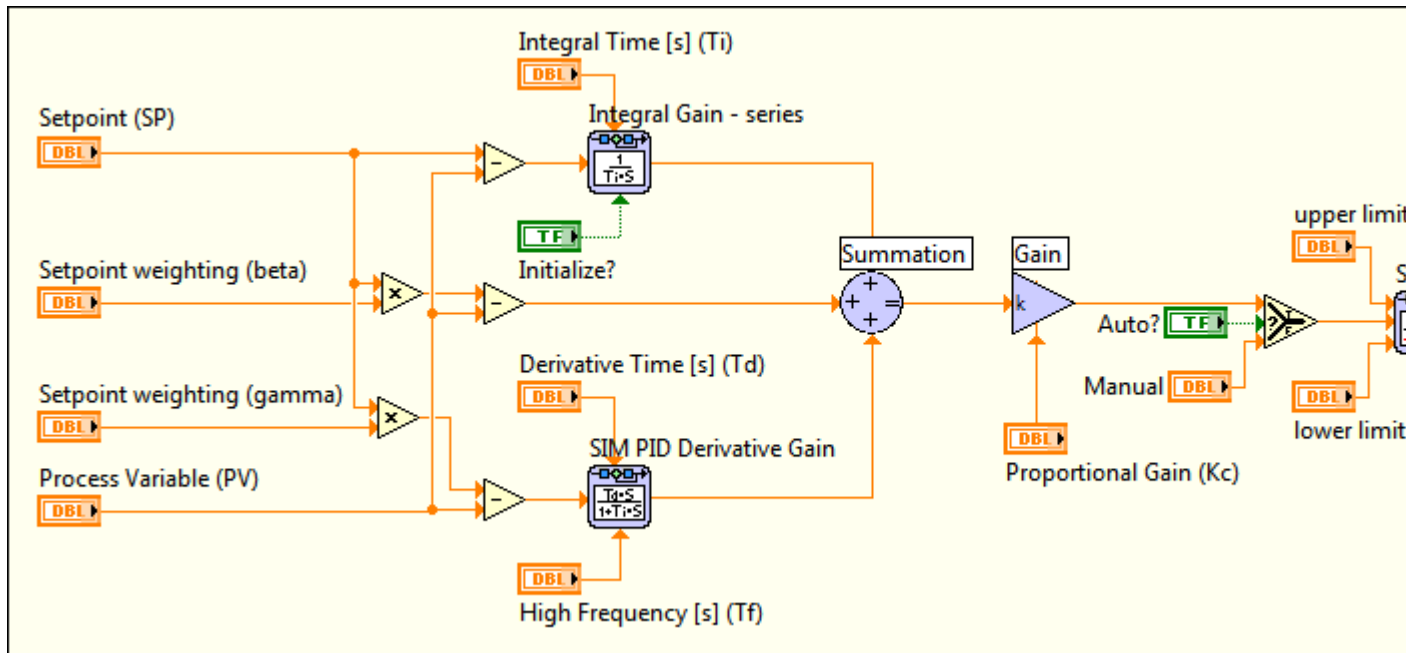
[PID Relay Autotuning Technique](#) on page 1748

[Relay Feedback Autotuning Technique](#) on page 1775

Advanced PID Algorithm

The PID Advanced VI uses the same algorithm and assumptions as the [PID VIs algorithm](#), but adds the functionality described in the following sections.

The following simulation diagram represents the PID implementation provided by the PID Advanced VI:



NOTE

Constructing a simulation diagram like the one in the previous image requires the LabVIEW Control Design and Simulation Module. However, the PID VIs implement PID controllers for you. You do not need the Control Design and Simulation Module to build PID controllers. Refer to the National Instruments website at ni.com for more information about the Control Design and Simulation Module.

Manual/Automatic Operation

In some situations, you might need to switch off the PID controller and operate the system in open-loop, or manual, mode. The PID Advanced VI uses a method similar to the [anti-windup algorithm](#) to ensure the transition between manual and automatic modes does not cause any bump to the output of the controller.

Setpoint Weighting for Proportional Action and Derivative Action

Setpoint weighting refers to corrections applied to the error value of the controller. The **beta** input (B), which can range between 0 and 1, is applied to the proportional action as:

$$eb(k) = (\beta SP(k) - PV(k))$$

$$u_P(k) = K_C * eb(k)$$

where

e is the error

SP is the setpoint

PV is the process variable

K_C is controller gain

You can use the **beta** input to reduce the overshoot of setpoint change and to allow increasing the gain of the controller to improve the disturbance rejection of the controller.

The **gamma** (γ) input, which also can range between 0 and 1, is applied to the error and to the derivative as:

$$ed(k) = (\gamma SP(k) - PV(k))$$
$$u_D(k) = K_c \frac{T_d}{\Delta T} (ed(k) - ed(k-1))$$

Filter for Derivative Action

Error-Squared PID Implemented by the Linearity Factor (L)

The *derivative action* implemented by the PID Advanced VI is based on the backward method for derivative action. However, depending on the values of the derivative action, proportional gain, and sampling time, the controller could generate responses with fast oscillation, especially when performing autotuning with PID. To prevent this issue, this VI provides an **alpha** parameter that implements an additional filter for the derivative action D. This filter reduces the bandwidth content as defined by:

$$u_D(s) = -K_c \frac{T_d s}{1 + \alpha T_d s}$$

where T_d is the derivative time in minutes, also called the rate time.

The **linearity** input introduces linearity change on the gain, as shown in the following equations:

$$e(k) = (SP - PV_f) \left(L + (1 - L) * \frac{|SP - PV_f|}{SP_{range}} \right)$$

The error for calculating proportional action with two degree-of-freedom control is shown in the following formula:

$$eb(k) = (\beta * SP - PV_f) \left(L + (1 - L) * \frac{|\beta SP - PV_f|}{SP_{range}} \right)$$

where	SP_{range} is the range of the setpoint
β is the setpoint factor for the two degree-of-freedom PID algorithm described by the proportional action formula	
L is the linearity factor that produces a nonlinear gain term in which the controller gain increases with the magnitude of the error	

If L is 1, the controller is linear. A value of 0.1 makes the minimum gain of the controller 10% K_C . Use of a nonlinear gain term is referred to as an error-squared PID algorithm.

Error-Squared PID Implemented by the Linearity Factor (L)

The **linearity** input introduces linearity change on the gain, as shown in the following equations:

$$e(k) = (SP - PV_f) \left(L + (1 - L) * \frac{|SP - PV_f|}{SP_{range}} \right)$$

The error for calculating proportional action with two degree-of-freedom control is shown in the following formula:

$$eb(k) = (\beta * SP - PV_f) \left(L + (1 - L) * \frac{|\beta SP - PV_f|}{SP_{range}} \right)$$

where

SP_{range} is the range of the setpoint

β is the setpoint factor for the two degree-of-freedom PID algorithm described by the proportional action formula

L is the linearity factor that produces a nonlinear gain term in which the controller gain increases with the magnitude of the error

If L is 1, the controller is linear. A value of 0.1 makes the minimum gain of the controller 10% K_C . Use of a nonlinear gain term is referred to as an error-squared PID algorithm.

Autotuning Methods for Testing PID Parameters

Autotuning automatically tests different parameter values for a PID controller to determine the best response. A variety of methods exist for testing these parameters. PID and Fuzzy Logic VIs support the following methods for PID autotuning:

- Ziegler-Nichols
- Internal Model Control
- Cohen-Coon
- Chien-Hrones-Reswick

The Autotuning Wizard and most of the autotuning PID VIs use the Ziegler-Nichols method or, if Ziegler-Nichols is not possible, the Internal Model Control method. Only the PID

Autotuning Design VI can implement the Cohen-Coon and Chien-Hrones-Reswick tuning methods.



RELATED INFORMATION

- [Ziegler-Nichols Autotuning Method](#) on page 1779
- [Internal Model Control Autotuning Method](#) on page 1767
- [Cohen-Coon Autotuning Method](#) on page 1786
- [Chien-Hrones-Reswick Autotuning Method](#) on page 1784

Autotuning PID Controllers (PID and Fuzzy Logic)

Often, many PID controllers are poorly tuned. As a result, some controllers are too aggressive and some controllers are too sluggish. Tuning a controller involves adjusting the parameters of the system to get the desired, or optimal, response. However, manually tuning PID controllers can be difficult when you do not know the process dynamics or disturbances. In this case, you can use autotuning to automatically test different parameter values and determine the best response.

Choose from the following VIs that integrate autotuning based on your specific application needs and your knowledge of tuning theory.

Use Case	VI Name
- Autotuning that runs with minimal user input or knowledge of tuning algorithms. - Single VI implements a controller for a system and allows tuning.	PID Advanced Autotuning VI
- Autotuning that runs with minimal user input or knowledge of tuning algorithms. - Single VI implements a controller for a system with significant deadtime and allows tuning. How-To: Programmatically Implementing and Autotuning a PID Controller	PID Autotuning (Temperature) VI
- Controlling the autotuning technique used to identify the system. Other PID VIs automatically determine the technique. - Tuning system parameters, then passing the output parameters to the PID Advanced VI to implement that controller. How-To: Programmatically Autotuning a Previously Implemented PID Controller	PID Online Autotuning VI
- Generating PID parameters for a system given an input/output signal. This technique is useful when you have insufficient information about the system and you want to develop initial parameters based on system response. - Provides access to multiple methods for determining PID parameters. Other PID VIs automatically determine the method.	PID Autotuning Design VI

Use Case	VI Name
- Performing interactive autotuning in a dialog box that you use to configure the process. - Not supported on RT or FPGA hardware targets. How-To: Implementing Autotuning with the Autotuning Wizard	PID Autotuning VI
- PID applications with high-speed control and/or high channel count on an FPGA target. - Implements a fixed-point PID algorithm.	PID (FPGA) VI



RELATED INFORMATION

[Online Autotuning Techniques](#) on page 1769

[Programmatically Autotuning a Previously Implemented PID Controller](#) on page 1750

[Implementing Autotuning with the Autotuning Wizard](#) on page 1762

[Updating PID Parameters After Autotuning](#) on page 1777

[Logging Parameter Values After Autotuning](#) on page 1768

[Online Autotuning Techniques](#) on page 1769

[Autotuning Methods for Testing PID Parameters](#) on page 1773

Relay Feedback Autotuning Technique

The Relay Feedback autotuning technique, which is used by the Relay Feedback instance of the PID Online Autotuning VI, uses an on-off relay in a feedback loop to determine ultimate gain and ultimate frequency. This technique is a variation on a closed-loop (ultimate gain) test, but is more efficient for systems with a large time constant. This method also incorporates hysteresis to avoid noise in the system.

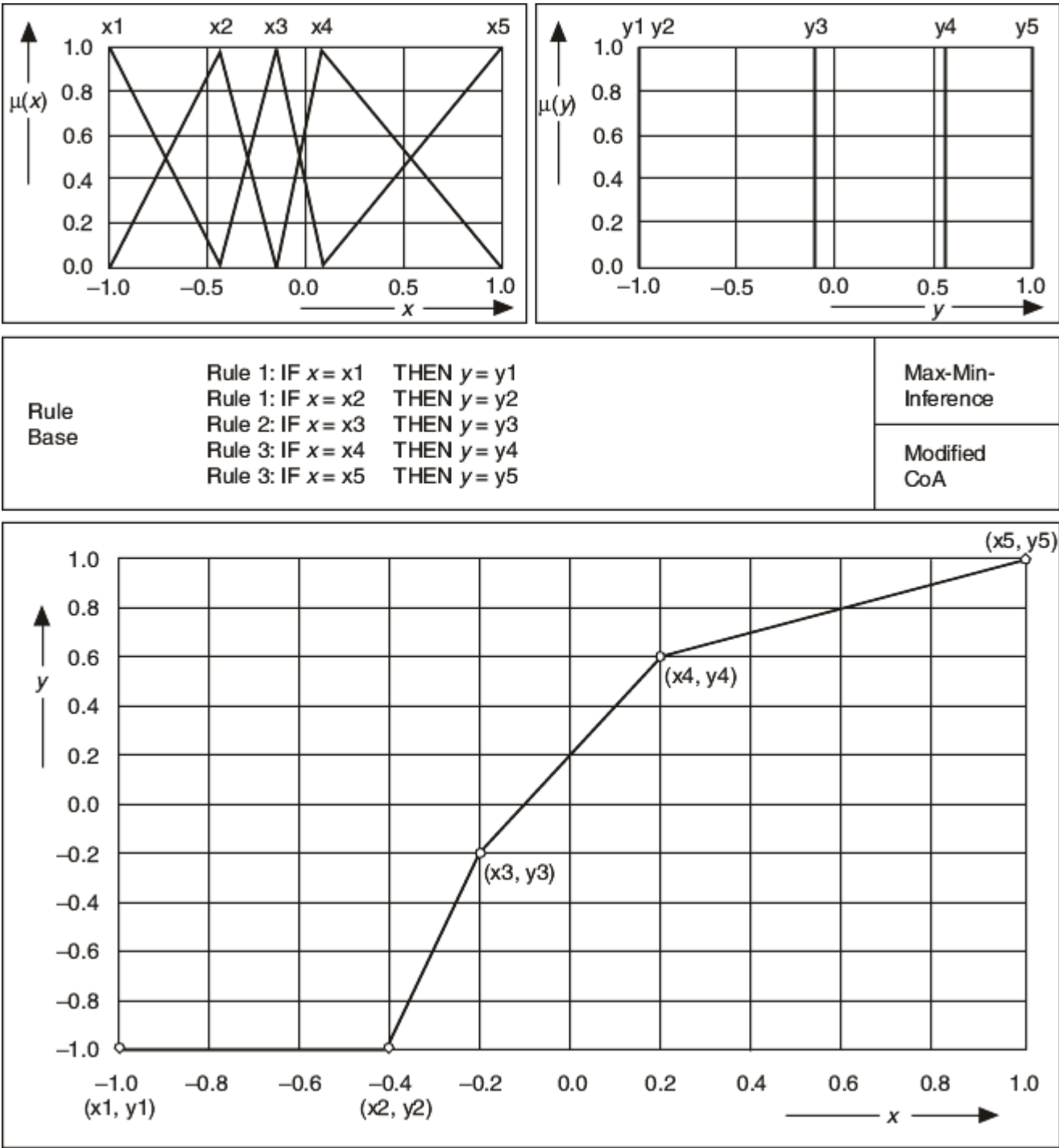


RELATED INFORMATION

[Online Autotuning Techniques](#) on page 1769

Reproducing a Given I/O Characteristic

Consider the stepped linear characteristic curve shown in the following image. You can describe the four linear sections with the five circled base points (x_i, y_i) .



To use a single-input fuzzy controller to reproduce the given characteristic, use five linguistic terms each for the input and output quantities, naming them $x_1, x_2, \dots, x_5$ and $y_1, y_2, \dots, y_5$, respectively. To obtain the stepped linear sections between the base points, you always must have exactly two available active **rules**. To implement this situation, entirely overlap the triangular **membership functions** for the input variable, giving each a typical value that corresponds to a certain base point component, x_i .

To obtain characteristic sections that are exactly linear, you must model the output variable with **singleton** membership functions, each of which has a typical value that corresponds to a certain base point component, y_i . The rule base is then a linguistic enumeration of the five base points.

Step Closed Loop Autotuning Technique

The Step Closed Loop autotuning technique, which is used by the Step Closed Loop instance of the PID Online Autotuning VI, performs a closed-loop step test. This technique is very accurate and reaches steady state faster than the Step Open Loop technique.

In closed-loop tuning, the PID Online Autotuning VI applies the Step Open Loop response with the controller in a closed loop, and then back-calculates the dynamics of the system by removing the closed loop gain.

The Step Closed Loop autotuning technique measures the deadtime T_d , time constant T , and process gain K values of a controller and multiplies them according to the formulas for the heuristic method in place. For example, most PID VIs use the Ziegler-Nichols method.



RELATED INFORMATION

[Online Autotuning Techniques](#) on page 1769

[Step Open Loop Autotuning Technique](#) on page 1777

[Step Open Loop Autotuning Technique](#) on page 1777

[Autotuning Methods for Testing PID Parameters](#) on page 1773

[Ziegler-Nichols Autotuning Method](#) on page 1779

Step Open Loop Autotuning Technique

The Step Open Loop autotuning technique, which is used by the Step Open Loop instance of the PID Online Autotuning VI, performs an open-loop step test. This technique assumes that you can model any process as a first-order lag and a pure deadtime.

The Step Open Loop autotuning technique measures the deadtime T_d , time constant T , and process gain K values of a controller and multiplies them according to the formulas for the heuristic method in place. For example, most PID VIs use the Ziegler-Nichols method.



RELATED INFORMATION

[Online Autotuning Techniques](#) on page 1769

[Autotuning Methods for Testing PID Parameters](#) on page 1773

[Ziegler-Nichols Autotuning Method](#) on page 1779

Updating PID Parameters After Autotuning

The PID [autotuning](#) VIs, including PID Autotuning, PID Advanced Autotuning, and PID Online Autotuning, calculate new parameters for a PID controller that result in the best response. When autotuning, you must write your code to apply the new PID controller parameters (the PID gains) when the autotuning is complete.

You can use the following methods to update PID parameters after autotuning.

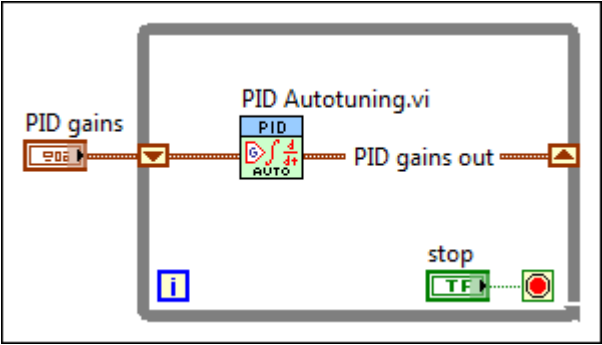


NOTE

In both of the following methods, you must manually save the **PID gains** so that you can use the **PID gains out** values the next time you run the control application. Ensure that the **PID gains** control shows the current updated parameters, select **Operate»Make Current Values Default**, and save the VI. If you do not want to manually save the VI after each run, you can [use a datalog file to save PID gains values](#).

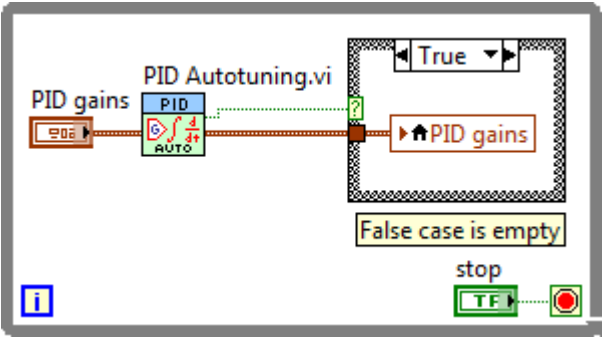
Using Shift Registers to Update PID Parameters

The following block diagram shows a simple implementation of using shift registers to update PID parameters in the PID Autotuning VI. The shift register on the left stores the initial value of the PID gains. **PID gains out** then passes an updated value to the right-hand shift register terminal when each control loop iteration completes. This method is simple, but limited in that the user cannot change **PID gains** manually while the control loop runs.



Using Local Variables to Update PID Parameters

In place of shift registers, you can use a local variable to store updated PID gains values. In the following block diagram, PID Autotuning VI reads the **PID gains** control on each iteration of the While Loop, and a local variable updates the control only when **tuning complete?** is TRUE. This method allows for manual control of the **PID gains** while the control loop executes.



Ziegler-Nichols Autotuning Method

Ziegler and Nichols developed heuristic methods for determining the parameters of a PID controller. The Autotuning Wizard and PID VIs that use the [PID Relay autotuning technique](#) use this method.

When using the Ziegler-Nichols autotuning method, select one of the following three types of desired loop response performance: **fast** (1/4 damping ratio), **normal** (some overshoot), and **slow** (little overshoot). The following tables show the tuning formulas for each type of loop performance.



NOTE

During tuning, the process remains under closed-loop PID control. You do not need to switch off the existing controller and perform the experiment under open-loop conditions. In the setpoint relay experiment, the setpoint signal mirrors the setpoint for the PID controller.

Ziegler-Nichols Closed Loop/Frequency Response Method

The following tables show the tuning formulas for different types of loop performance under proportional-only control, where K_u is the controller gain and T_u is the period of oscillation.

Fast Performance			
Controller	K_c	T_i	T_d
P	$0.5 K_u$	—	—
PI	$0.4 K_u$	$0.8 T_u$	—
PID	$0.6 K_u$	$0.5 T_u$	$0.12 T_u$

Normal Performance			
Controller	K_c	T_i	T_d
P	$0.2 K_u$	—	—
PI	$0.18 K_u$	$0.8 T_u$	—
PID	$0.25 K_u$	$0.5 T_u$	$0.12 T_u$

Slow Performance			
Controller	K_c	T_i	T_d

Slow Performance			
P	$0.13 K_u$	—	—
PI	$0.13 K_u$	$0.8 T_u$	—
PID	$0.15 K_u$	$0.5 T_u$	$0.12 T_u$

Ziegler-Nichols Open Loop/Step Response Method

The following tables shows the tuning formulas for different types of loop performance under PI or PID control, where T_P is the time constant and τ is the dead time.

Fast Performance			
Controller	K_c	T_i	T_d
P	T_P / τ	—	—
PI	$0.9 T_P / \tau$	3.33τ	—
PID	$1.1 T_P / \tau$	2.0τ	0.5τ

Normal Performance			
Controller	K_c	T_i	T_d
P	$0.44 T_P / \tau$	—	—
PI	$0.4 T_P / \tau$	5.33τ	—
PID	$0.53 T_P / \tau$	4.0τ	0.8τ

Slow Performance			
Controller	K_c	T_i	T_d
P	$0.26 T_P / \tau$	—	—
PI	$0.24 T_P / \tau$	5.33τ	—
PID	$0.32 T_P / \tau$	4.0τ	0.8τ



NOTE

The previous tables provides proportional gain K_c values. The following equation describes the relationship between proportional gain and proportional band PB:
 $K_c = 100 / PB$

Defuzzification

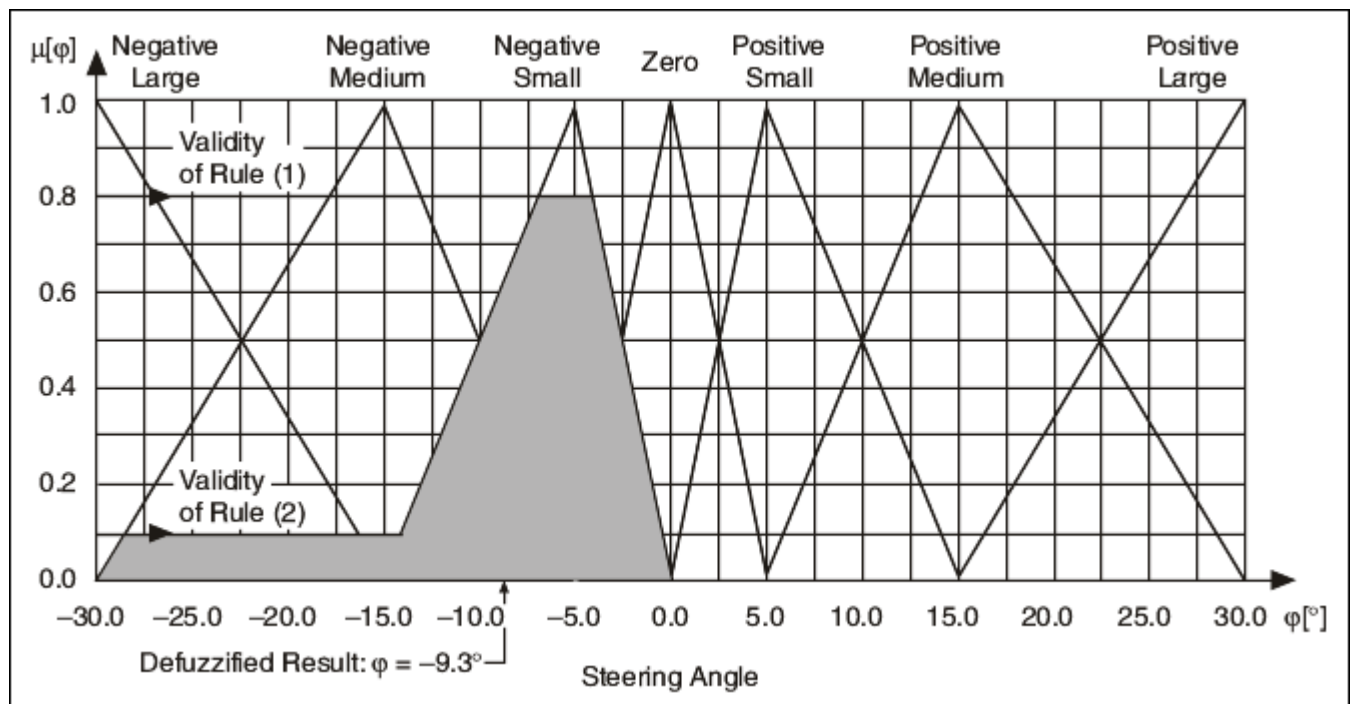
Center of Area (CoA)

In the Center of Area (CoA) [defuzzification method](#), also called the Center of Gravity (CoG) method, the fuzzy controller first calculates the area under the scaled [membership functions](#) and within the range of the output variable. The fuzzy logic controller then uses the following equation to calculate the geometric center of this area.

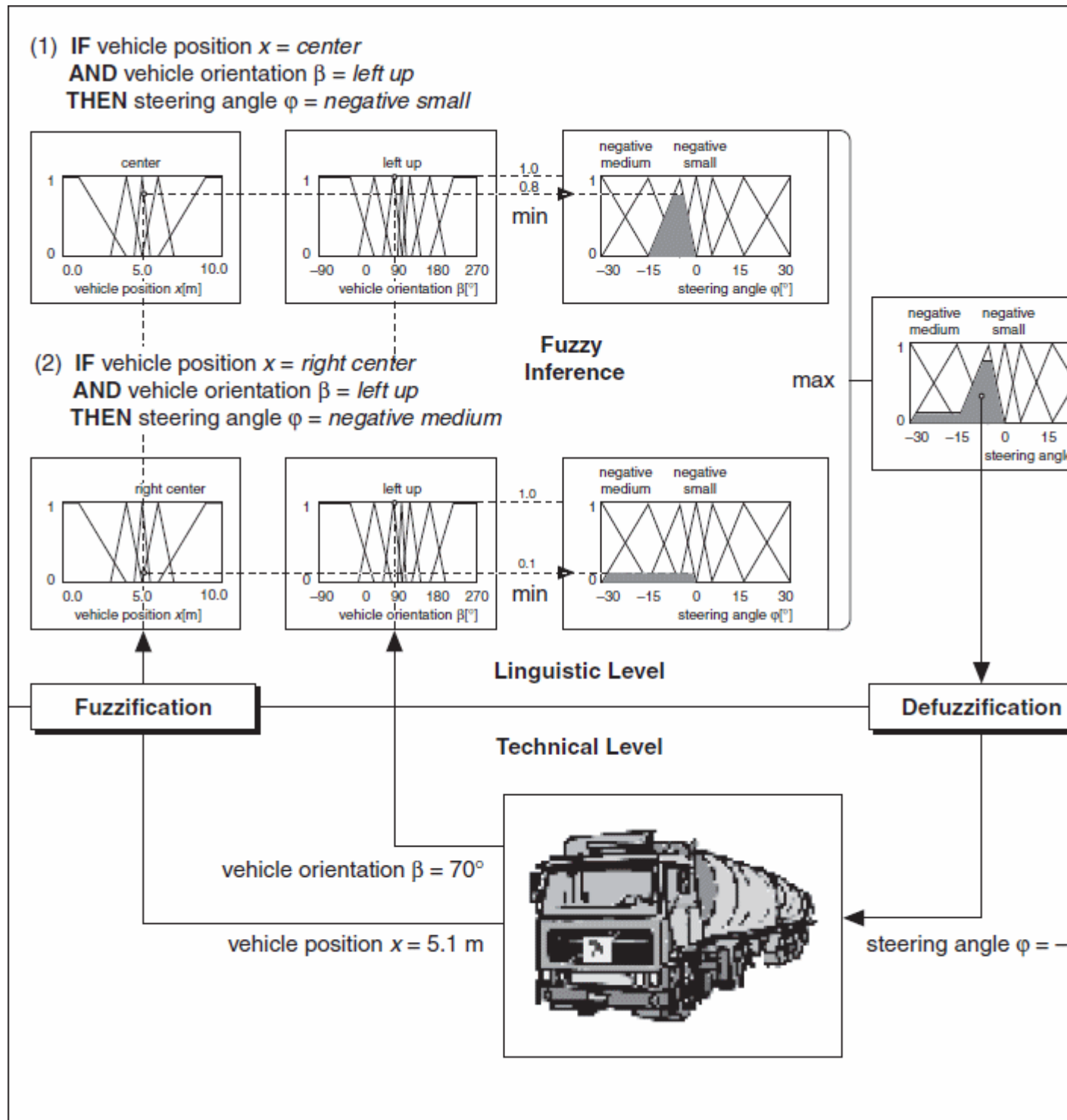
$$CoA = \frac{\int_{x_{min}}^{x_{max}} f(x) * x \, dx}{\int_{x_{min}}^{x_{max}} f(x) \, dx}$$

where CoA is the center of area, x is the value of the linguistic variable, and x_{min} and x_{max} represent the range of the [linguistic variable](#). The Center of Area defuzzification method effectively calculates the best compromise between multiple output linguistic terms.

The following image shows the Center of Area defuzzification method for the Steering Angle ϕ output linguistic variable of a vehicle maneuvering fuzzy system, assuming the [Minimum](#) implication method. The shaded portion of the graph represents the area under the scaled membership functions.



The following image summarizes the process of a fuzzy controller for the vehicle maneuvering example using the Center of Area defuzzification method.



Center of Maximum (CoM)

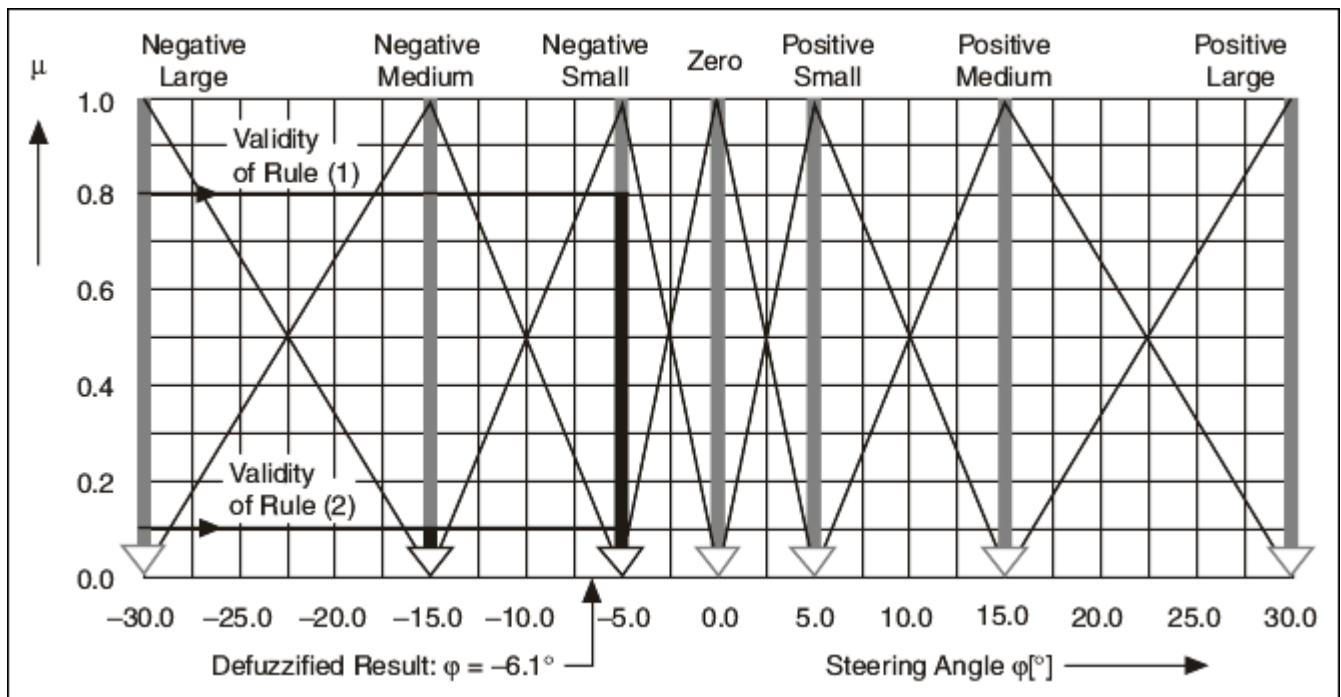
In the Center of Maximum (CoM) [defuzzification method](#), the fuzzy logic controller first determines the typical numerical value for each scaled [membership function](#). The typical

numerical value is the mean of the numerical values corresponding to the degree of membership at which the membership function was scaled.

$$x_{final} = \frac{(x_1\mu_1 + x_2\mu_2 + \dots + x_n\mu_n)}{(\mu_1 + \mu_2 + \dots + \mu_n)}$$

where x_n is the typical numerical value for the scaled membership function n , and μ_n is the degree of membership at which the membership function n was scaled.

The following image illustrates how to use the CoM defuzzification method with the vehicle maneuvering example.



The values -15° and -5° are the typical values of the *linguistic terms* Negative Medium and Negative Small. The degrees of truth for these linguistic terms are 0.1 and 0.8, respectively. Therefore, the defuzzified crisp output value ϕ_{final} is calculated by the following equation:

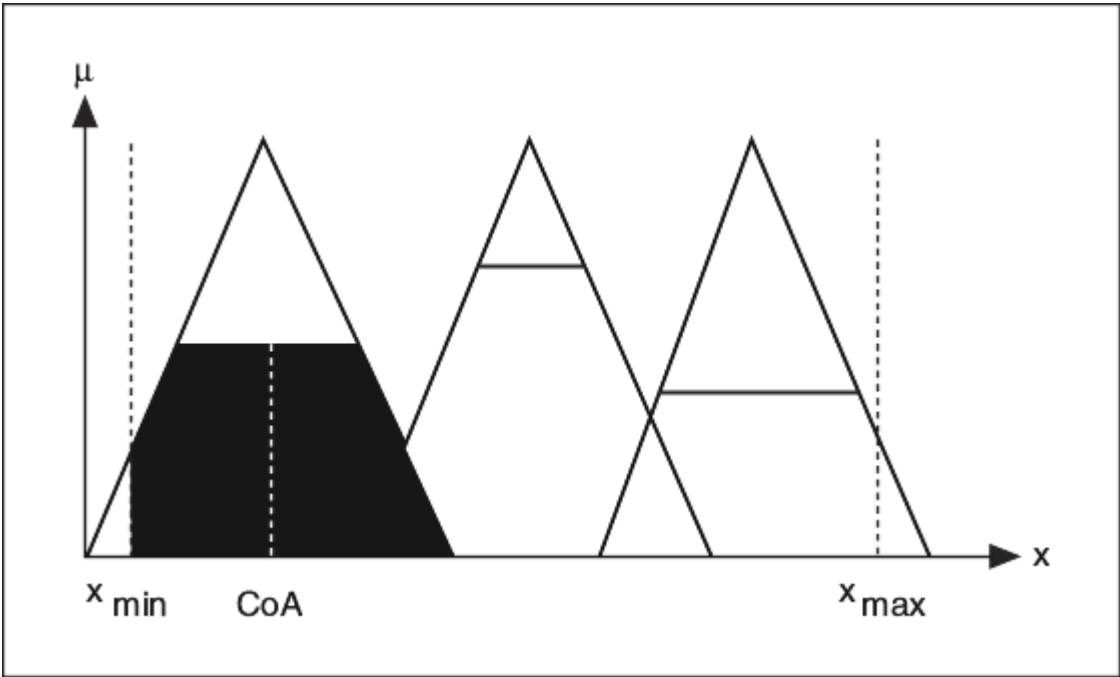
$$\phi_{final} = \frac{((-15^\circ)(0.1) + (-5^\circ)(0.8))}{(0.1 + 0.8)} = -6.1^\circ$$

The CoM defuzzification method is identical to the Center of Area (CoA) method with singleton membership functions.

The CoM and CoA defuzzification methods usually apply to closed-loop control applications of fuzzy logic. These methods usually result in continuous output signals because a small change in input values does not change the best compromise value for the output.

Center of Sums (CoS)

In the Center of Sums (CoS) *defuzzification method*, the fuzzy logic controller first calculates the geometric center of area for each *membership function*, as shown in the following image.



The fuzzy controller then uses the following equation to calculate a weighted average of the geometric center of area for all membership functions.

$$x_{final} = \frac{(CoA_1 area_1 + CoA_2 area_2 + ... + CoA_n area_n)}{(area_1 + area_2 + ... + area_n)}$$

where CoA_n is the geometric center of area of the scaled membership function n , and $area_n$ is the area of the scaled membership function n .

Chien-Hrones-Reswick Autotuning Method

The Chien-Hrones-Reswick autotuning method focuses on setpoint response and disturbance response. This method provides formulas for 0% and 20% overshoot.



NOTE
You must use the PID Autotuning Design VI to implement this autotuning method. Select the Chien-Hrones-Reswick polymorphic instance and use the **Tuning specifications** input to configure which formula you want to use.

The following tables shows the Chien-Hrones-Reswick recommendations for each tuning formula:

Regulator – 0% overshoot

Controller	K_c	T_i	T_d
P	$0.3 T_P / \tau$	—	—
PI	$0.6 T_P / \tau$	4τ	—
PID	$0.95 T_P / \tau$	2.4τ	0.42τ

where T_P is the time constant and τ is the deadtime.

Regulator – 20% overshoot

Controller	K_c	T_i	T_d
P	$0.7 T_P / \tau$	—	—
PI	$0.7 T_P / \tau$	2.3τ	—
PID	$1.2 T_P / \tau$	2τ	0.42τ

where T_P is the time constant and τ is the deadtime.

Servo – 0% overshoot

Controller	K_c	T_i	T_d
P	$0.3 T_P / \tau$	—	—
PI	$0.35 T_P / \tau$	1.2τ	—
PID	$0.6 T_P / \tau$	τ	0.5τ

where T_P is the time constant and τ is the deadtime.

Servo – 20% overshoot

Controller	K_c	T_i	T_d
P	$0.7 T_P / \tau$	—	—
PI	$0.6 T_P / \tau$	τ	—
PID	$0.95 T_P / \tau$	1.4τ	0.47τ

where T_P is the time constant and τ is the deadtime.

Cohen-Coon Autotuning Method

The Cohen-Coon autotuning method is an offline method for tuning PID controllers. This method uses PID parameters obtained from an open-loop transfer function experiment. This method is similar to the [Ziegler-Nichols method](#), but provides better results when the controller has a large deadtime T_d relative to the time constant T .



NOTE
You must use the PID Autotuning Design VI to implement this autotuning method.

The following table shows the Cohen-Coon recommendations for tuning parameters under different tuning specifications:

Controller	K_c	T_i	T_d
P	$\frac{T}{K_c * T_d} \left[1 + \frac{T_d}{3T} \right]$	—	—
PI	$\frac{T}{K_c * T_d} \left[0.9 + \frac{T_d}{12T} \right]$	$\frac{T_d \left[30 + 3 \left(\frac{T_d}{T} \right) \right]}{9 + 20 \left(\frac{T_d}{T} \right)}$	—
PID	$\frac{T}{K_c * T_d} \left[\frac{4}{3} + \frac{T_d}{4T} \right]$	$\frac{T_d \left[32 + 6 \left(\frac{T_d}{T} \right) \right]}{13 + 8 \left(\frac{T_d}{T} \right)}$	$\frac{4T_d}{11 + 2 \left(\frac{T_d}{T} \right)}$

Where T is the time constant

K_c is the proportional gain

T_d is the deadtime

Complete Rule Bases

A [rule base](#) with at least one active rule for each possible combination of input linguistic variables and linguistic terms is a complete rule base. If you define an incomplete rule base, you must specify a default linguistic term for each output linguistic variable so the fuzzy controller can handle situations in which no rules are active.

The Vehicle Position x input linguistic variable has five linguistic terms, and the Vehicle Orientation β linguistic variable has seven linguistic terms. Therefore, the rule base of the vehicle maneuvering example consists of $N = 5 * 7 = 35$ rules. You can document the

complete rule base in matrix form, as shown in the following image for the vehicle maneuvering fuzzy system.

AND		Vehicle Position x [m]				
		Left	Left Center	Center	Right Center	Right
Vehicle Orientation β [°]	Left Down	Negative Small	Negative Medium	Negative Medium	Negative Large	Negative Large
	Left	Positive Small	Negative Small	Negative Medium	Negative Large	Negative Large
	Left Up	Positive Medium	Positive Small	Negative Small	Negative Medium	Negative Large
	Up	Positive Medium	Positive Medium	Zero	Negative Medium	Negative Medium
	Right Up	Positive Large	Positive Medium	Positive Small	Negative Small	Negative Medium
	Right	Positive Large	Positive Large	Positive Medium	Positive Small	Negative Small
	Right Down	Positive Large	Positive Large	Positive Medium	Positive Medium	Negative Small

In the previous image, each column or row represents an antecedent of a rule. The term at the intersection of a column and a row is the consequent of the rule corresponding to the aggregated rule antecedent. For example, the following rule is highlighted:

IF Vehicle Position x is Left Center AND Vehicle Orientation β is Left, THEN Steering Angle ϕ is Negative Small.

Cascading Fuzzy Systems

Plotting a rule base in matrix form, as in the previous image, is helpful for detecting inconsistencies, such as contradictory rules. However, plotting a rule base in matrix form is efficient only for small rule bases. Detecting inconsistencies in large rule bases is difficult. For fuzzy systems with numerous controller inputs, you can use cascading fuzzy systems to avoid large rule bases. In cascading fuzzy systems, the outputs of the first fuzzy system serve as the inputs to the next fuzzy system, and so on.

Control Output Rate Limiting

Sudden changes in control output are often undesirable or even dangerous for many control applications. For example, a sudden large change in setpoint can cause a very large change in controller output. Although in theory this large change in controller output results in fast response of the system, it also can cause unnecessary wear on actuators or sudden large power demands. In addition, the PID controller can amplify noise in the system and result in a constantly changing controller output.

You can use the PID Output Rate Limiter VI to avoid the problem of sudden changes in controller output. Wire the **output** value of the PID VI to the **input (controller output)** input of the PID Output Rate Limiter VI to limit the slew, or rate of change, of the output to the value of the **output rate (EGU/min)**.

Assign a value to **initial output** to specify the output value on the first call to the VI. Use the **dt (s)** input to specify the same control-loop cycle time you use for the PID VI.

Converting Between Percentage of Full Scale and Engineering Units

The default setpoint, process variable, and output ranges for the PID VIs correspond to percentage of full scale. In other words, proportional gain K_c relates percentage of full scale output to percentage of full scale input. This is the default behavior of many PID controllers used for process control applications. To implement PID in this way, you must scale all inputs to percentage of full scale and all controller outputs to actual engineering units, for example, volts for analog output.

You can use the PID EGU to Percentage VI to convert any input from real engineering units to percentage of full scale, and you can use the PID Percentage to EGU VI to convert the controller output from percentage to real engineering units.



NOTE

The PID VIs do not use the setpoint range and output range information to convert values to percentages in the PID algorithm. The controller gain relates the output in engineering units to the input in engineering units. For example, a gain value of 1 produces an output of 10 for a difference between setpoint and process variable of 10, regardless of the output range and setpoint range.

Creating a Rule Base

After *create linguistic variables*, *defining linguistic terms*, and *creating membership functions*, the last step of *fuzzy system design* is creating a rule base. Rules describe, in words, the relationships between input and output linguistic variables based on their linguistic terms. A rule base is the set of rules for a fuzzy system.

To create a rule, you must specify the antecedents, or IF portions, and consequents, or THEN portions, of the rule. For example, consider the following rule for a vehicle maneuvering fuzzy system:

IF Vehicle Position x is Left Center AND Vehicle Orientation β is Left Up, THEN Steering Angle ϕ is Positive Small.

The clauses " Vehicle Position x is Left Center " and " Vehicle Orientation β is Left Up " are the antecedents of this rule. The clause " Steering Angle ϕ is Positive Small " is the consequent of this rule.

Associate an input linguistic variable with a corresponding linguistic term to form an antecedent. Associate an output linguistic variable with a corresponding linguistic term to form a consequent. The consequent of a rule represents the action you want the fuzzy controller to take if the linguistic terms of the input linguistic variables in the rule are met.

When constructing a rule base, avoid contradictory rules, or rules with the same IF portion but different THEN portions. A consistent rule base is a rule base that has no contradictory rules.



NOTE

The rule base itself is the biggest influence on the *I/O characteristics* of a fuzzy controller. The rule base determines the principal functionality of the controller. Refer to the *Using the Mean of Maximum (MoM) Defuzzification Method* example to see how changing the rule base can affect the output characteristic of a fuzzy controller.

Total Possible Rules

The total number N of possible rules for a fuzzy system is defined by the following equation:

$$N = p_1 * p_2 * \dots * p_n$$

where p_n is the number of linguistic terms for the input linguistic variable n .

If each input linguistic variable has the same number of linguistic terms, the total number N of possible rules is defined by the following equation:

$$N = p^m$$

where p is the number of linguistic terms for each input linguistic variable and m is the number of input linguistic variables. For example, for three input linguistic variables with five linguistic terms each, the total number of possible rules is $N = 5^3 = 125$.

Additional Information About Rule Bases

The following topics contain additional important information about creating rule bases:

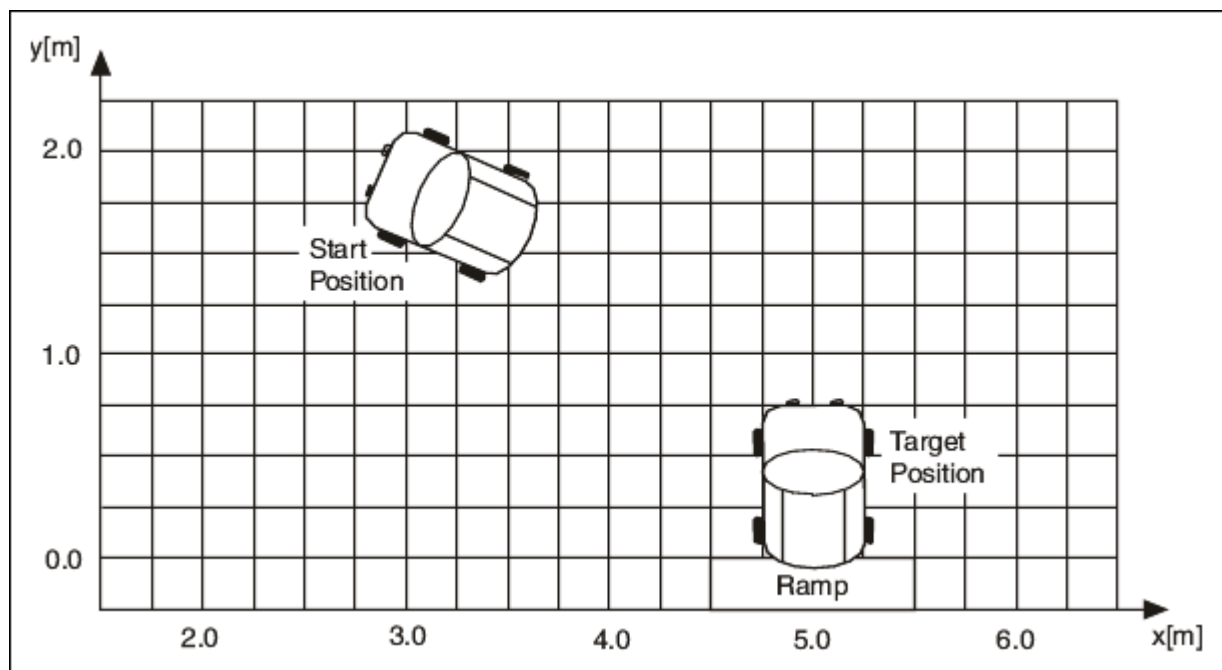
- [Complete Rule Bases](#)
- [Specifying an Antecedent Connective](#)
- [Specifying a Consequent Implication Method](#)

Creating Linguistic Variables

When you [design a fuzzy system](#), linguistic variables represent, in words, the input variables and output variables of the system you want to control.

When you create a linguistic variable to represent an input or output variable, decide how many linguistic terms, or categories of values of the linguistic variable, you want to create. Linguistic variables usually have an odd number of linguistic terms, with a middle linguistic term and symmetric linguistic terms at each extreme. In most applications, three to seven linguistic terms are sufficient for categorizing the values of a linguistic variable.

Consider an example in which you want to automate a vehicle to park itself from an arbitrary starting position. A driver can control the vehicle by constantly evaluating the current status of the vehicle, such as the distance from the target position and the orientation of the vehicle, to derive the correct steering angle. The following image illustrates this example.



You can define two input linguistic variables for this example. Vehicle Position x represents the vehicle position in relation to the destination. Vehicle Orientation β represents the orientation of the vehicle. You also can define an output linguistic variable, Steering Angle ϕ , to represent the steering angle of the vehicle that you want to control.

Defining Linguistic Terms

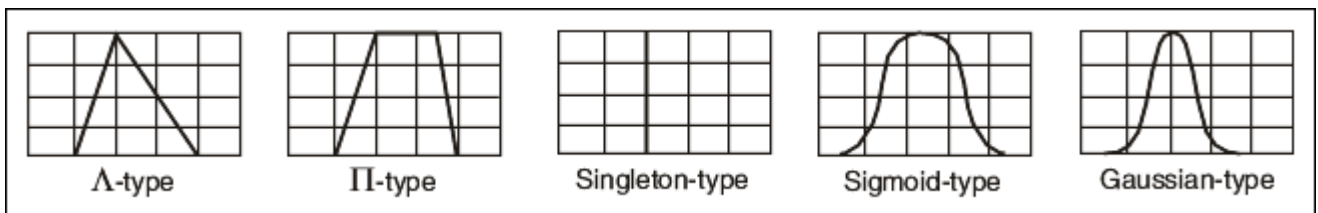
You can define linguistic terms of Left, Left Center, Center, Right Center, and Right for the Vehicle Position x input linguistic variable to describe the possible positions of the vehicle in relation to the destination. You can define linguistic terms of Left Down, Left, Left Up, Up, Right Up, Right, and Right Down for the Vehicle Orientation β input linguistic variable to describe the possible orientations of the vehicle. The linguistic terms of the Steering Angle ϕ output linguistic variable must represent both the direction and magnitude that the steering angle changes. Therefore, you can use the linguistic terms Negative Large, Negative Medium, Negative Small, Zero, Positive Small, Positive Medium, and Positive Large for this output linguistic variable.

The next step in designing a fuzzy system is *creating membership* functions that correspond to the linguistic terms you define.

Creating Membership Functions

When you *design a fuzzy system*, you first *create linguistic variables* for which you *define linguistic terms*. After you define the linguistic terms, you must apply membership functions to those terms. Membership functions are simply numerical functions that correspond to linguistic terms. A membership function represents the degree of membership of linguistic variables within their linguistic terms.

You can apply the normalized standard membership functions shown in the following image to most technical processes. These standard functions include Λ -type (triangular shape), Π -type (trapezoidal shape), singleton-type (vertical line shape), Sigmoid-type (wave shape), and Gaussian-type (bell shape) membership functions.

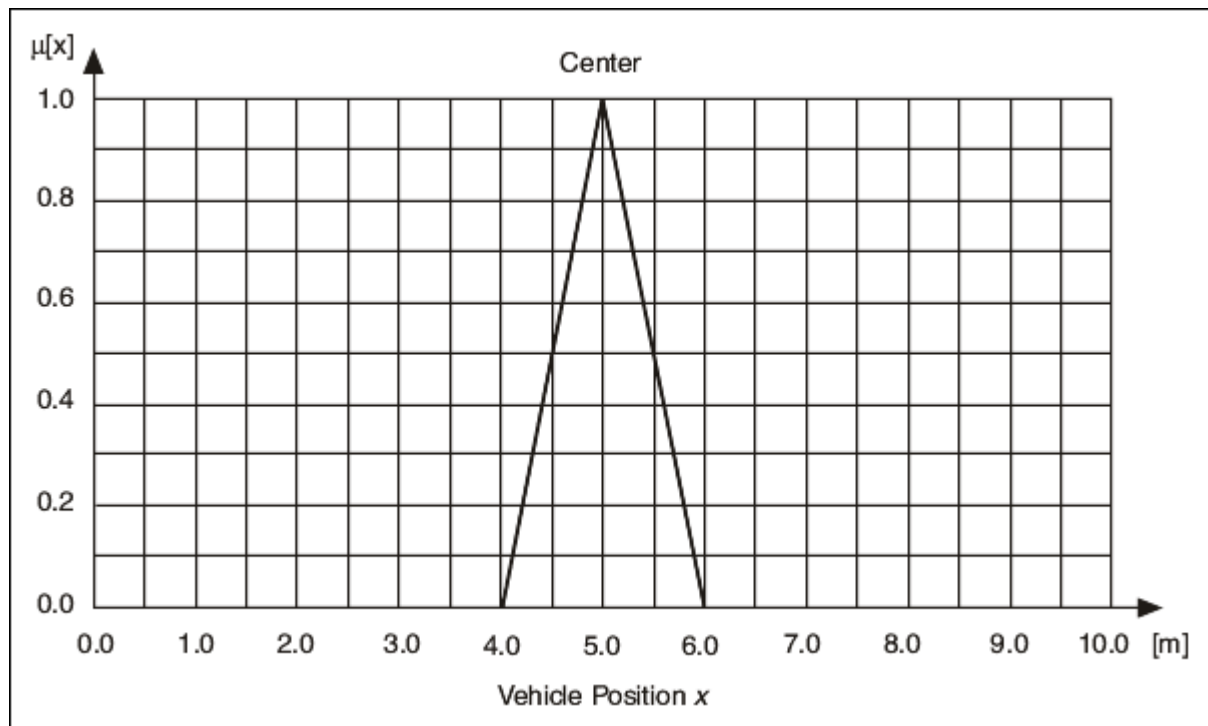


Example Triangle Membership Function for the Linguistic Term

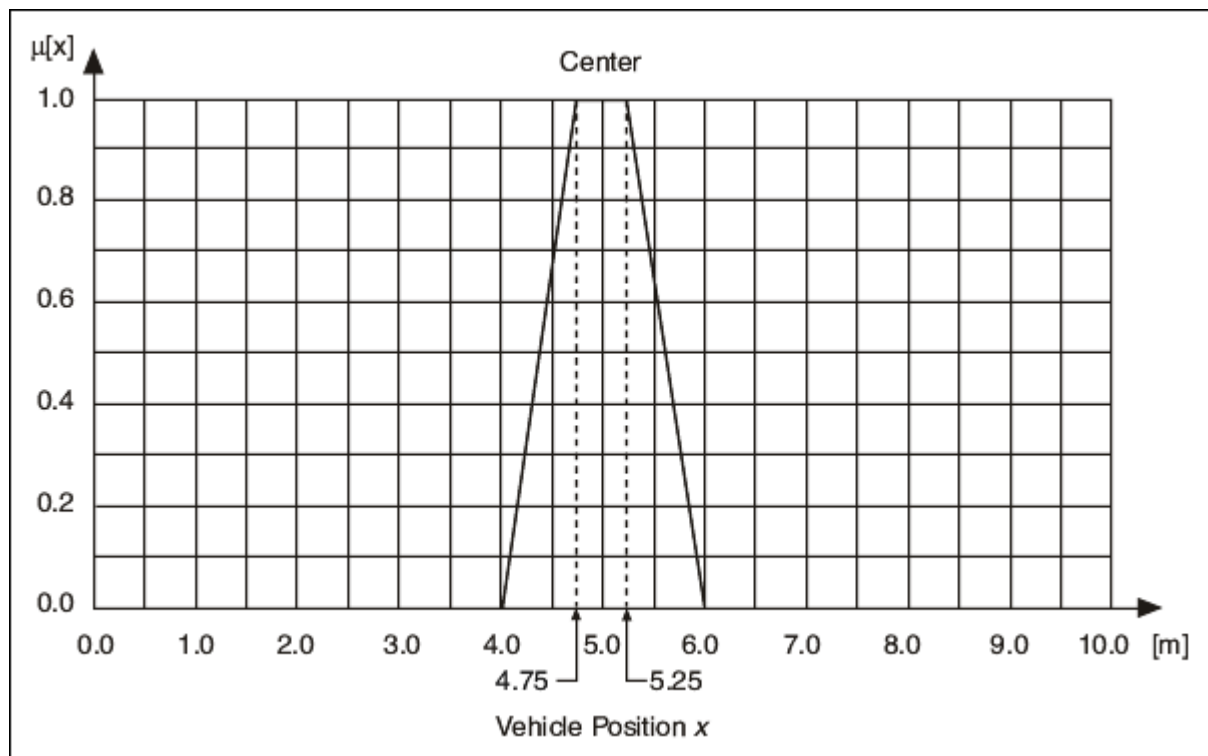
Example Trapezoidal Membership Function for the Linguistic Term Center

Membership Functions for a Vehicle Maneuvering Fuzzy System

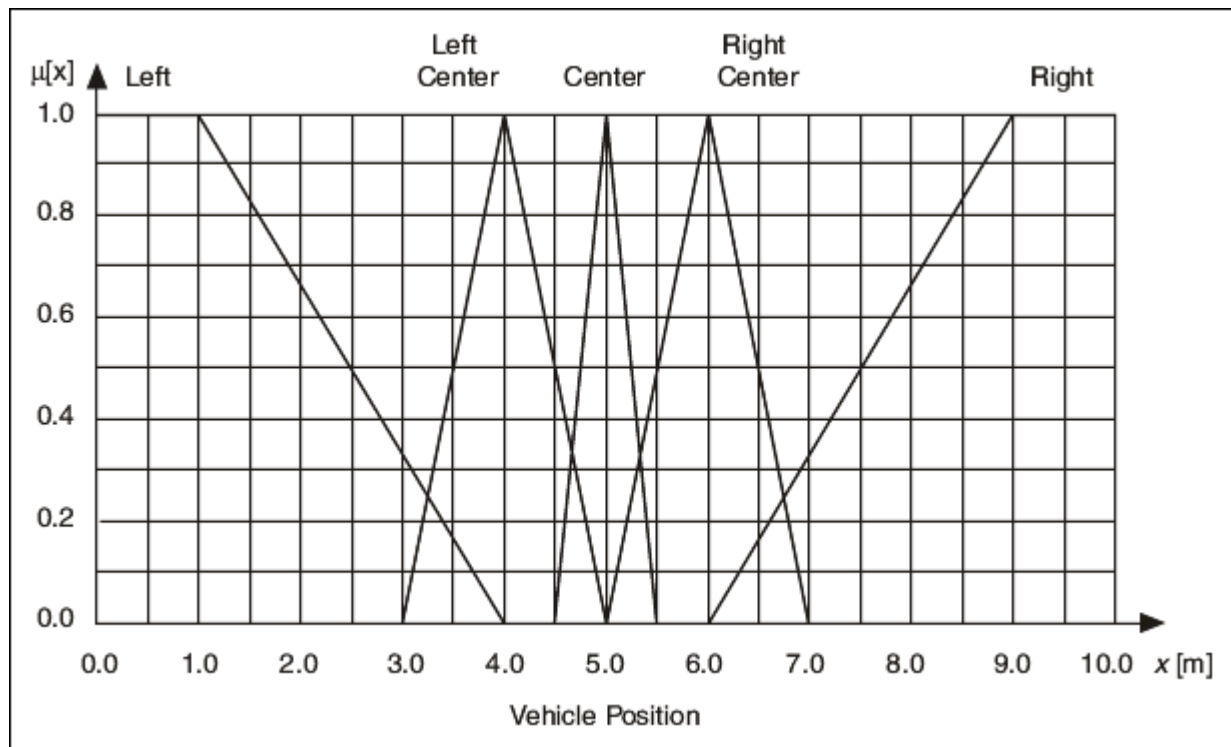
For example, the linguistic variable Vehicle Position x might have full membership (1) within the linguistic term Center at 5 meters, no membership (0) within that term at 4 meters or less and 6 meters or greater, and partial membership at all distances between 4 and 6 meters. If you plot the degree of membership against the value of Vehicle Position x , you can see that the resulting membership function is a triangle function.



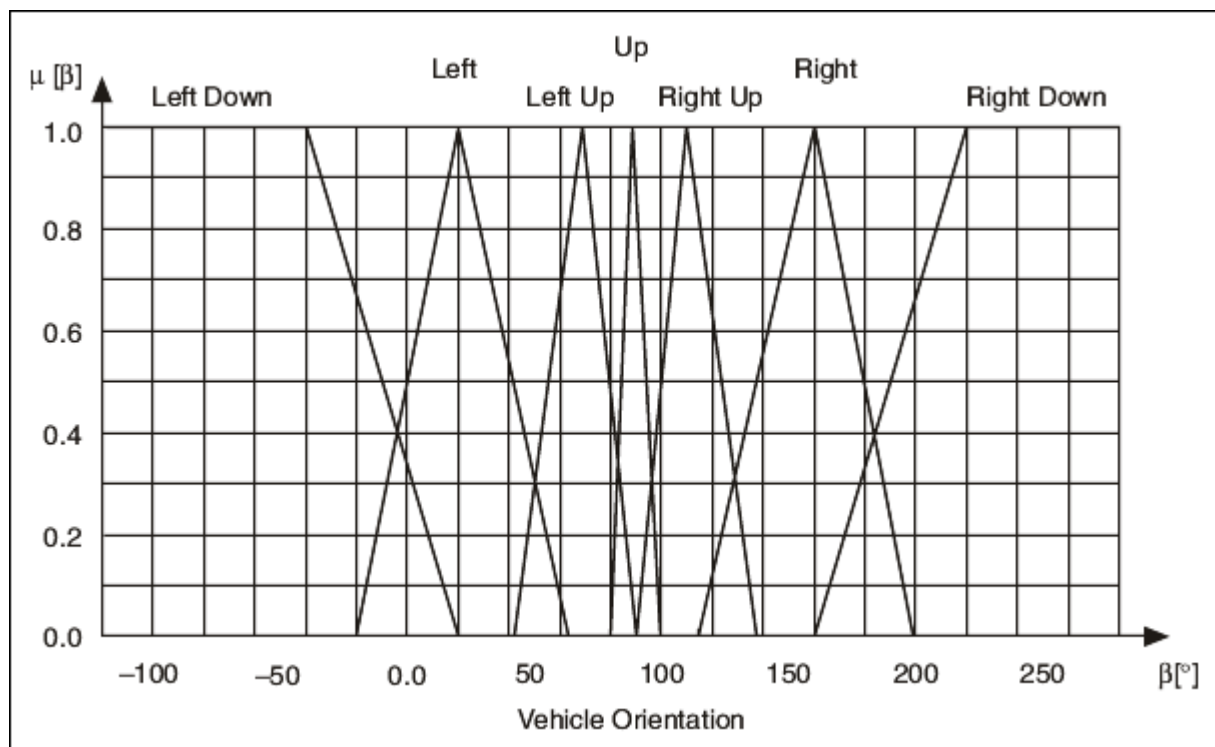
Sometimes a linguistic variable has full membership within a linguistic term at a range of values rather than at a point value. If, for example, the linguistic variable Vehicle Position x has full membership within the linguistic term Center at values $x = 5 \pm 0.25$ m, a trapezoidal membership function applies, as shown in the following image.



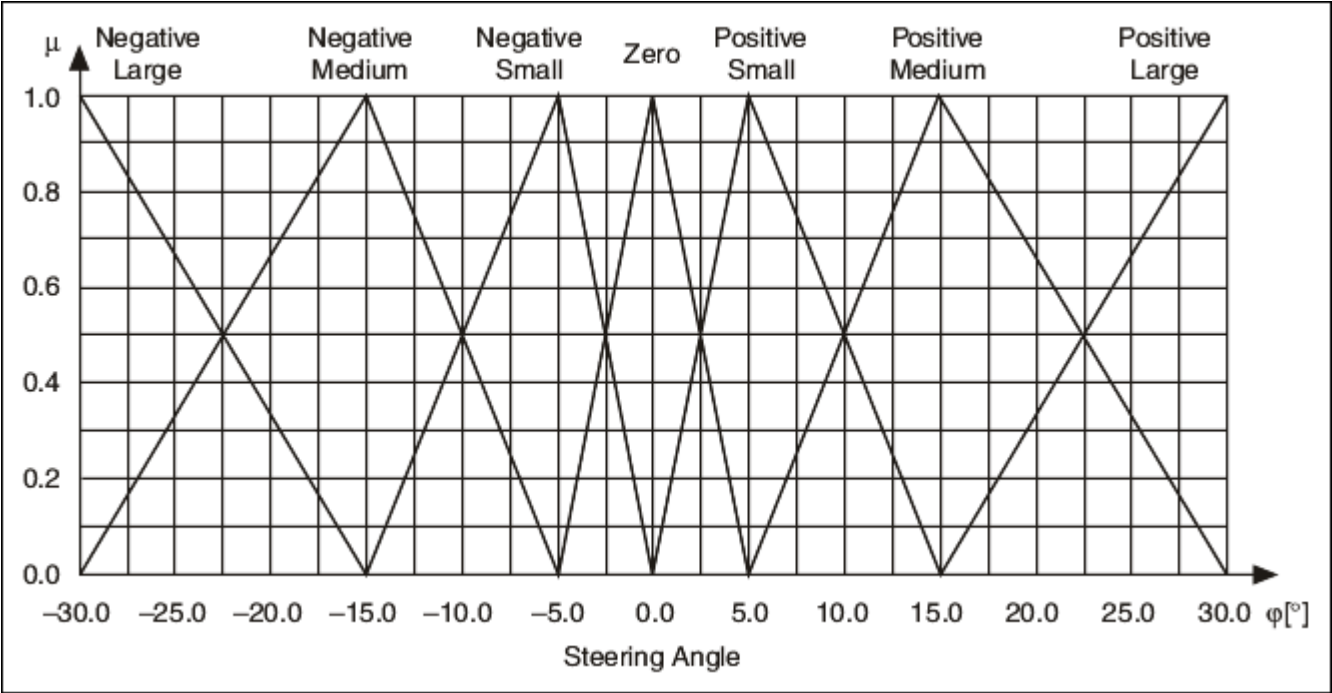
The following image shows the membership functions for Vehicle Position x .



The following image shows the membership functions for Vehicle Orientation β .



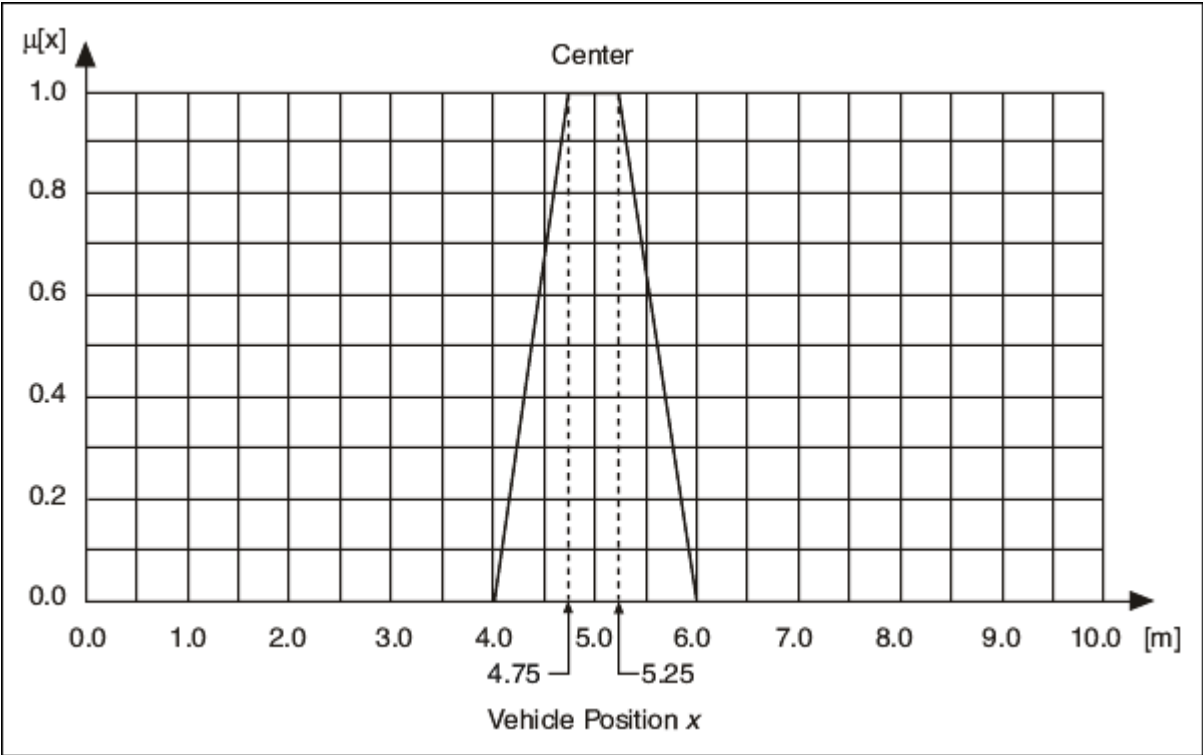
The following image shows the membership functions for Steering Angle ϕ .



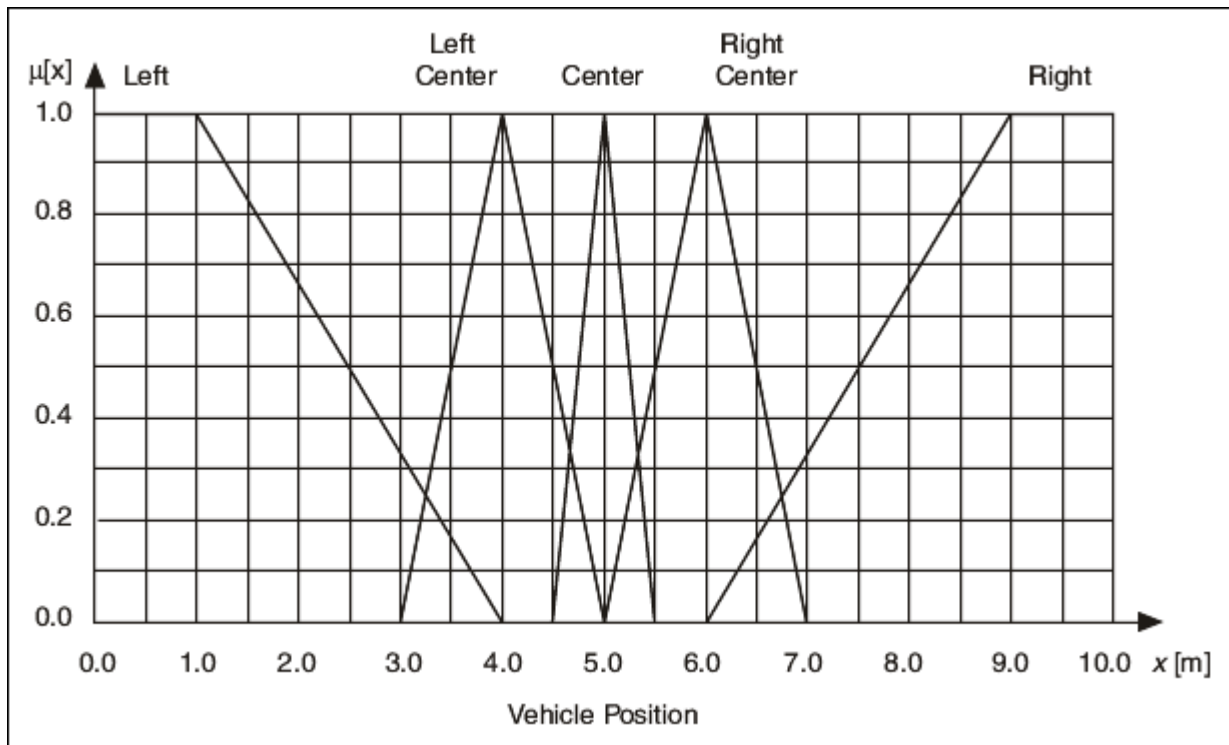
Example Trapezoidal Membership Function for the Linguistic Term

Membership Functions for a Vehicle Maneuvering Fuzzy System

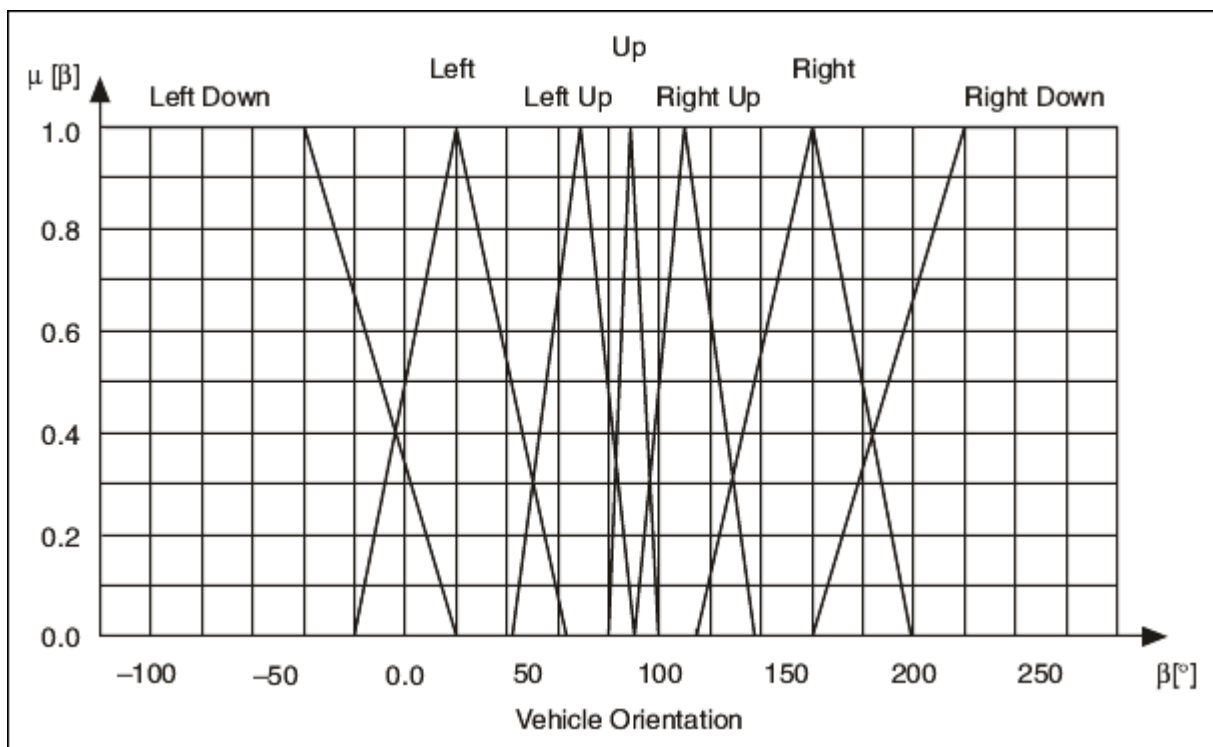
Sometimes a linguistic variable has full membership within a linguistic term at a range of values rather than at a point value. If, for example, the linguistic variable Vehicle Position x has full membership within the linguistic term Center at values $x = 5 \pm 0.25$ m, a trapezoidal membership function applies, as shown in the following image.



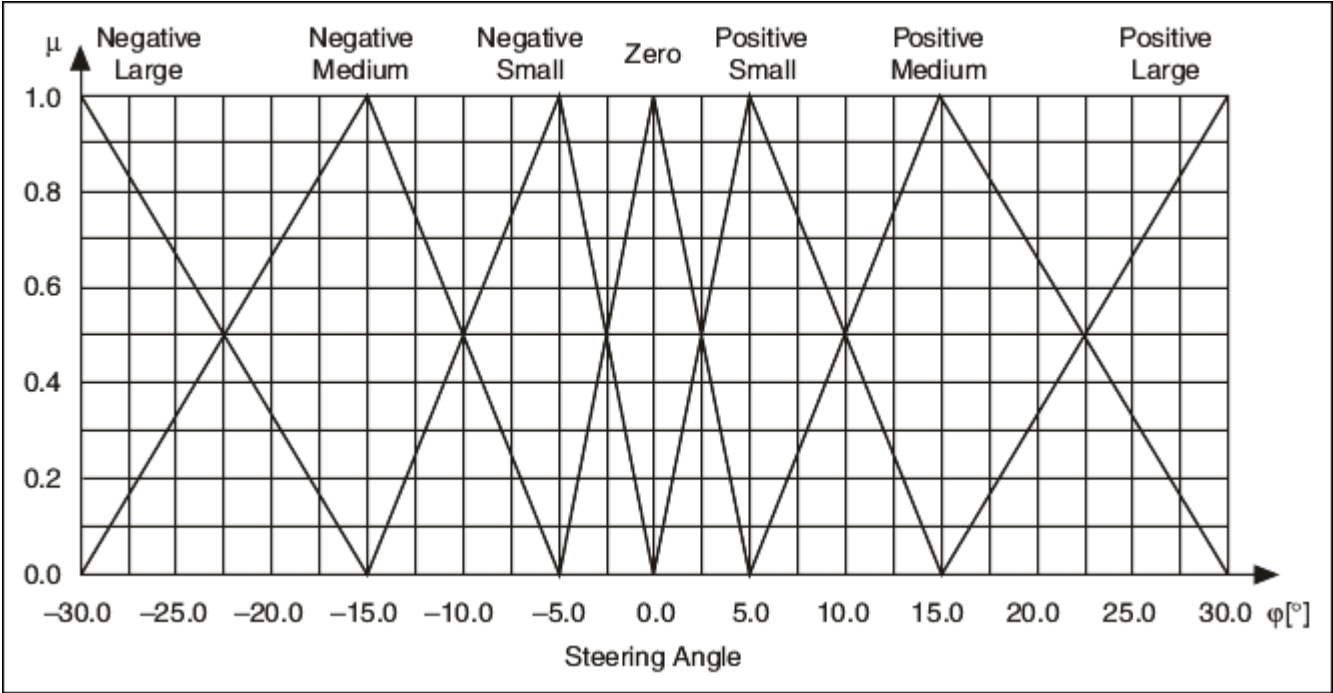
The following image shows the membership functions for Vehicle Position x .



The following image shows the membership functions for Vehicle Orientation β .

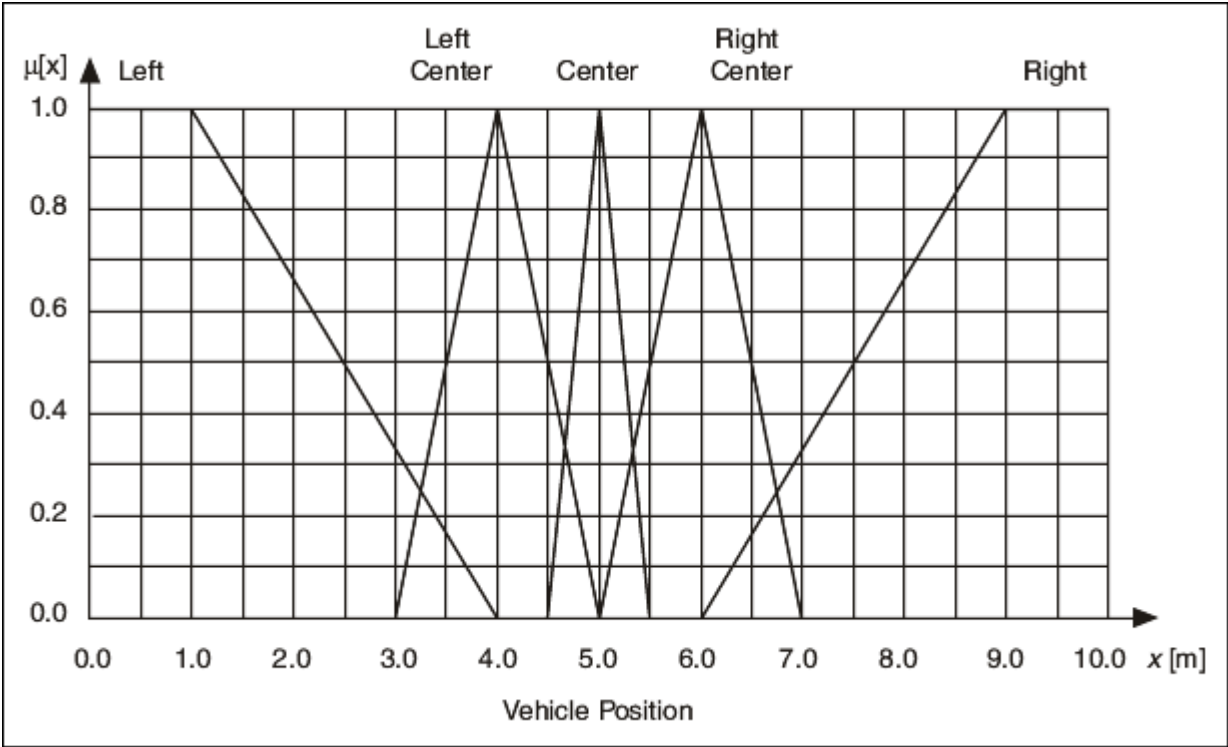


The following image shows the membership functions for Steering Angle ϕ .

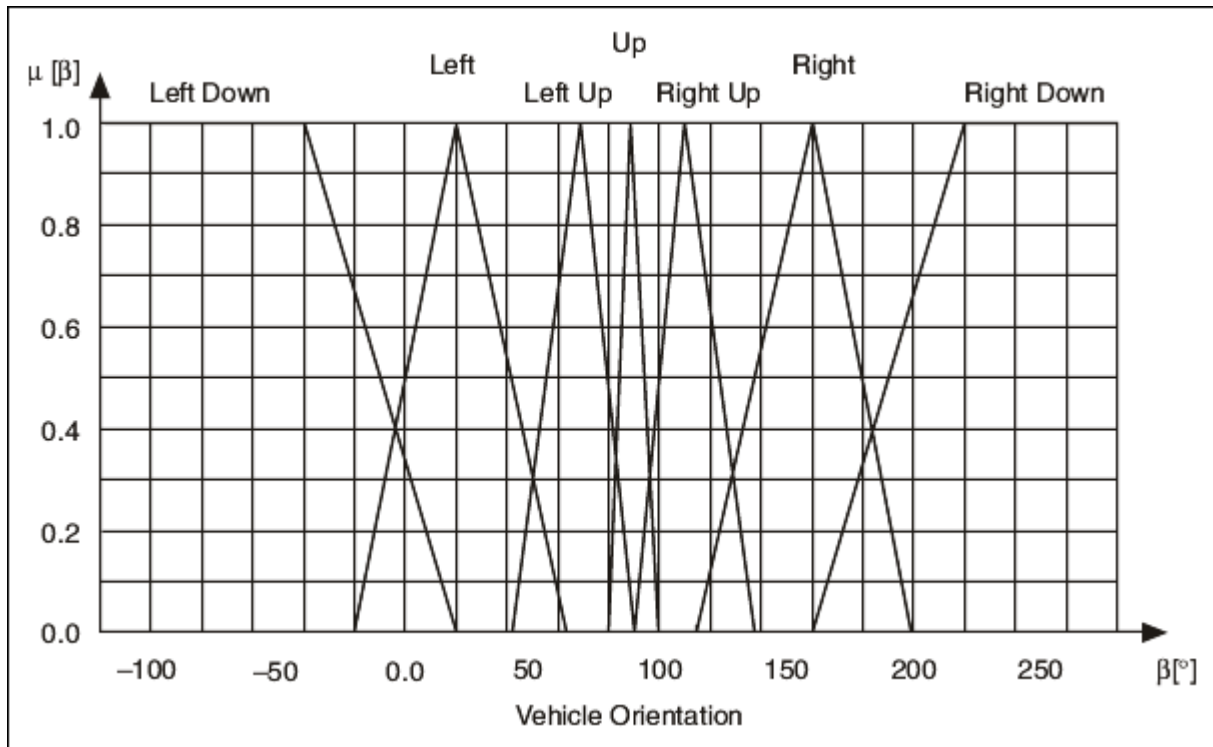


Membership Functions for a Vehicle Maneuvering Fuzzy System

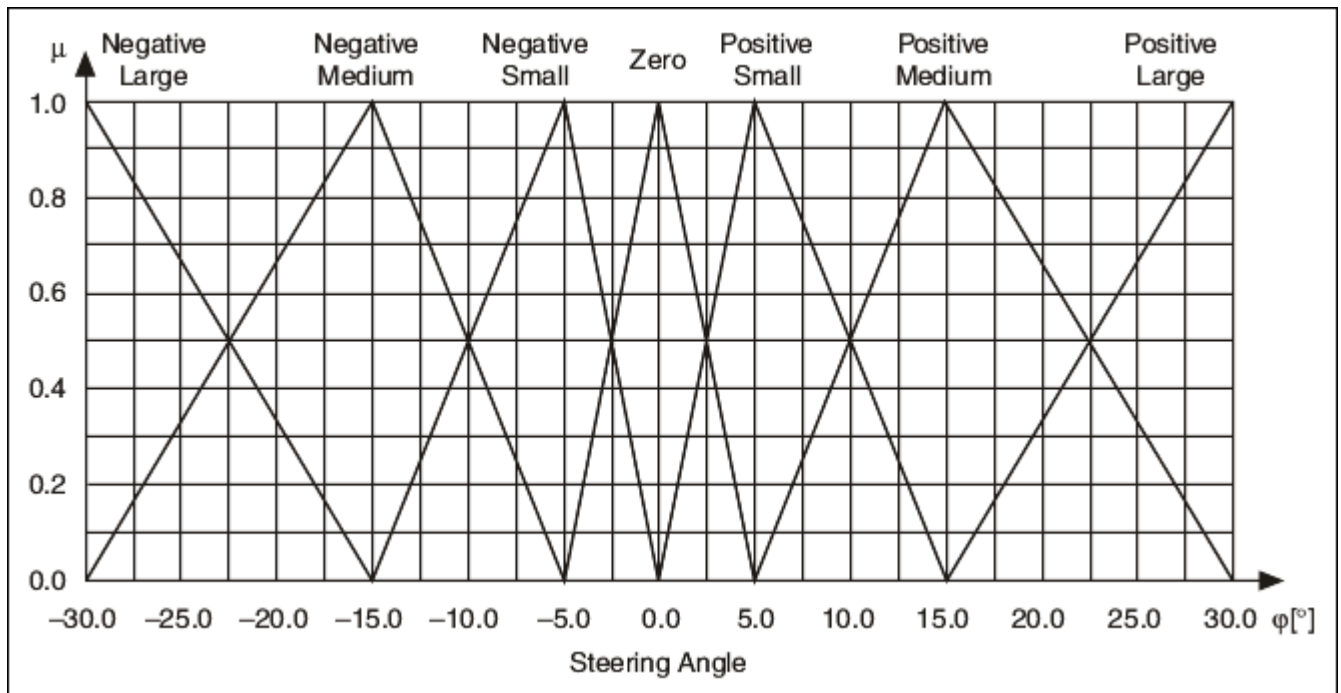
The following image shows the membership functions for Vehicle Position x .



The following image shows the membership functions for Vehicle Orientation β .



The following image shows the membership functions for Steering Angle ϕ .



Defuzzification Methods

Defuzzification is the process of converting the degrees of membership of output linguistic variables within their linguistic terms into crisp numerical values. Consider the following rules:

(1)	IF Vehicle Position x is Center (degree of membership = 0.8)	AND (Minimum)	Vehicle Orientation β is Left Up (degree of membership = 1.0) = 0.8
THEN Steering Angle ϕ is Negative Small			
(2)	IF Vehicle Position x is Right Center (degree of membership = 0.1)	AND (Minimum)	Vehicle Orientation β is Left Up (degree of membership = 1.0) = 0.1
THEN Steering Angle ϕ is Negative Medium			

These two rules specify two non-zero values for the Steering Angle ϕ output linguistic variable:

Negative Medium Negative Small	to a degree of to a degree of	0.1 0.8



NOTE

Fuzzy controllers use an implication method to scale the membership functions of output linguistic variables before performing defuzzification.

A fuzzy controller can use one of several mathematical methods to perform defuzzification: Center of Area (CoA), modified Center of Area (mCoA), Center of Sums (CoS), Center of Maximum (CoM), or Mean of Maximum (MoM). Selecting a defuzzification method depends on the context of the design you want to calculate with the fuzzy controller.

Related Information

- Center of Area (CoA)
- Modified Center of Area (mCoA)
- Center of Maximum (CoM)
- Mean of Maximum (MoM)
- Center of Sums (CoS)

Selecting a Defuzzification Method



RELATED INFORMATION

[Creating Linguistic Variables](#) on page 1790

[Fuzzy Controllers](#) on page 1737

[Specifying a Consequent Implication Method](#) on page 1755

[Creating Membership Functions](#) on page 1791

[Center of Area \(CoA\)](#) on page 1781

[Modified Center of Area \(mCoA\)](#) on page 1804

[Center of Sums \(CoS\)](#) on page 1784

[Center of Maximum \(CoM\)](#) on page 1782

[Mean of Maximum \(MoM\)](#) on page 1803

[Selecting a Defuzzification Method](#) on page 1805

[Center of Area \(CoA\)](#) on page 1781

[Modified Center of Area \(mCoA\)](#) on page 1804

[Center of Maximum \(CoM\)](#) on page 1782

[Mean of Maximum \(MoM\)](#) on page 1803

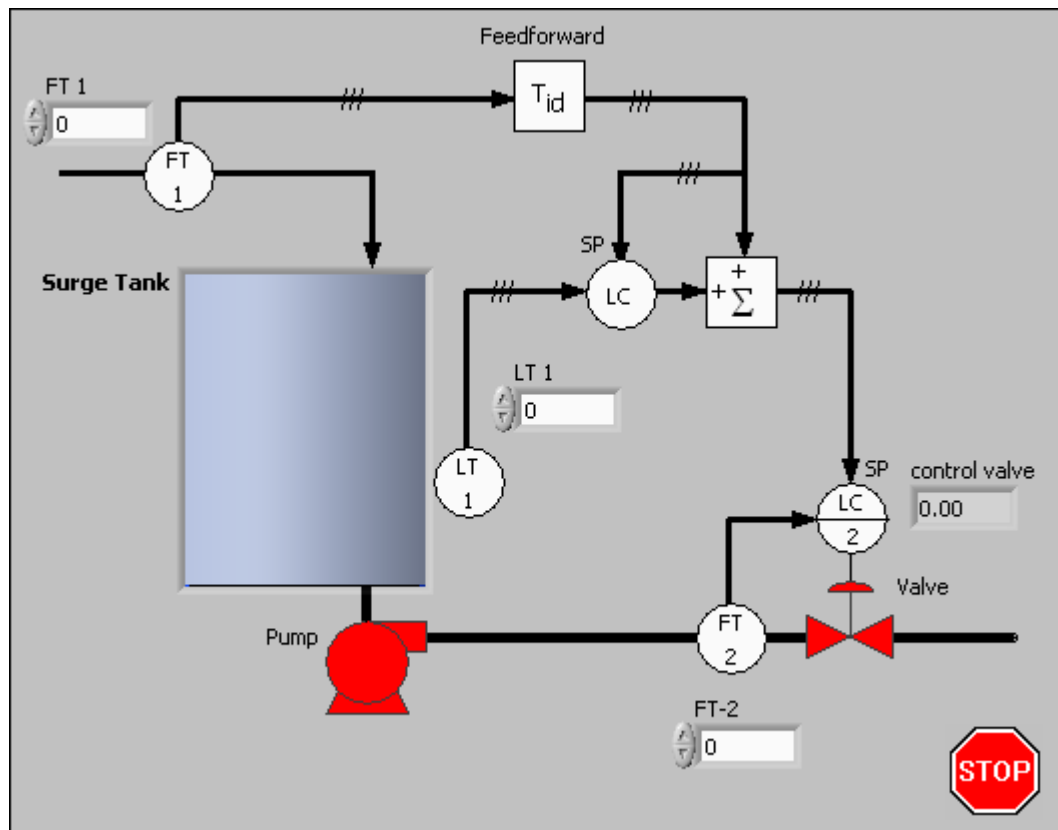
[Center of Sums \(CoS\)](#) on page 1784

[Selecting a Defuzzification Method](#) on page 1805

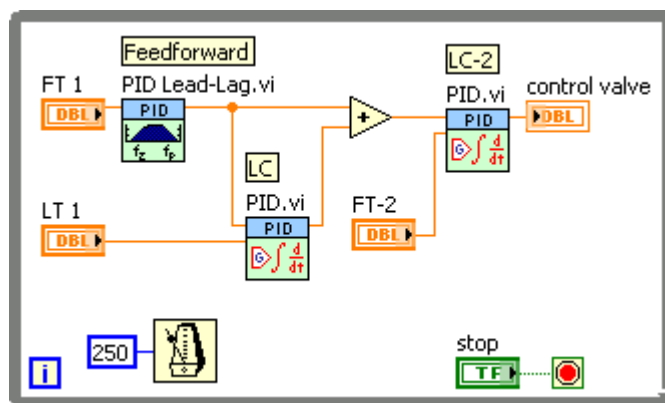
Designing a Control Strategy

You can design a control strategy by sketching a flowchart that includes the physical process and control elements such as valves and measurements. Also include feedback from the process and any required computations. You then can use the PID VIs, combined with the math and logic VIs and functions in LabVIEW, to translate the flowchart into a LabVIEW block diagram.

The following image shows a sample control flowchart:



The following image shows the equivalent LabVIEW block diagram:



In the previous images, **FT** is the flow transmitter, **LT** is the level transmitter, **LC** is the level of control, and **SP** is the setpoint. The only elements missing from this simplified VI are the loop-tuning parameters and the automatic-to-manual switching.

You can handle the inputs and outputs through DAQ devices, FieldPoint I/O modules, GPIB instruments, or serial I/O ports. You can adjust polling rates in real time. Potential polling rates are limited only by your hardware and by the number and graphical complexity of your VIs.

Setting Timing

The PID VI and the PID Lead-Lag VI are time-dependent. A VI can acquire timing information either from a value you supply to the cycle time control, **dt**, or from a time keeper such as those built into the PID VIs. If **dt** is less than or equal to zero, the VI calculates new timing information each time LabVIEW calls it. At each call, the VI measures the time since the last call and uses that difference in its calculations. If you call a VI from a While Loop that uses one of the LabVIEW Timing VIs and functions, you can achieve fairly regular timing, and the internal time keeper compensates for variations. However, the resolution of the Tick Count (ms) function is limited to 1 ms.

If **dt** is a positive value in seconds, the VI uses that value in the calculations, regardless of the elapsed time. National Instruments recommends you use this method for fast loops, such as when you use acquisition hardware to time the controller input or real-time applications. Refer to the example VIs located in the `labview\examples\control\PID` directory for examples of using timing with the PID VIs.



Find related examples

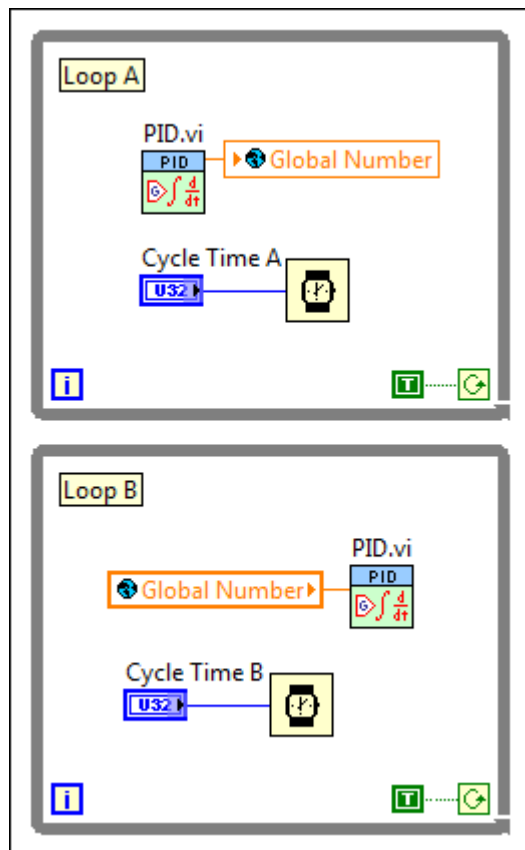


NOTE

If you installed NI-DAQmx, you also can view relevant examples in the `labview\examples\DAQmx\Control\Control.11b` directory.

According to control theory, a control system must sample a physical process at a rate about 10 times faster than the fastest time constant in the physical process. For example, a time constant of 60 seconds is typical for a temperature control loop in a small system. In this case, a cycle time of about 6 seconds is sufficient. Faster cycling does not improve performance. In fact, running all your control VIs too fast can degrade the response time of your LabVIEW application.

All VIs within a loop execute once per iteration at the same cycle time. To run several control VIs at different cycle times and still share data between them, as for example in a cascade, you must separate the VIs into independently timed While Loops. The following block diagram shows a cascade with two independently timed While Loops:

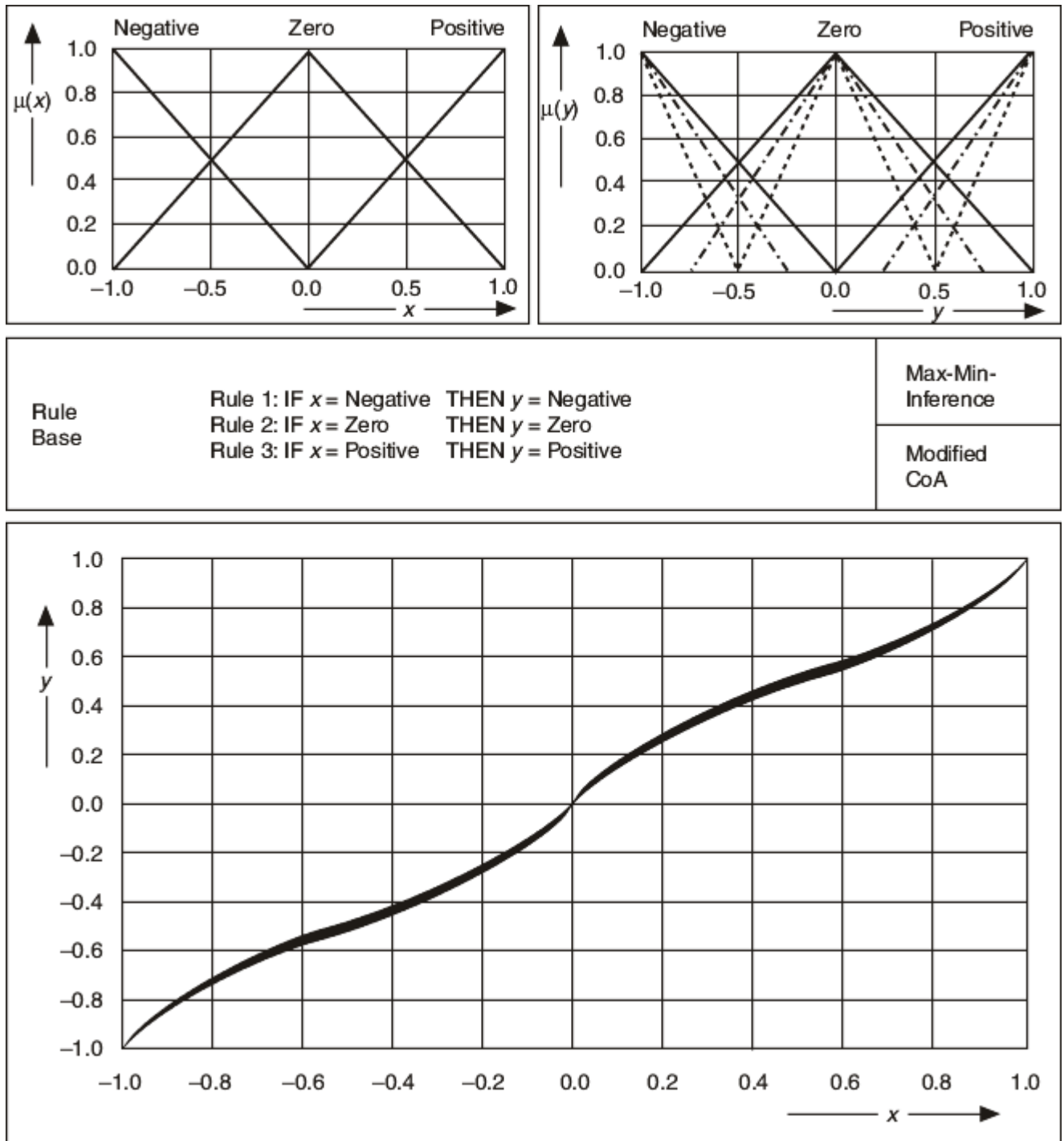


A global variable passes the **output** of the PID VI in **Loop A** to the **process variable** input of the PID VI in Loop B. You can place both While Loops on the same block diagram or in separate VIs. Use additional global or local variables to pass any other necessary data between the two While Loops.

If the front panel of your VI does not contain graphics that LabVIEW must update frequently, the PID VIs can execute at kilohertz (kHz) rates. However, actions such as mouse activity and window scrolling interfere with these rates.

Different Overlapping Degrees of Membership Functions for Output Terms

The following image shows that if all the consequent terms are equal in width, the overlapping degree of the membership functions for the consequent terms has no significant influence on the controller characteristic.



RELATED INFORMATION

[Creating Membership Functions](#) on page 1791

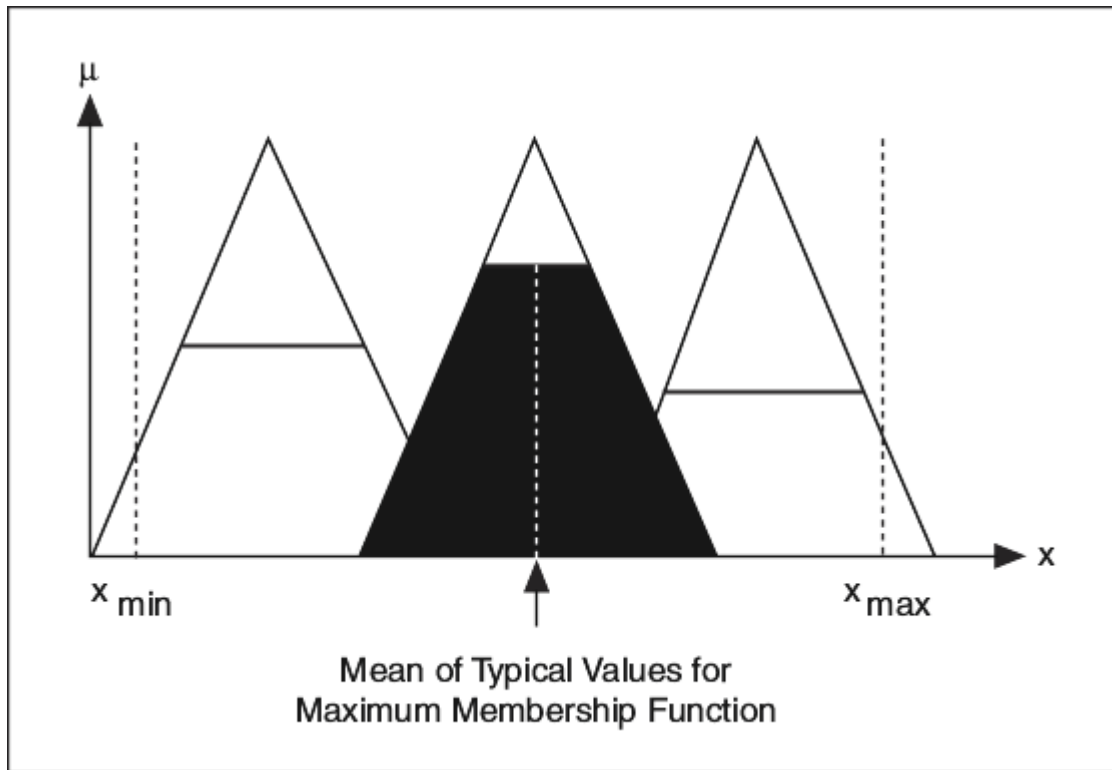
[I/O Characteristics of Fuzzy Controllers](#) on page 1744

Mean of Maximum (MoM)

Use the Mean of Maximum (MoM) [defuzzification method](#) for pattern recognition applications. This defuzzification method calculates the most plausible result. Rather than

averaging the degrees of membership of the output *linguistic terms*, the MoM defuzzification method selects the typical value of the most valid output linguistic term.

The following image illustrates the MoM defuzzification method.



Modified Center of Area (mCoA)

Because the *Center of Area (CoA)* defuzzification method evaluates the area under the scaled *membership functions* only within the range of the output *linguistic variable*, the resulting crisp output values cannot span the full range. To solve this problem, use the modified Center of Area defuzzification method.

The modified Center of Area defuzzification method is similar to the Center of Area defuzzification method. However, the fuzzy logic controller considers the full area under the scaled membership functions, even if this area extends beyond the range of the output variable. The fuzzy logic controller uses the following equation to calculate the geometric center of the full area under the scaled membership functions.

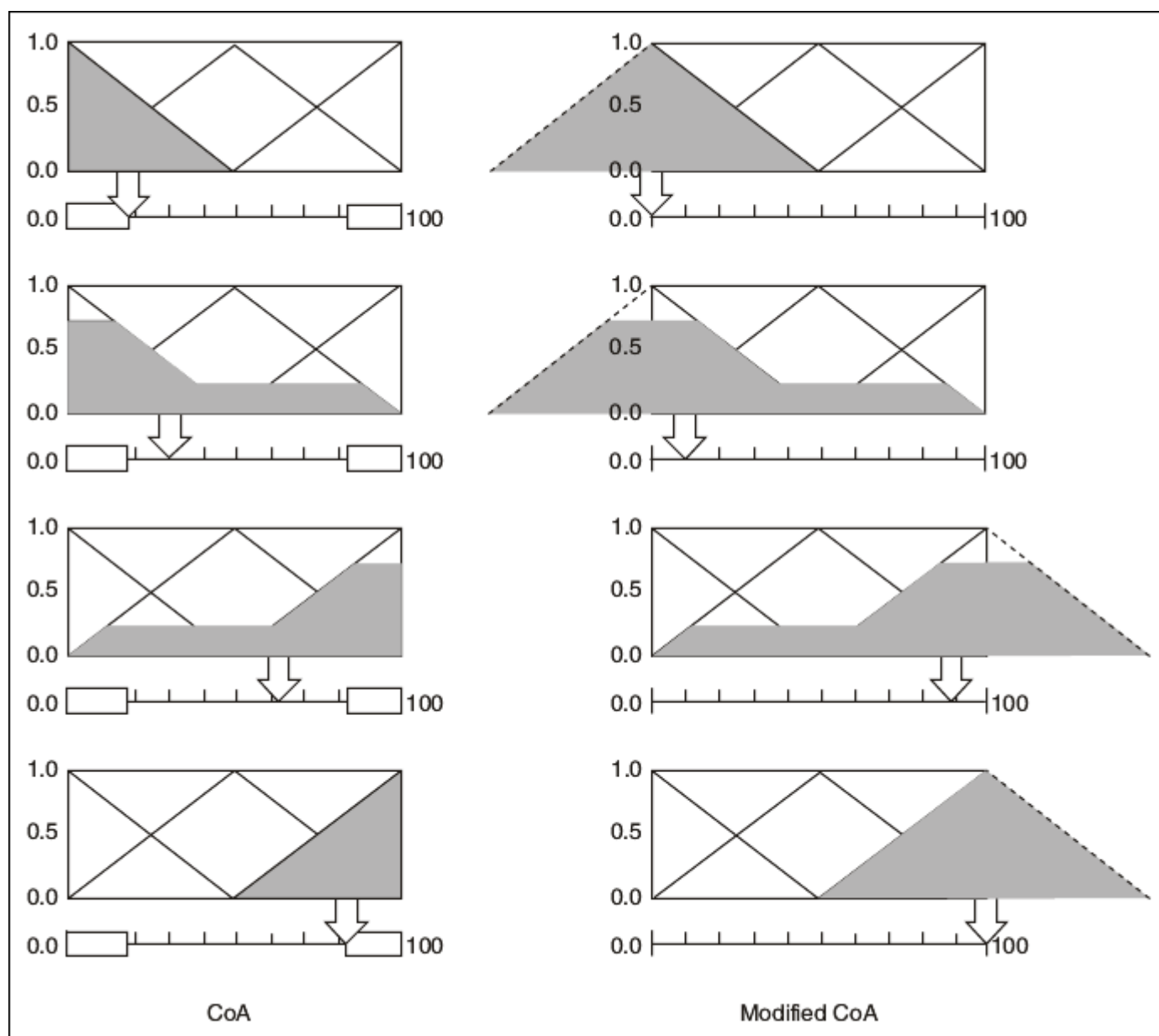
$$mCoA = \frac{\int f(x) * x \, dx}{\int f(x) \, dx}$$

where mCoA is the modified center of area.

The interval of integration is between the minimum membership function value and the maximum membership function value. Note that this interval might extend beyond the range of the output variable.

Comparing CoA and Modified CoA Defuzzification Methods

The following image illustrates the difference between the Center of Area and modified Center of Area defuzzification methods.



Selecting a Defuzzification Method

In decision support systems, the choice of the *defuzzification method* depends on the context of the decision you want to calculate with the fuzzy controller. For quantitative decisions like project prioritization, apply the *Center of Maximum (CoM)* method. For qualitative decisions, such as an evaluation of credit worthiness, *Mean of Maximum (MoM)* is the correct method.

An important aspect of a defuzzification method is the continuity of the output signal. Consider a fuzzy system with a complete rule base and overlapping membership functions.

A defuzzification method is continuous if an arbitrary small change of an input value never causes an abrupt change in the output signal.

In this respect, the defuzzification methods CoM and [Center of Area \(CoA\)](#) are continuous because, assuming overlapping output membership functions, the best compromise does not jump to a different value with a small change to the inputs. The defuzzification method MoM, however, is discontinuous because an arbitrary small change in the input values of the fuzzy system can cause the output value to switch to another, more plausible result.

Using CoA or CoM as the defuzzification method results in continuous [controller characteristic](#) functions, especially within those intervals of the input values in which two or more rules are active simultaneously. This behavior results from the averaging character of the defuzzification methods.

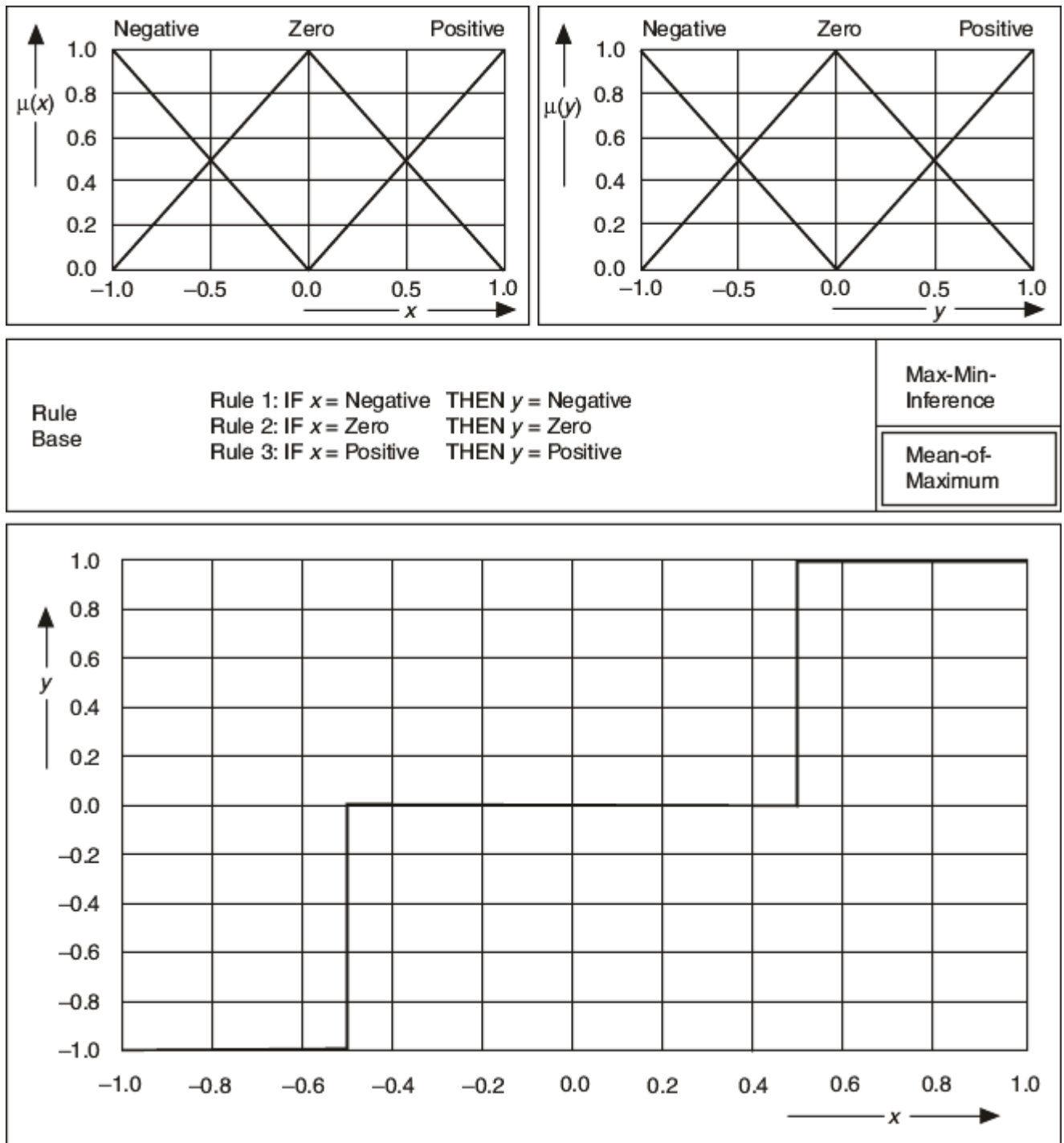
The following table compares the different defuzzification methods based on various assessment criteria.

Assessment Criteria	Method			
Center of Sums (CoS)	Center of Maximum (CoM)	Mean of Maximum (MoM)		
Linguistic Characteristic	Best Compromise	Best Compromise	Best Compromise	Most Plausible Result
Fit with Intuition	Implausible with varying membership function shapes and strong overlapping membership functions	Implausible with varying membership function shapes and strong overlapping membership functions	Good	Good
Continuity	Yes	Yes	Yes	No
Computational Effort	Very High	Medium	Low	Very Low
Application Field	Closed-Loop Control, Decision Support, Data Analysis	Closed-Loop Control, Decision Support, Data Analysis	Closed-Loop Control, Decision Support, Data Analysis	Pattern Recognition, Decision Support, Data Analysis

Using the Mean of Maximum (MoM) Defuzzification Method

When you use the [Mean of Maximum \(MoM\) defuzzification method](#), you calculate the most plausible result. In other words, the fuzzy controller uses the typical value of the

consequent term of the most valid rule as the crisp output value. This behavior results in stepped output characteristics, as shown in the following image.



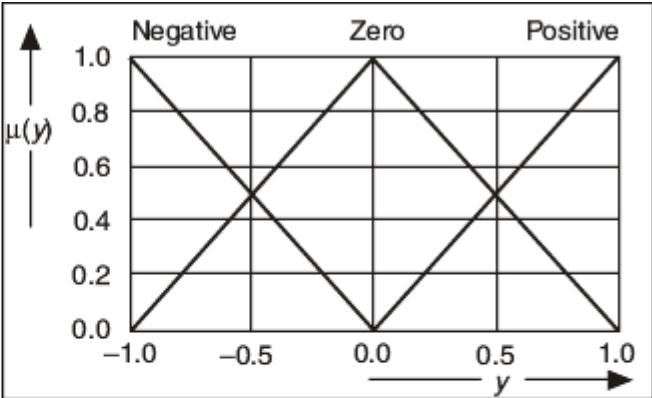
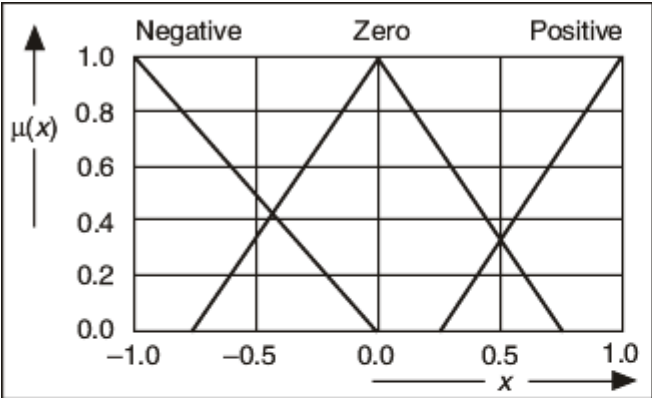
The previous image shows entirely overlapping *membership functions* for input and output terms.

Changing the Rule Base

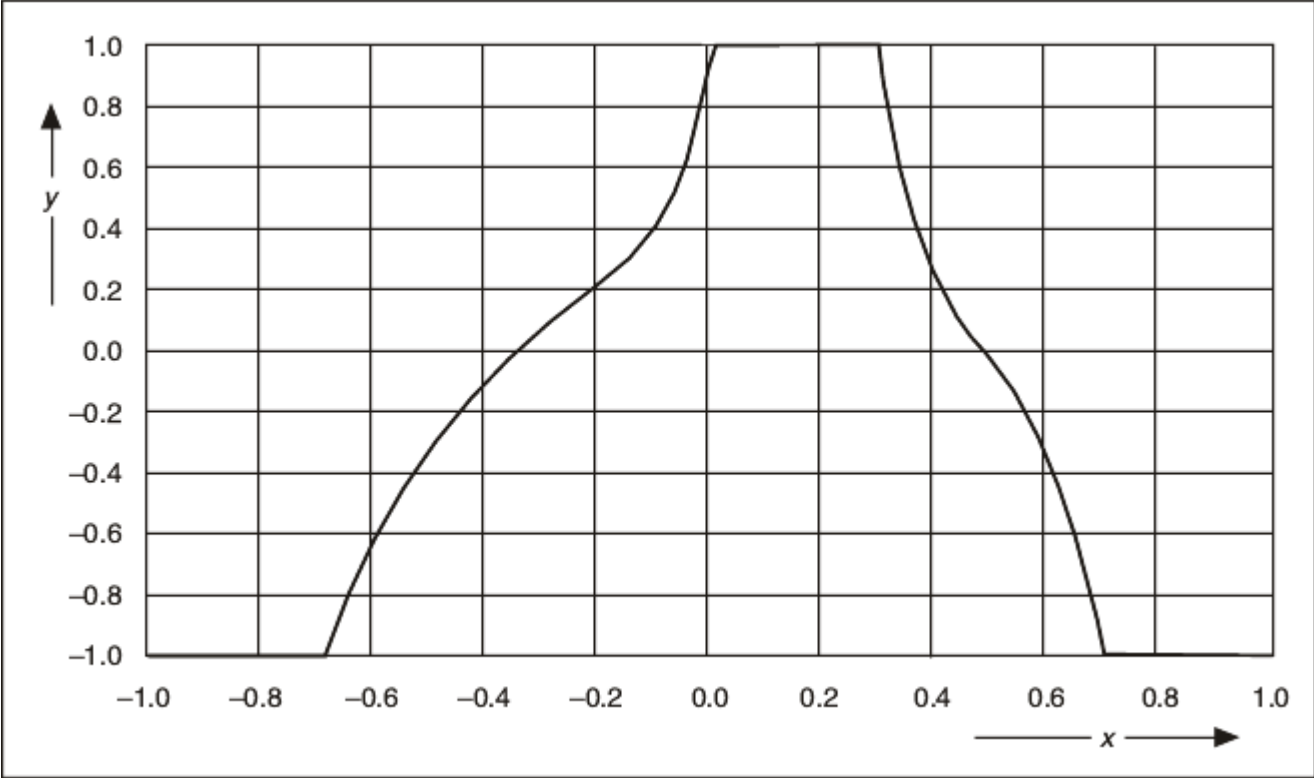
The *rule base* itself has the biggest influence on the controller characteristic. The rule base determines the principal functionality of the controller.

The following image illustrates how the controller characteristic changes if you change the rule base of the previous example to include the following rules:

Rule 1:	IF	x = negative	THEN	y = negative
Rule 2:	IF	x = zero	THEN	y = positive
Rule 3:	IF	x = positive	THEN	y = negative

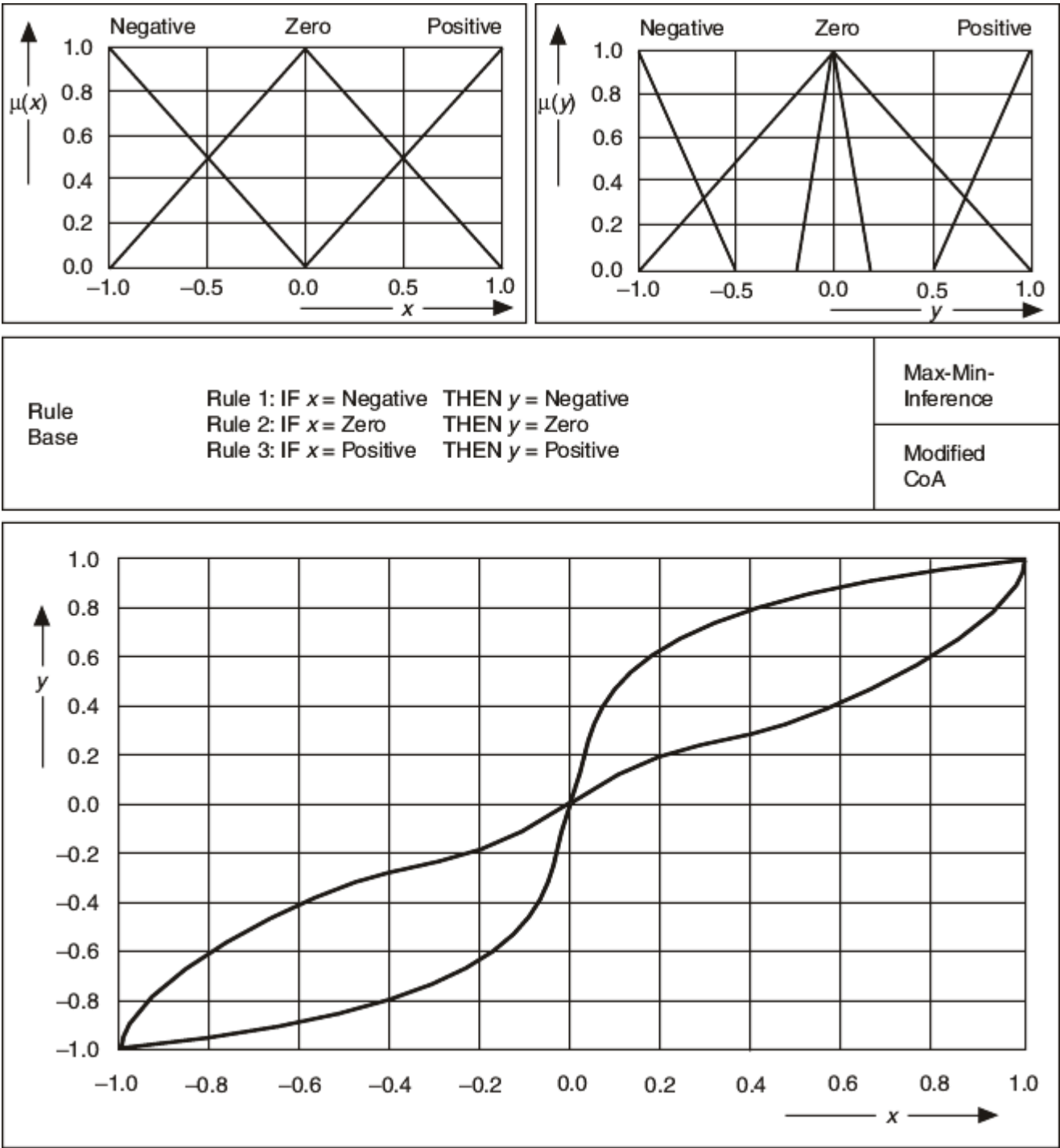


Rule Base	Rule 1: IF x = Negative THEN y = Negative Rule 2: IF x = Zero THEN y = Positive Rule 3: IF x = Positive THEN y = Negative	Max-Min-Inference
		Modified CoA



Wide and Small Membership Functions for Output Terms

If you want to significantly influence the controller characteristic, use output terms that membership functions model with equally distributed typical values but different scopes of influence instead. The different terms have different areas and thus different weights with respect to the defuzzification process. A wide output term has more influence on the inference result than a small neighboring output term. The following image demonstrates this effect.



RELATED INFORMATION

- [I/O Characteristics of Fuzzy Controllers](#) on page 1744
- [Creating Membership Functions](#) on page 1791
- [Defuzzification Methods](#) on page 1797

Distributed System Manager Overview

Use the Distributed System Manager to complete the following tasks:

- Create shared or network variables.
- Monitor shared or network variables.
- Create and monitor processes.
- Create and monitor I/O servers.
- Manage security.
- Manage aliases.
- Start and stop the Shared Variable Engine.
- Force I/O variable values.
- View and clear scan engine faults.
- View and set scan engine modes.
- Monitor real-time target resources.
- Monitor alarms and events.
- Acknowledge alarms and events.
- Configure alarms for shared variables.
- Monitor historical trends.
- Log historical trends.
- Synchronize time for networked computers.

**NOTE**

You must use an alternate method to start or stop the Shared Variable Engine on Windows 7/Vista.

Select **Start»National Instruments»NI Distributed System Manager** to launch the Distributed System Manager. In LabVIEW, you also can launch the Distributed System Manager by selecting **Tools»Distributed System Manager**.



RELATED INFORMATION

[Creating, Modifying, or Deleting Shared or Network Variables](#) on page 1837

[Monitoring Variables with the Watch List View](#) on page 1834

[Processes](#) on page 1841

[I/O Servers](#) on page 1817

[Shared Variable Engine](#) on page 1819

[Forcing I/O Variable Values \(NI Scan Engine\)](#) on page 1815

[Viewing and Clearing Faults \(NI Scan Engine\)](#) on page 1822

[Viewing and Setting the Scan Engine Mode \(NI Scan Engine\)](#) on page 1821

[Monitoring Alarms and Events](#) on page 1833

[Acknowledging Alarms and Events](#) on page 1822

[Configuring Alarms for Shared Variables \(DSC Module\)](#) on page 1832

[Synchronizing Time for Networked Computers](#) on page 1819

[Systems](#) on page 1820

[Processes](#) on page 1841

[Shared or Network Variables](#) on page 1818

[Monitoring Variables with the Watch List View](#) on page 1834

[CPU/Memory Probe View](#) on page 1824

[I/O Variable Probe View](#) on page 1825

[Process Probe View](#) on page 1827

[Scan Engine Probe View](#) on page 1827

[Variable Probe View](#) on page 1829

[Web Server Probe View](#) on page 1830

[Web Service Probe View](#) on page 1831

Distributed System Manager Components

Learn about the components of the Distributed System Manager.

- **System tree** —Displays all available objects on the network. The system tree appears as the primary navigation pane in the System Manager workspace.
- **System tree object** —Refers to all types of objects that might appear in the system tree. System tree object can refer to a system, process, variable, or additional object types installed with modules and toolkits.

- **Watch list** — Monitors a single variable or multiple variables contained within a system or process.
- **Auto View** —Displays the **Probe** view for any system tree object that you select. If you select multiple objects, this view displays a **Watch List** view that contains all of the selected objects.
- **Probe views** —Display information and configuration options for the following components:
 - CPU/Memory
 - I/O Variable
 - Process
 - Scan Engine
 - Variable
 - Web Server
 - Web Service

File Menu

The **File** menu contains items for managing System Manager workspaces. Workspaces display information about all *shared or network variables* that are currently deployed, including the URL, current value, data type, timestamp information, and any error or quality messages. Quality messages list errors and other conditions that apply to the variable.

- **New Workspace** —Creates a new workspace. When you select this menu item, the System Manager removes all variables from the **Watch List** view and resets the workspace layout.
- **Open Workspace** —Opens a saved workspace.
- **Save Workspace** —Saves the current workspace with the .smw extension.
- **Settings** —Displays the *Settings* dialog box where you can customize workspace layout, set the data update rate, and specify the System Manager to save its workspace between sessions.

Actions Menu

The **Actions** menu contains items for starting and stopping the Shared Variable Engine, refreshing your workspace, and logging into a system.

- **Start Local Shared Variable Engine** —Restarts the *Shared Variable Engine*.



NOTE

You must use an alternate method to start the Shared Variable Engine on Windows 7/Vista.

- **Stop Local Shared Variable Engine** —Shuts down the *Shared Variable Engine*.

**NOTE**

You must use an alternate method to stop the Shared Variable Engine on Windows 7/Vista.

- **Register Computer** —Displays the **Register Computer** dialog box, which you can use to register remote computers.
- **Alias Manager** —Displays the Alias Manager dialog box, which you can use to configure aliases for local computers and IP addresses.
 - (DSC Module) The Shared Variable Engine uses aliases when running custom I/O servers that refer to other variables in the system.
- **Log In** —Displays the NI Security Login dialog box.
- **Log Out** —Logs out of the session configured in the **NI Security Login** dialog box.
- **Configure Security** —Displays the NI ACL Configuration dialog box, which you can use to update security settings.
- **Add System to My Systems** —Displays the Add Systems to My Systems dialog box that allows you to add a system to **My Systems** on the system tree.

View Menu

The **View** menu contains the options to display certain windows in the workspace and access a system tree object that does not appear in the system tree.

- **Auto View** —Displays the Probe view for any system tree object that you select. If you select multiple objects, this view displays a Watch List that contains all of the selected objects.
- **Alarms and Events** —Displays the Alarms and Events view for any systems located within **My Systems** on the system tree.
- **Probe Location** —Displays the Probe Location dialog box for a system, process, or variable. You can enter the location of objects that appear in the system tree, as well as those that do not.
- **Configure Time Servers** — (DSC Module) Displays the Shared Variable Engine Time Server Configuration dialog box. You can use the dialog box to synchronize the time of machines on the network with a central time server.
- **New Watch List** —Creates a blank **Watch List** view. Drag a system tree object into this view to add it to the **Watch List** view.
- **Refresh** —Updates the contents of the system tree and **Watch List** views.

- **New Historical Trend** — (DSC Module) Creates a blank **Historical Trend** view. Drag a system tree pane object into this view to monitor the historical trend of this object.



RELATED INFORMATION

[Distributed System Manager Overview](#) on page 1811

[Distributed System Manager Overview](#) on page 1811

[Distributed System Manager Overview](#) on page 1811

[Monitoring Variables with the Watch List View](#) on page 1834

[Alarms and Events View](#) on page 1823

[Systems](#) on page 1820

[Processes](#) on page 1841

[Shared or Network Variables](#) on page 1818

Help Menu

The **Help** menu contains items to explain and define System Manager features and provide System Manager documentation.

- **System Manager Overview** —Displays the NI Distributed System Manager Help. Use this help file for information about creating and managing shared or network variables and other [system tree objects](#).



NOTE

Refer to the LabVIEW Help, NI Measurement Studio Help, or LabWindows/CVI Help for more information about using variables created with the System Manager in your applications.

- **About System Manager** —Displays general information about the current installation, such as the version number.

Forcing I/O Variable Values (NI Scan Engine)

You can use an [I/O variable probe](#) to force I/O variable values. Forcing an I/O variable causes the input or output and all associated aliases of the variable to assume the forced value until you unforce the value. If an input from hardware returns an error, and the associated I/O variable is forced, System Manager does not display the error. However, if the error results in a [fault](#), LabVIEW registers the fault. If an output returns an error, System Manager displays the error.



NOTE

If you link several I/O variables in a chain of aliases and force one of the I/O variables in the chain, the forced value applies to all I/O variables in the chain.

Enabling I/O Variable Forcing

Complete the following steps to enable and disable I/O variable forcing on a target with the *NI Scan Engine* installed.

1. Click a scan-enabled target in the system tree to display a target probe in the **Auto View**. You also can right-click a target and select **Probe** to display a target probe in a separate window.
2. The *Scan Engine* page of a real-time target probe displays the **Forcing Enabled** LED indicator, which indicates whether I/O variable forcing is enabled on the controller.
3. Use the **Enable Forcing** button to enable forcing. Use the **Disable Forcing** button to disable forcing and unforce all forced I/O variable values. Use the **Clear Forcing** button to clear forcing on all I/O variables.



NOTE

If you disable forcing and then enable forcing again, all previously-forced I/O variables return to their last forced value, unless you have cleared those values using the **Clear Forcing** button.

Forcing I/O Variable Values

Complete the following steps to force and unforce I/O variable values.

1. Click an I/O variable in the system tree to display an I/O variable probe in the **Auto View** or right-click an I/O variable and select **Probe** to display an I/O variable probe in a separate window.
2. In the **New Value** section of the I/O variable probe, enter or select the new value for the I/O variable.
3. Click the **Force** button to force the I/O variable to the value you specified.
4. Click the **Unforce** button to unforce the I/O variable value.



NOTE

Individual I/O variable forcing applies only if forcing is enabled on the target. Use the **Enable Forcing** and **Disable Forcing** buttons to enable and disable forcing on the target.

Batch Forcing I/O Variable Values

Complete the following steps to perform batch forcing on a set of I/O variables.

1. Disable forcing on the target.
2. Force each I/O variable to the desired value.
3. Enable forcing on the target.

I/O Servers

An I/O server is an application that communicates with and manages input and output devices such as programmable logic controllers (PLCs), remote input/output devices, remote Shared Variable Engines, and data acquisition (DAQ) plug-in devices. These I/O servers read selected input items and write to the selected output items on demand.

Shortcut Menu

Right-click an I/O server on the system tree to display the following options.

- **Watch List** —Creates a new *Watch List* view that contains the single *variable* or multiple variables contained within the selected *system* or *process*.
- **Probe** —Displays the **I/O Server Probe** view for the selected system.
- **Edit I/O Server** —Configures the selected I/O server.
- **Remove I/O Server** —Removes the selected I/O server.

Observing Trend View of Variables

Use the *Variable Probe* view in the System Manager to monitor the value of a *variable* over time using the trend view graph.

Complete the following steps to monitor the value of a variable over time using the trend view graph.

1. Right-click a variable in the *system tree*.
2. Select **Probe** from the shortcut menu to open a new **Variable Probe** view for the selected variable.
3. Place a checkmark in the **Show Trend** checkbox to display the trend view graph.

Programmatically Accessing Published Values

You can access *system tree objects* such as *shared or network variables* from outside the System Manager environment. Use the network path for a system tree object to access that object from another environment.

The network path to shared or network variables consists of the system name, the name of the process in which the shared variable resides, and the shared variable name: `\system\process\variable`. For example, the network path `\system1\myprocess\myvariable` identifies a shared variable named `myvariable` in the process named `myprocess` on a system named `system1`.

You can determine the network path using the [Probe](#) view for any system tree object. The **Location** field displays the network path for the selected object.



NOTE

Refer to the LabVIEW Help, NI Measurement Studio Help, or LabWindows/CVI Help for more information about accessing shared or network variables created or modified with the System Manager from other applications.

Saving and Restoring a System Manager Workspace

When you [customize the System Manager workspace](#), you can save the customized workspace and restore it in a later System Manager session.

Complete the following steps to save and restore a customized System Manager workspace.

1. Customize the System Manager workspace.
2. Select **File»Save Workspace** to save your workspace as a .smw file.
3. To restore a workspace in a later session, select **File»Open Workspace** and navigate to the .smw file you want to restore.

Shared or Network Variables

LabVIEW uses the term shared variable and Measurement Studio and LabWindows/CVI use the term network variable to describe software items that exist on a network and can communicate between programs, applications, remote computers, and hardware through the [Shared Variable Engine \(SVE\)](#). Variables can represent a value or an I/O point. The SVE uses the NI Publish-Subscribe Protocol (NI-PSP) data transfer protocol to write data and allow users to read live data. NI-PSP is a proprietary technology that provides fast and reliable data transmission for large and small applications and is installed as a service on the computer when you install LabVIEW, Measurement Studio, or LabWindows/CVI.

Use the System Manager to [create, modify, or delete shared or network variables](#). You also can [monitor shared or network variables](#) using the **Watch List** view.

Shortcut Menu

Right-click a variable on the [system tree](#) to display the following options.

- **Watch List** —Creates a new [Watch List](#) view that contains the selected variable.
- **Probe** —Displays the [Variable Probe](#) view for the selected system.
- **Edit Variable** —Displays the [Shared Variable Properties](#) dialog box, which you can use to configure the selected variable.
- **Remove Variable** —Removes the selected variable from the owning process.

Shared Variable Engine

The Shared Variable Engine (SVE) uses the NI Publish-Subscribe Protocol (NI-PSP) to enable the data transfer of [shared or network variables](#) through a network. The SVE manages the use and connectivity of shared or network variables on single or multiple [systems](#).

Stopping Shared Variable Engine

Select **Actions»Stop Local Shared Variable Engine** to shut down the SVE. You can [create, configure, and remove variables](#) in the System Manager after you shut down the SVE. While the SVE is shut down, variable values do not update, and the SVE does not log alarms for variables.



NOTE

You must use an alternate method to stop the Shared Variable Engine on Windows 7/Vista.

Starting Shared Variable Engine

Select **Actions»Start Local Shared Variable Engine** to restart the SVE. The SVE is always running after you install LabVIEW, LabWindows/CVI, or Measurement Studio unless you select **Actions»Stop Local Shared Variable Engine** to manually shut down the SVE.



NOTE

You must use an alternate method to start the Shared Variable Engine on Windows 7/Vista.



RELATED INFORMATION

[Actions Menu](#) on page 1813

Synchronizing Time for Networked Computers

Complete the following steps to set time synchronization.

1. Select **View»Configure Time Servers** to display the [Shared Variable Engine Time Server Configuration](#) dialog box.
2. Notice the **Time server search order** list. Any computer that is running the time synchronization service can serve as a time server or a time client. The primary time server is the first computer listed in the **Time Server Search Order** list. If no computer is set as a primary time server, the computer synchronizes to itself.
3. To add a computer to the **Time server search order** list, click the **Add** button to display the [Select Computer](#) dialog box.

4. If you know the name of the computer you want to add, you can type the name into the **Host** text box. If you do not know the exact name of the computer, you can browse for the computer using the network tree in the **Machines** list. To remove a computer from the **Time server search order** list, select the computer name and click the **Remove** button.
5. To change the order in which computers search for a time synchronization server, select the computer name and click the **Up** or **Down** button.
6. Use **Sleep time (seconds)** to set how long each computer waits between each synchronization. Set the primary time synchronization server sleep time to 60 seconds. If the primary server is off-line for some reason, a computer scheduled to synchronize automatically seeks out the second computer on the synchronization server list. At the time of the next synchronization, the computer first looks for the primary server before seeking a secondary synchronization server.
7. Click the **OK** button.
8. Repeat steps 4 through 7 for all computers on the network that you want to synchronize to make sure that the order of search for time servers is the same for all the computers, including the primary time synchronization server.

Systems

Systems are local computers or LabVIEW Real-Time targets available through the System Manager. Systems available on your local subnetwork appear by default. Select **Actions»Add System to My Systems** to access systems outside your local subnetwork that do not appear in the [system tree](#).

Shortcut Menu

Right-click a system on the system tree to display the following options.

- **Add to My Systems** —Moves a selected system listed in **Network Items** to **My Systems** on the system tree.
- **Remove From My Systems** —Removes a selected system listed in **My Systems**.
- **Watch List** —Creates a new [Watch List](#) view that contains the single [variable](#) or multiple variables contained within the system or [process](#) that you select.
- **Probe** —Displays the **Systems Probe** view, which displays the location address for the selected system.
- **Add Process** —Displays the Add Process dialog box.
- **Register Computer** — Registers the selected system.
- **Unregister Computer** — Unregisters the selected system.

VI States

Requires: Real-Time Module

Select a real-time target to display a target probe in the **Auto View**. You also can right-click the target and select **Probe** to display the probe in a separate window. Click the **VI States** tab of a real-time target probe to display this view. Use this view to start, stop, and monitor VIs on the real-time target.

This view includes the following components:

- **Port** —Specifies the TCP/IP port at which the VI server listens for requests on the real-time target. Right-click the real-time target in the *Project Explorer* window and select **Properties** from the shortcut menu to display the **RT Target Properties** dialog box. Select **VI Server** from the **Category** list to display the **VI Server** page, where you can configure the TCP/IP port for the VI Server.
- **Update Interval** —Sets the minimum time that the server waits between updates. The actual update interval might exceed the time you specify if higher priority tasks are running on the real-time target.
- **Show Top Level VIs Only** —Filters all subVIs from the list of real-time target VIs.
- **Start Monitoring** —Starts monitoring VIs on the real-time target.
- **Stop Monitoring** —Stops monitoring VIs on the real-time target.
- **Start VI** —Starts an idle VI that you select from the list of real-time target VIs.
- **Stop VI** —Stops a VI that you select from the list of real-time target VIs.

Viewing and Setting the Scan Engine Mode (NI Scan Engine)

You can use the *Scan Engine* page of a target probe to view and set the *scan engine mode* of a target with the *NI Scan Engine* installed.

Complete the following steps to view and set the scan engine mode.

1. Click a scan-enabled target in the system tree to display a target probe in the **Auto View**. You also can right-click a target and select **Probe** to display a target probe in a separate window.
2. The Scan Engine page of a target probe displays scan engine modes under the **Scan Engine Enabled** LED indicator and highlights the current scan engine mode with a green LED indicator.
3. Use the **Change to Config** button to set the scan engine mode to **Config Mode** or the **Change to Active** button to set the scan engine mode to **Active Mode**. You cannot enter **Active Mode** directly from **Fault Mode**. You must first *clear* all major *faults* on the target.



NOTE

Entering **Config Mode** from **Fault Mode** automatically converts all major faults to minor faults. You cannot enter fault mode manually. The target enters fault mode only when a *major* or *unrecoverable* fault occurs.

Viewing and Clearing Faults (NI Scan Engine)

You can view and clear **faults** on a target with the **NI Scan Engine** installed. Complete the following steps to view and clear faults.

1. Click a scan-enabled target in the system tree to display a target probe in the **Auto View**. You also can right-click a target and select **Probe** to display a target probe in a separate window. The Scan Engine page of the target probe displays the faults active on the target in a table that includes the fault code and description. The **Fault Present** LED indicator appears black when no fault is present, orange when only **minor** faults are present, and red when **major** or **unrecoverable** faults are present.
2. Click the **Clear Faults** button to clear all faults on the target.
3. To ensure that you successfully cleared all faults on the target, check the **Fault Present** LED indicator.

Alarms and Events

Acknowledging Alarms and Events

You can acknowledge alarms and events that the **Alarms and Events** view displays by using the **Acknowledge Selected Alarms and Events** or **Acknowledge All Alarms and Events** dialog box.



NOTE

When you acknowledge an alarm, the alarm appears in purple as acknowledged in the **Alarms and Events** view. When the alarm is no longer in the alarm state, the alarm disappears from the **Alarms and Events** view. When you acknowledge an event, the event disappears from the **Alarms and Events** view.

Complete the following steps to acknowledge alarms or events by using the **Alarms and Events** view.

1. Select the alarms and events that you want to acknowledge. You can press the <Ctrl> key and select multiple alarms and events.
2. Right-click a selected alarm or event and select **Acknowledge** from the shortcut menu to display the **Acknowledge Selected Alarms and Events** dialog box.
3. (Optional) Enter comments in the **Comments** text box.



NOTE

Comments apply to alarms only.

4. Click the **OK** button to acknowledge the selected items and close the dialog box.

**NOTE**

To acknowledge all the alarms and events at one time, you can right-click in this view and select **Acknowledge All** from the shortcut menu to display the **Acknowledge All Alarms and Events** dialog box. Then, complete steps 3 and 4 to acknowledge all alarms and events in this view.

Alarm Probe View

Requires: DSC Module

In the system tree pane, right-click an alarm and select **Probe** to display this view. This view shows the status and properties of the selected alarm.

This view includes the following components:

- **Location** —Displays the location of the alarm.
- **Status** —Displays the status of the alarm.
- **Priority** —Displays the priority of the alarm.
- **Level** —Displays the level of the alarm.
- **Description** —Displays the description of the alarm.

Alarms and Events View

Requires: NI Distributed System Manager (Windows)

Select **View»Alarms and Events** to display or hide the **Alarms and Events** view. The NI Distributed System Manager supports only one **Alarms and Events** view.

Use the **Alarms and Events** view to *monitor* and *acknowledge* alarms and events. You can view alarms and events of other processes by using the *Select Processes* dialog box. By default, the **Alarms and Events** view displays alarms and events of systems in **My Systems**. You can *add a system* to **My Systems** to access systems that do not appear in the system tree pane.

**NOTE**

You must have the LabVIEW Datalogging and Supervisory Control (DSC) Module installed to monitor and acknowledge alarms.

This view includes the following components:

- **Select Processes** —Displays the *Select Processes* dialog box, which you use to select multiple processes from the system tree pane. The **Alarms and Events** view then displays alarms and events that these processes generate.
- **Show Alarms** —Specifies whether to display alarms in the **Alarms and Events** view. By default, this checkbox contains a checkmark.

- **Show Events** —Specifies whether to display events in the **Alarms and Events** view. By default, this checkbox contains a checkmark.
- **Acknowledge** —Displays the *Acknowledge Selected Alarms and Events* dialog box, which you use to acknowledge alarms and events you select in the **Alarms and Events** view.
- **Acknowledge All** —Displays the *Acknowledge All Alarms and Events* dialog box, which you use to acknowledge all alarms and events in the **Alarms and Events** view.
- **Filter** —Displays the *Filter Options* dialog box, which you use to display alarms and events that meet certain criteria.
- **Columns** —Displays the **Show Columns** dialog box, which you use to show or hide columns in the **Alarms and Events** view.

Alerts Probe View

Requires: Real-Time Module

Select a real-time target to display a target probe in the **Auto View**. You also can right-click the target and select **Probe** to display the probe in a separate window. Click the **Alerts** tab of a real-time target probe to display this view. Use this view to configure and monitor alerts on the real-time target.

This view includes the following components:

- **Log Alert when memory usage is above** —Enables alert logging when memory usage on the real-time target exceeds the value you specify.
- **Log Alert when CPU usage is above** —Enables alert logging when the real-time target CPU usage exceeds the value you specify.
- **Log Alert when VI changes state** —Enables alert logging when a VI listed in the *VI States* view changes state.
- **Recent Alerts** —Lists recently-logged alerts.
- **Clear** —Removes all entries from the **Recent Alerts** list.
- **Save** —Saves the **Recent Alerts** list as a text file on the local computer.

CPU/Memory Probe View

Requires: NI Scan Engine

Select a real-time target to display a target probe in the **Auto View**. You also can right-click the target and select **Probe** to display the probe in a separate window. Click the **CPU/Memory** tab of a real-time target probe to display this view. Use this view to monitor CPU and memory usage on the target.

This view includes the following components:

- **CPU N Usage Chart (%)** —Tracks utilization of CPU N over time, as a percentage of capacity.
- **Memory Usage Chart (MB)** —Tracks target memory usage over time.
- **CPU N Statistics** —Includes current CPU usage statistics including total CPU usage, CPU usage by priority level, and CPU usage dedicated to Timed Structures and interrupt service routines (ISRs). Each value represents a percentage of total capacity.
- **Memory Statistics** —Displays current memory usage data.
 - **Allocated** —The amount of memory, in MB, currently allocated on the target.
 - **Available** —The amount of memory, in MB, currently available on the target.
 - **Contiguous** —The largest contiguous block of available memory on the target, in MB.
 - **Total** —The total amount of memory available to the operating system, in MB.
 - **Usage** —The percentage of total memory currently allocated on the target.

Date Time Probe View

Requires: NI Distributed System Manager

Select a system from the system tree to display the system in the **Auto View**. You also can right-click the system and select **Probe** to display the **Probe** view in a separate window. Click the **Date Time** tab of a system probe to display this view. Use this view to set the date and time on the system. **Date Time** only appears for systems that you have sufficient privileges to modify the date and time settings.

This view includes the following components:

- **Local System Time (UTC)** —Displays the time on the current system running NI Distributed System Manager.
- **Target System Time (UTC)** —Displays the time on the selected target system displayed in the **Probe** view or **Auto View**.
- **Set to local** —Assigns the value of the **Target System Time (UTC)** to the value of the **Local System Time (UTC)**. Use this button to sync times between the system running NI Distributed System Manager and the system displayed in the **Probe** view or **Auto View**.
- **Refresh** —Refreshes the values that appear in **Local System Time (UTC)** and **Target System Time (UTC)**.

I/O Variable Probe View

Requires: NI Scan Engine

Click an I/O variable in the system tree to display this view in the **Auto View**. You also can right-click an I/O variable and select **Probe** to display this view in a separate window.

This view displays the location of an I/O variable and allows you to view or customize the following components:

- **Location** —Specifies the location of the item you want to probe. To probe a variable, for example, enter the information in the following format: `//SystemName/Process/Variable`.
- **Current Value** —Displays the current value of the selected variable.
- **New Value** —Allows you to enter a new value to assign to the variable.
- **Force** — Forces the variable to assume **New Value** until you unforce the variable, reboot the target, or force the variable to assume another **New Value**.
- **Unforce** —Stops forcing the variable and returns control of the I/O value to the scan engine.
- **Set** —Sets the value in the **New Value** field as the current value.
- **Show Trend** —Displays the *trend view* graph of the selected variable.
- **Data Type** —Displays the data type of the selected variable.
- **Timestamp** —Displays the timestamp of the selected variable.
- **Quality** —Displays the quality of the selected variable. The quality indicates errors or other conditions that apply to the variable.
- **Access Type** —Displays the supported access mode(s) of the variable. This includes Read-only, Write-only, or Read/Write. LabVIEW modules and toolkits you have installed might provide additional access types.

OpenCSRs Probe View

Requires: NI Distributed System Manager (Windows)

Right-click **OpenCSRs** and select **Probe** from the shortcut menu to display this view. This view displays *open certificate signing requests* available on the Web Server. You can create new CSRs, display or delete existing CSRs, and install digitally signed certificates that correspond to an open CSR.

This view includes the following components:

- **Open Certificate Signing Requests** —Displays open CSRs on the Web Server.
- **Create** —Launches the *Create a Certificate Signing Request* dialog box.
- **Delete** —Deletes the selected CSR.
- **Show** —Displays the value of the selected CSR.
- **Install** —Displays the text field to paste the *signed certificate returned by the certificate authority (CA)*.

Refer to the LabVIEW Help for information about enabling SSL on the LabVIEW Web Server.

Process Probe View

Requires: NI Distributed System Manager (Windows)

Right-click a process and select **Probe** from the shortcut menu to display this view. This view shows the status of the selected *process* and allows you to start or stop that process.

This view includes the following components:

- **Location** —Displays the location of the selected process.
- **State** —Displays whether the process is online or offline, or displays if you do not have sufficient access privileges to modify the state.
- **Start Process** —Starts the selected process. You must have sufficient access privileges to start a process.
- **Stop Process** —Stops the selected process. You must have sufficient access privileges to stop a process.

Scan Engine Probe View

Requires: NI Scan Engine

Select a target with the *NI Scan Engine* installed to display this view in the **Auto View**. You also can right-click the target and select **Probe** to display this view in a separate window. Use this view to monitor and manage the scan engine running on the target.

This view includes the following components:

- **Forcing** —Contains information and options related to *I/O variable forcing*.



NOTE

Forcing does not persist after a reboot of the target.

- **Forcing Enabled** —Indicates whether forcing is enabled on the target. Forcing must be enabled on the target for individual I/O variable or I/O alias forces to take effect. This setting serves as a master switch for all I/O variable and I/O alias forcing on the target.
- **Enable Forcing** —Enables forcing on the target, switching the state of the **Forcing Enabled** setting to TRUE. This option is available only when I/O variable forcing is disabled.
- **Disable Forcing** —Disables forcing on the target, switching the state of the **Forcing Enabled** setting to FALSE. This option is available only when I/O variable forcing is enabled.



NOTE

Unless you clear forcing, the previous forced values of all forced I/O variables and I/O aliases return when you re-enable forcing on the target.

- **Clear Forcing** —Unforces all currently forced I/O variables and I/O aliases on the target and returns control of the I/O values to the NI Scan Engine.

**NOTE**

This button does not affect the master **Forcing Enabled** setting on the target.

- **Faults** —Contains information and options related to *scan engine faults*.
 - **Fault Present** —Indicates the presence of *faults* on the target. The LED appears black when no fault is present, orange when only minor faults are present, and red when major or unrecoverable faults are present.
 - **Clear All Faults** —Clears all faults on the target. If the target is in Fault mode, clearing all faults returns the target to the previous mode in effect before entering *Fault Mode*.
 - **Clear Selected Fault** —Clears the fault currently selected in the **Fault List**. If no fault is currently selected, no action occurs.
 - **Fault List** —Lists the faults active on the target.
 - **Error Code** —The error code associated with the fault.
 - **Highest Level** —The highest level of the fault since last cleared.
 - **First Occurrence** —The time of the first occurrence of the fault since last cleared.
 - **Num of Occurrences** —The number of occurrences of the fault since last cleared.
 - **Description** —The fault description.
- **Scan Engine Mode** —Contains information and options related to the *mode* of the NI Scan Engine on the target.

**NOTE**

This section is available only with expansion I/O hardware. This section is not available if only local I/O drivers are installed on the target. Refer to the I/O hardware documentation for information about hardware-specific mode behavior.

- **Configuration** —Indicates whether the scan engine is in Configuration mode.
- **Active** —Indicates whether the scan engine is in Active mode.
- **Fault** —Indicates whether the scan engine is in Fault mode.
- **I/O Drivers** —Lists the I/O drivers installed on the target that interface with the NI Scan Engine.
- **Change to Configuration** —Switches the scan engine to Configuration mode. If the target was previously in Fault mode, switching to Configuration mode automatically changes the level of all active major faults to minor.
- **Change to Active** —Switches the scan engine to Active mode. You cannot switch directly to Active mode from Fault mode. You must clear all major faults to leave Fault mode before you can switch to Active mode.
- **Refresh Local Modules** —Discovers new I/O modules on the target. If a local I/O module is detected in a slot that was not previously deployed from a LabVIEW project, this operation creates new I/O variables for the I/O module. These automatically created I/O variables use the default I/O variable settings. This

operation does not add the new I/O variables to a LabVIEW project, and the new I/O variables do not persist when the target restarts. If you physically change an I/O module deployed from a LabVIEW project, this operation cannot overwrite the deployed configuration until you undeploy the I/O module in the LabVIEW project. However, if you physically change an I/O module that was automatically detected by this operation or the corresponding Refresh Local Modules VI, this operation adds and deletes automatically created I/O variables to match the new physical configuration.

ServerCertificates Probe View

Requires: NI Distributed System Manager (Windows)

Right-click **ServerCertificates** and select **Probe** from the shortcut menu to display this view. This view displays *self-signed certificates* available on the Web Server, including the default LabVIEW self-signed certificate. You can create a new self-signed certificate and display or delete existing self-signed certificates.

This view includes the following components:

- **Server Certificates** —Displays self-signed certificates on the Web Server.
- **Create** —Launches the *Create a Self-Signed Certificate* dialog box.
- **Delete** —Deletes the selected self-signed certificate.
- **Show** —Displays the value of the selected self-signed certificate.

Refer to the LabVIEW Help for information about enabling SSL on the LabVIEW Web Server and the default LabVIEW self-signed certificate.

Variable Probe View

Requires: NI Distributed System Manager (Windows)

Right-click a variable and select **Probe** from the shortcut menu to display this view. This view displays the location of the selected *shared or network variable* and allows you to view or customize aspects of that variable.

This view includes the following components:

- **Location** —Displays the location of the selected variable.
- **Current Value** —Displays the current value of the selected variable.
- **New Value** —Allows you to enter a new value to assign to the variable.
- **Set** —Sets the value in the **New Value** field as the current value.
- **Show Trend** —Displays the *trend view* graph of the selected variable.
- **Data Type** —Displays the data type of the selected variable.

- **Timestamp** —Displays the timestamp of the selected variable.
- **Quality** —Displays the quality of the selected variable. The quality indicates errors or other conditions that apply to the variable.
- **Access Type** —Displays the supported access mode(s) of the variable. This includes Read-only, Write-only, or Read/Write. LabVIEW modules and toolkits you have installed might provide additional access types.

Web Server Probe View

Requires: NI Distributed System Manager (Windows)

Right-click a Web Server and select **Probe** from the shortcut menu to display this view. This view displays Web services available on the Web Server. You can pause, resume, restart, delete, or reset statistics for one or all of the available Web services. Refer to the LabVIEW Help for information about LabVIEW Web services and related terminology.

This view includes the following components:

- **Location** —Displays the location of the selected Web Server.
- **Deployed Web Services** —Displays a listbox of all Web services available on the Web Server.
 - **Operations on selected** —Displays components that perform operations on the selected Web service.
 - **Pause** —Pauses the selected Web service.
 - **Resume** —Resumes the selected Web service.
 - **Restart** —Restarts the selected Web service.
 - **Reset Statistics** —Resets all statistical variables for the selected Web service.
 - **Delete** —Removes the selected Web service from the Web Server.
- **Operations on all** —Displays components that perform operations on all Web services on the Web Server.
 - **Pause All** —Pauses all Web services on the Web Server.
 - **Resume All** —Resumes all Web services on the Web Server.
 - **Restart All** —Restarts all Web services on the Web Server.
 - **Reset All Statistics** —Resets statistical variables for all Web services on the Web Server.

Web Server URL Mapping View

Requires: NI Distributed System Manager (Windows)

Select a URL mapping and click the **Run Selected** button on the [Web Server Probe](#) view to display this view. This view allows you to assign values to connector pane inputs of the

Web method VI specified by the selected URL mapping. You then can send an HTTP request with these values to the Web method VI. Refer to the LabVIEW Help for information about LabVIEW Web services and related terminology.

This view includes the following components:

- **Parameters** —Displays available connector pane inputs for the Web method VI associated with the selected URL mapping. You must assign connector pane inputs before you send an HTTP request to the Web method VI.
- **Values** —Specifies values for the connector pane inputs of the Web method VI associated with the selected URL mapping.
- **Results** —Displays the response from the Web service after you send an HTTP request.
- **Host** —Displays the computer name of the Web Server running the selected Web service.
- **Port** —Displays the port of the Web Server interface running the selected Web service.
- **Request** —Displays the complete URL mapping for the HTTP request.
- **Send request** —Sends an HTTP request to the Web method VI associated with the selected URL mapping. You must specify **Values** for the connector pane inputs of the Web method VI before you send an HTTP request.

Web Service Probe View

Requires: NI Distributed System Manager (Windows)

Right-click a Web service and select **Probe** from the shortcut menu to display this view. This view displays the state of the selected Web service and allows you to pause, resume, restart, or reset statistics for the Web service. You also can send HTTP requests to the Web service using available URL mappings. Refer to the LabVIEW Help for information about LabVIEW Web services and related terminology.

This view includes the following components:

- **Location** —Displays the location of the selected Web Service.
- **State** —Displays whether the Web service is running or paused.
- **Pause** —Pauses the selected Web service.
- **Resume** —Resumes the selected Web service.
- **Restart** —Restarts the selected Web service.
- **Reset Statistics** —Resets all statistical variables for the selected Web service.
- **URL Mappings** —Displays a listbox of available URL Mappings for the selected Web service.
 - **Run Selected** —Displays the [Web Server URL Mapping](#) view for the selected URL mapping.

Configure Web Services Security View

Requires: NI Distributed System Manager (Windows)

Right-click a Web Server in the system tree and select **Web Services Security** from the shortcut menu to display this dialog box. Use this dialog box to configure security for LabVIEW Web services. Refer to the LabVIEW Help for information about LabVIEW Web services security and related terminology.

This view includes the following components:

- **Web Services security key** —Configures the API key used for Web services security.
 - **Access ID** —Specifies the public ID component of the API key. The **Access ID** works similarly to a username.
 - **Secret ID** —Specifies the private ID component of the API key. The **Secret ID** works similarly to a password and only authorized clients should possess this string.
 - **Paste from Clipboard** —Pastes an API key located on the clipboard into the **Access ID** and **Secret ID** fields. If the content of the clipboard does not conform to the correct format of an API key, the content of the clipboard does not import and the current **Access ID** and **Secret ID** remain.
 - **Reset** —Resets the **Access ID** and **Secret ID**.

Configuring Alarms for Shared Variables (DSC Module)

Complete the following steps to configure alarms for shared variables.

1. In the system tree pane, locate the alarm you want to configure and expand this alarm. The System Manager displays the attributes of the alarm.
2. Right-click the attribute you want to configure and select **Probe** from the shortcut menu. A **Probe** view of the attribute appears.
3. Specify a new value of the attribute in the **New Value** text box or option buttons to configure this attribute.



NOTE

You cannot configure the alarm statuses, such as **Ack**, **Description**, and **Set**.

4. Click the **Set** button to set the new value of the attribute.



NOTE

You also can configure alarms by using the **Auto View**.

Monitoring Alarms and Events

Use the [Alarms and Events](#) view to monitor alarms and events. Complete the following steps to monitor alarms and events by using the **Alarms and Events** view.

1. Locate the **Alarms and Events** view. If you do not see this view, select **View»Alarms and Events** to display this view.
2. (Optional) Configure the filter options to display alarms, events, or both alarms and events in this view.
3. Click the **Select Processes** button to display the [Select Processes](#) dialog box. You also can display this dialog box by right-clicking in the **Alarms and Events** view and selecting **Select Processes** from the shortcut menu.
4. In the **Available Processes** list, select the processes you want to monitor. Press and hold the <Shift-Ctrl> keys to select multiple processes from the list.
5. Click the **Add** button to add this process to the **Selected Processes** list.



NOTE

You can select a process from the **Selected Processes** list and click the **Remove** button to remove the process from this list. You also can click the **Clear** button to remove all processes from this list.

6. Repeat steps 4 and 5 to add all the processes that you want to monitor to the **Selected Processes** list.
7. Click the **OK** button to close the **Select Processes** dialog box.

The **Alarms and Events** view displays alarms and events that the selected processes generate. You can [acknowledge](#) these alarms and events by using the **Alarms and Events** view.

Monitoring and Editing Data Item Ranges

Complete the following steps to monitor and edit data items by using the **Probe** view or **Auto View**.



NOTE

You cannot monitor or edit extended data items or array data items in the NI System Manager.

1. In the [system tree](#), right-click a data item range and select **Probe** from the shortcut menu to create a **Probe** view. The **Probe** view displays detailed information about the data item range.
2. In the **Change item** text box, enter the data item that you want to monitor or edit.
3. Click the **Apply** button to display the value of the data item.
4. Enter the new value of the data item in the **New Value** section.
5. Click the **Set** button to apply the new value to the data item.

**NOTE**

You also can click the **Auto View** to display the detailed information.

Monitoring Data Items in Data Item Ranges

Complete the following steps to create a new **Watch List** view for data item ranges or add a data item range to an existing **Watch List** view.

1. In the **system tree**, right-click a data item range and select **Watch List** from the shortcut menu. The System Manager displays the **Add Item to Watch List** dialog box. You also can drag a data item to an existing **Watch List** view to display this dialog box.
2. In the **Items from range** text box, enter the data item to monitor.
3. Click the **OK** button to create a watch list for the data item.

**NOTE**

You can create a **Watch List** view for multiple data item ranges or multiple system tree objects that include data item ranges. You also can add these system tree objects to an existing **Watch List** view. When you create a **Watch List** view for these system tree objects or add these system tree objects to an existing **Watch List** view, the **Add Item to Watch List** dialog box does not appear. The System Manager adds the first data items of the data item ranges by default.

Monitoring Variables with the Watch List View

The **Watch List** view displays information about a single **variable** or multiple variables contained within a system or process. The **Watch List** view displays the variable location, current value, data type, timestamp, access (read/write) status, and quality. The quality indicates errors or other conditions that apply to the variable.

**NOTE**

Variables that use the array and cluster data type do not display the values of individual components.

Complete the following steps to monitor variables using the **Watch List** view.

1. Right-click a **system tree object**, which includes all **systems**, **processes**, and variables available in the system tree.
2. Select **Watch List** from the shortcut menu to open a new **Watch List** view.
3. To customize the **Watch List** view, right-click anywhere inside the **Watch List** view.
4. Select **Show Columns** from the shortcut menu to show and hide any of the information columns available in the **Watch List** view. You also can reorder the columns by dragging the column headings.

Creating Certificates and Processes

Creating a Certificate from a CSR

After you [create a certificate signing request \(CSR\)](#), you must have the CSR digitally signed by a trusted certificate authority (CA) in order to create a certificate.

Complete the following steps to create a certificate from a CSR.

1. Create a CSR.
2. In System Manager, expand the tree item for the system, such as **localhost**.
3. Expand the item **NI_SSLConfiguration**.
4. Expand the item **OpenCSRs**.
5. Select a CSR from the tree, such as Request Handle -100.
6. Copy the text from the **Certificate Signing Request** field of the [OpenCSRs Probe](#) view for the item. If the **OpenCSRs Probe** view does not appear, right-click the CSR and select **Probe** from the shortcut menu.
7. Send the text to a trusted CA that can digitally sign the CSR. If you do not have a trusted CA to sign the request, you also can [create a self-signed certificate](#) rather than using a CSR and a CA.
8. When a CA returns a signed certificate, right-click the CSR from the tree and select **Install** from the shortcut menu.
9. Paste the character string provided by the CA into the text box.
10. Click the **Install** button.

If the installation is successful, a new certificate with the same name as the CSR appears under the **ServerCertificates** item in the tree. The System Manager automatically deletes the CSR from both the tree and the file system.

Creating a Certificate Signing Request

You can create a certificate signing request (CSR) using the System Manager.

Complete the following steps to create a CSR in the System Manager.

1. Enable SSL on an instance of the LabVIEW Web Server. Refer to the LabVIEW Help for information about enabling SSL on the LabVIEW Web Server.
2. Expand the tree item for a system, such as **localhost**.
3. Expand the item **NI_SSLConfiguration**.
4. Right-click **OpenCSRs** from the tree and select **Probe** from the shortcut menu to display the [OpenCSRs Probe](#) view.

5. In the **OpenCSRs Probe** view, click the **Create** button to display the Create a Certificate Signing Request dialog box.
6. In the **Create a Certificate Signing Request** dialog box, enter the appropriate identifying information into the **Values** column.
7. Click the **Submit** button. If you experience errors, confirm that the identifying information uses the correct syntax.

The dialog box displays the name and character string for the new CSR. You must send this character string to a trusted CA to [create a certificate from the CSR](#).



RELATED INFORMATION

[Create a Certificate Signing Request Dialog Box](#)

Creating a Self-Signed Certificate

You can create a custom self-signed certificate using the System Manager. You can use this self-signed certificate in place of the default LabVIEW self-signed certificate.

Complete the following steps to create a custom self-signed certificate in the System Manager.

1. Enable SSL on an instance of the LabVIEW Web Server. Refer to the LabVIEW Help for information about enabling SSL on the LabVIEW Web Server.
2. Expand the tree item for a system, such as **localhost**.
3. Expand the item **NI_SSLConfiguration**.
4. Right-click **ServerCertificates** from the tree and select **Probe** from the shortcut menu to display the [ServerCertificates Probe](#) view.
5. In the **ServerCertificates Probe** view, click the **Create** button to display the [Create a Self-Signed Certificate](#) dialog box.



NOTE

The default LabVIEW self-signed certificate appears as **Certificate Handle-0** in the tree. All custom self-signed certificates that you create are assigned a subsequent number starting at 100, such as **Certificate Handle-100**.

6. In the **Create a Self-Signed Certificate** dialog box, enter the [appropriate identifying information](#) into the **Values** column.
7. Select the duration of the certificate in the **Valid for how many days?** option.
8. Click the **Submit** button. If you experience errors, confirm that the identifying information uses the correct syntax.

The dialog box displays the name and public key for the new certificate. The certificate also appears in the tree under the **ServerCertificates** item.

Creating Processes

You can create processes on a local or LabVIEW Real-Time target.

Complete the following steps to create a process with the System Manager.

1. Right-click a system in the system tree.
2. Select **Add Process** from the shortcut menu to display the Add Process dialog box.
3. Configure the process in the **Add Process** dialog box. You can use this dialog box to configure the process name and whether the process will run after you create it.

After you create a process, you can add shared or network variables or add I/O servers to that process.



RELATED INFORMATION

[Processes](#) on page 1841

[Systems](#) on page 1820

[Creating, Modifying, or Deleting Shared or Network Variables](#) on page 1837

[I/O Servers](#) on page 1817

Creating, Modifying, or Deleting Shared or Network Variables

You can create [shared or network variables](#) within [processes](#) on a local or real-time target.

Complete the following steps to create a variable with the System Manager.

1. Select an existing process in the system tree or [create a new process](#).
2. Right-click the process and select **Add Variable** from the shortcut menu to display the [Shared Variable Properties](#) dialog box.
3. Configure the variable in the **Shared Variable Properties** dialog box. You can use this dialog box to configure settings such as the variable name, data type, and buffering.

You now can use the variable in an application. When the process is running, the [Watch List](#) view updates to include information about the variable such as current value, timestamp, and quality messages. Quality messages list errors and other conditions that apply to the variable.

Modifying a Shared or Network Variable

Right-click the variable in the system tree and select **Edit Variable** from the shortcut menu to display the **Shared Variable Properties** dialog box.

Deleting a Shared or Network Variable

Right-click the variable in the system tree and select **Remove Variable** from the shortcut menu.

Custom VI-based Server - On Input Change Wizard

Requires: DSC Module

Click the **New** button on the Configure Custom VI - On Input Change I/O Server dialog box to display this wizard. You also can select **Tools»DSC Module»VI Based Server Wizard»On Input Change** in LabVIEW to display this wizard.

This wizard includes the following pages:

- **Select VI** —Use this page to specify a VI you want to convert into an I/O server. Contains the following options:
 - **Path to VI** —Specifies the path to the VI that you want to convert to a custom I/O server.
 - **Run VI Analyzer on VI** —Displays the LabVIEW VI Analyzer Toolkit, which runs tests that can identify improvements you can make to the VI. You must have the LabVIEW VI Analyzer Toolkit installed to use this option.
- **Configure Server Distribution Component** —Specifies build options for the I/O server. Contains the following options:
 - **Configuration name** —Specifies the name of the configuration for the custom I/O server. You can have multiple configurations for each custom I/O server. The default name is the name of the VI converted to a custom I/O server.
 - **Support files** —Lists additional VIs you want to include in the LabVIEW project library to deploy. The wizard automatically includes any subVIs or other files that are in the VI hierarchy. You only need to add dynamically called VIs or files.
 - **Select Support File** —Specifies the path to the support files needed to run the VI-based server. Click the **Browse** button to navigate to and select the support files. You must select a file before you can click the **Add** button to add the file to the **Support files** list.
 - **Add** —Adds the VI or file to the **Support files** list.
 - **Remove** —Removes the VI or file you select in the **Support files** list.
- **Server Distribution Component** —Displays a summary of the server the Custom VI-based Server - On Input Change Wizard builds and allows you to build the server. Contains the following options:

- **Server configuration file** —Specifies the path to the VI that the wizard converts to a custom I/O server.
- **Server support files directory** —Specifies the path to the files created when the wizard converts the VI to a custom I/O server.

Click the **Build** button to build the on input change I/O server. When the Custom VI Server - On Input Change Wizard generates the files, you can open the **Configure Custom VI - On Input Change I/O Server** dialog box to configure the I/O server.



RELATED INFORMATION

[Configure Custom VI - On Input Change I/O Server Dialog Box](#)

Custom VI-based Server - Periodic Wizard

Requires: DSC Module

Click the **New** button on the Configure Custom VI - Periodic I/O Server dialog box to launch the Custom VI Server - Periodic Wizard. You also can select **Tools»DSC Module»VI Based Server Wizard»Periodic** to launch this dialog box. Use this wizard to create a custom VI-based periodic I/O server.

This wizard includes the following pages:

- **Select VI** —Use this page to specify a VI you want to convert into an I/O server. Contains the following options:
 - **Path to VI** —Specifies the path to the VI that you want to convert to a custom I/O server.
 - **Run VI Analyzer on VI** —Displays the LabVIEW VI Analyzer Toolkit, which runs tests that can identify improvements you can make to the VI. You must have the LabVIEW VI Analyzer Toolkit installed to use this option.
- **Select Controls and Indicators To Publish** —Specifies the controls and indicators of the VI for which you want to publish values. Contains the following options:
 - **Controls** —Lists the controls on the VI you can select to convert to a custom I/O server.
 - **Select All** —Selects all of the controls or indicators on the VI.
 - **Deselect All** —Deselects all of the controls or indicators on the VI.
 - **Indicators** —Lists the indicators on the VI you can select to convert to a custom I/O server.
 - **Select All** —Selects all of the controls or indicators on the VI.
 - **Deselect All** —Deselects all of the controls or indicators on the VI.
- **Select Method To Stop The Server** —Specifies a mechanism to stop the periodic I/O server. Contains the following options:
 - **Abort VI** —Configures the custom I/O server to abort the VI.

- **Stop the following While Loops** —Configures the custom I/O server to use the While Loop you select to stop running the VI. Contains the following options:
 - **While Loops** —Lists the While Loops available to stop the VI.
 - **View on block diagram** —Opens the block diagram from which you can select a While Loop to stop the VI.
 - **Timeout** —Specifies the amount of time, in milliseconds, you want the VI to wait before timing out.
- **Configure Server Distribution Component** —Specifies build options for the I/O server. Contains the following options:
 - **Configuration name** —Specifies the name of the configuration for the custom I/O server. You can have multiple configurations for each custom I/O server. The default name is the name of the VI converted to a custom I/O server.
 - **Support files** —Lists additional VIs you want to include in the LabVIEW project library to deploy. The wizard automatically includes any subVIs or other files that are in the VI hierarchy. You only need to add dynamically called VIs or files.
 - **Select Support File** —Specifies the path to the support files needed to run the VI-based server. Click the **Browse** button to navigate to and select the support files. You must select a file before you can click the **Add** button to add the file to the **Support files** list.
 - **Add** —Adds the VI or file to the **Support files** list.
 - **Remove** —Removes the VI or file you select in the **Support files** list.
- **Server Distribution Component** —Displays a summary of the files the Custom VI Server - Periodic Assistant builds. Contains the following options:
 - **Server configuration file** —Specifies the path to the VI that the wizard converts to a custom I/O server.
 - **Server template file** —Specifies the path to the template file created when the wizard converts the VI to a custom I/O server.
 - **Server support files directory** —Specifies the path to the files created when the wizard converts the VI to a custom I/O server.

Click the **Build** button to build the periodic I/O server. When LabVIEW generates the files, you can edit the configuration and set initial values for the I/O server in the **Configure Custom VI - Periodic I/O Server** dialog box.



RELATED INFORMATION

[Configure Custom VI - On Input Change I/O Server Dialog Box](#)

Customizing the System Manager Workspace

The System Manager allows you to customize the workspace. Most System Manager views appear as a window by default. You can dock these views on the workspace to build a custom work environment.

Complete the following steps to customize the System Manager workspace.

1. Open a new view, such as a [Watch List](#) or **Event Window** view.
2. Click and hold the top bar of the desired view window.
3. Drag the window to the top, bottom, or sides of the System Manager workspace to dock the window.



NOTE

If you dock multiple views in an area of the workspace, the System Manager automatically adds tabs to allow navigation between the views.

4. To undock a view to a separate window, select the **Restore Down** button  in the top-right of the view.



NOTE

Select **View»Auto View** to display the **Variable Details** view for the selected [system tree object](#). The **Variable Details** view appears docked by default when you select **Auto View**.

5. Select **File»Save Workspace** to save the workspace as a .smw file.
6. Select **File»Open Workspace** and navigate to a .smw file to restore a previously customized workspace.

Processes

Processes are hierarchies of [shared or network variables](#) and [I/O servers](#). Processes provide a logical means of organizing data. For example, if multiple, independent applications run on one server, you can create a process for each application. Similarly, if you have a very large application, you might divide the application into discrete subsystems and create a process for each subsystem. In addition to organizing data, you can start and stop each process independently, which allows you to take one subsystem offline while the rest of the application continues to run. When you stop a process, the values of variables in that process become unavailable and do not update.

Use the System Manager to [create processes](#) and start and stop existing processes.

Shortcut Menu

Right-click a process on the [system tree](#) to display the following options.

- **Watch List** —Creates a new [Watch List](#) view that displays the information for all variables contained within the process you select.
- **Probe** —Displays the [Process Probe](#) view for the selected process.
- **Add Variable** —Displays the [Shared Variable Properties](#) dialog box, which you can use to [create a new variable](#) within the selected process.
- **Add I/O Server** —Displays the [Create New I/O Server](#) dialog box to create an I/O server and add it to the [Shared Variable Engine](#).

- **Stop Process** —Stops the selected process in the system tree. When a process stops, the values of variables in that process become unavailable and do not update. You must have sufficient access privileges to stop a process.
- **Start Process** —Starts the selected process in the system tree. Processes automatically start when you create them. If you select the **Stop Process** item to stop a process, you can use **Start Process** to restart the process. You must have sufficient access privileges to start a process.
- **Remove Process** —Removes the selected process in the system tree. You must have sufficient access privileges to remove a process.

