MATRIXx™

AutoCode™ User's Guide

The MATRIXx products and related items have been purchased from Wind River Systems, Inc. (formerly Integrated Systems, Inc.). These reformatted user materials may contain references to those entities. Any trademark or copyright notices to those entities are no longer valid and any references to those entities as the licensor to the MATRIXx products and related items should now be considered as referring to National Instruments Corporation.

National Instruments did not acquire RealSim hardware (AC-1000, AC-104, PCI Pro) and does not plan to further develop or support RealSim software.

NI is directing users who wish to continue to use RealSim software and hardware to third parties. The list of NI Alliance Members (third parties) that can provide RealSim support and the parts list for RealSim hardware are available in our online KnowledgeBase. You can access the KnowledgeBase at `www.ni.com/support`.

NI plans to make it easy for customers to target NI software and hardware, including LabVIEW real-time and PXI, with MATRIXx in the future. For information regarding NI real-time products, please visit `www.ni.com/realtime` or contact us at `matrixx@ni.com`.

NATIONAL INSTRUMENTS™

May 2003 Edition
Part Number 370767A-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599,
Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949,
Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530,
China 86 21 6555 7838, Czech Republic 420 2 2423 5774, Denmark 45 45 76 26 00,
Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427,
India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970,
Korea 82 02 3451 3400, Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466,
New Zealand 1800 300 800, Norway 47 0 66 90 76 60, Poland 48 0 22 3390 150, Portugal 351 210 311 210,
Russia 7 095 238 7139, Singapore 65 6226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197,
Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227,
Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support Resources and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

AutoCode™, DocumentIt™, LabVIEW™, MATRIXx™, National Instruments™, NI™, ni.com™, RealSim™, SystemBuild™, and Xmath™ are trademarks of National Instruments Corporation.

Product and company names mentioned herein are trademarks or trade names of their respective companies.

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/patents`.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Contents

6 Customizing AutoCode and Generated Code

7 Introduction to Software Constructs with AutoCode

A AutoCode Options

Using This Manual

AutoCode® is a powerful automatic code generator, With AutoCode you will soon be automatically generating robust, high-quality, real-time C or Ada source code from SystemBuild™ block diagrams.

Organization

This guide provides the information you need to get started using AutoCode. This guide is organized as follows.

- Chapter 1, *Introduction*, provides an overview of the rapid prototyping concept, the automatic code generation process, and the nature of the real-time generated code.
- Chapter 2, *Using AutoCode*, explains how to generate real-time code by invoking AutoCode from SystemBuild, the Xmath® Commands window, or the operating system command line. Generated code applications are also discussed.
- Chapter 3, *Managing and Scheduling Applications*, details the management of the application control flow via the real-time scheduler. Topics of discussion include scheduler operation sequence, subsystem properties, subsystem interruption, and examples of scheduler operation.
- Chapter 4, *Code Generation for Discrete Systems*, describes the scheduler architecture as it relates to discrete code generation. Topics include IPAR and LPAR.
- Chapter 5, *Code Generation for Continuous Systems*, describes the scheduler architecture as it relates to continuous code generation. Topics include fixed-step integrators, user-defined integrators, and how to generate code for continuous and hybrid systems.

- Chapter 6, *Customizing AutoCode and Generated Code*, provides advanced methods for customizing AutoCode and its output real-time code using AutoCode configuration options, templates (see also the *Template Programming Language User's Guide*), BlockScript, and %variables.
- Chapter 7, *Introduction to Software Constructs with AutoCode*, describes User Code Blocks, Macro Procedure Blocks, and Procedure SuperBlocks.
- Appendix A, *AutoCode Options*, describes options that can be used when invoking AutoCode from within the Xmath Commands window. This appendix also describes how to use an `autostart.opt` file.

This guide also has an *Index*.

For more advanced details and information necessary to customize both AutoCode and the generated real-time output code, see the *Template Programming Language User's Guide* or the *AutoCode Reference*.

Conventions

This section describes the conventions used in this manual.

Font Conventions

Fonts other than the standard text default font are used as follows:

%	Represents the C shell system prompt in command examples.
`courier`	Denotes directory names, file names, menu names, menu options, commands, Template Programming Language (TPL) segments, syntax statements, operators, and scope classes within text. This font type is also used in programming examples.
italic	*Italic* is used with the default font for emphasis, first instance of terms, and publication titles.
`courier italic`	Denotes placeholders in syntax examples.
`courier bold`	Denotes command-line input.
Bold Helvetica narrow	Buttons, fields, and icons in a graphical user interface are set in **bold Helvetica narrow** type. Keyboard keys are also set in this type.
<*hll*>	Single-letter abbreviation denoting a high-level language supported by AutoCode: c (C) or a (Ada).
<*HLL*>	Abbreviation for the full name of one of the two high-level languages supported by AutoCode: C or Ada.

Format Conventions

This manual uses two special formatting conventions: one to present code and programming examples, as well as sample procedures, and one to present sample input/output.

Sample Procedures

Example 1 shows the formatting convention for sample procedures and code and programming examples.

EXAMPLE 1: Code, Programming, and Sample Input/Output Example Format

```
#include "XmathLNX.h"

extern void cosine_();

void myfun(nlhs, lhs, nrhs, rhs)
int nlhs, nrhs;
externType **lhs, **rhs;
{
   /* y = cos(x) */

   et_matrix *x, *y;
   x = (et_matrix*)rhs[0];
   y = AllocateMatrix(x->rows, x->columns, 1);
   lhs[0] = (externType*)y;
   cosine_(&x->rows, &x->columns, x->real, y->real);
}

static functionData fdata[] = {
   {"myfun", myfun, 1, 1, 1, 1, (char*)0},
   {0,0,0,0,0,0,0}
};

main(argc, argv)
int argc;
char **argv;
{
   XmathMain(argc, argv, fdata, 1);
   return 0;
}
```

Sample Input/Output

The formatting convention for sample input/output is shown in the next several paragraphs.

Sample input is shown in bold and italic (used for placeholders):

`makeproject` ***projname***

Sample output is shown in Example 1 above.

Mouse Conventions

This document assumes you have a standard, right-handed three-button mouse. From left to right, the buttons are referred to as MB1, MB2, and MB3. All instructions assume MB1 unless otherwise noted.

Note and Caution Conventions

Within the text of this manual, you may find notes and cautions. These statements are used for the purposes described below.

NOTE: Notes provide special considerations or details which are important to the procedures or explanations presented.

CAUTION: **Cautions indicate actions that may result in possible loss of work performed and associated data. An example might be a system crash that results in the loss of data for that given session.**

How to Access Integrated Systems-Supplied Files

At several places in this user's guide, you are asked to execute, study, copy, and/or modify certain files we provide for your use. This document uses environment variables (created during the installation) to specify paths. These variables are:

UNIX:	`$ISIHOME`	`$XMATH`	`$SYSBLD`	`$CASE`
Windows:	`%ISIHOME%`	`%XMATH%`	`%SYSBLD%`	`%CASE%`

Xmath will process these environment variables properly, but the same call issued from the Command Prompt may fail if the variable is not set in your working environment. In this case the absolute path must be specified in lieu of the environment variable. For example,

Xmath:	`oscmd("ls $SYSBLD/examples")`
UNIX:	`ls /host/mtx/platform/ver/sysbld/examples`
Xmath:	`oscmd("dir %SYSBLD%\examples")`
Windows:	`dir \\host\mtx\version\sysbld\examples`

If you want to perform file operations from your operating system (rather than via MATRIX$_X$) you must know the values of these variables for your installation. Use `oscmd("env")` or `oscmd("set")` as appropriate for your operating system.

The operating system output will be displayed in the Xmath log area. Locate the absolute path value for the environment variable you want to use and use it in your file specification. Alternatively, define these environment variables in your personal work environment using the absolute paths you have identified.

Environment Variables

The Xmath Commands window is the only place that can recognize environment variables. Xmath commands use *`$env_var`* (for example, `$SYSBLD` or `$XMATH`) whereas commands that go directly to the operating system, such as `oscmd`, use the operating system convention (for example, *`%env_var%`* for Windows).

Related Publications

Integrated Systems provides a complete library of publications to support its products. In addition to this guide, publications that you may find particularly useful when using AutoCode include the following:

- *Xmath Basics*
- *SystemBuild User's Guide*
- *RealSim User's Guide*
- *AutoCode Reference*
- *Template Programming Language User's Guide*
- *DocumentIt User's Guide*

For additional documentation, see the MATRIX$_X$ Help or the Integrated Systems home page at `http://www.isi.com`.

Support

You can contact MATRIX$_X$ Technical Support in any of the three ways listed below. When Technical Support responds, you will be given a Call ID specific to the problem you have reported. *Please record the Call ID* and use it whenever you contact Technical Support regarding the issue.

- Submit a problem report via the ISI web site using the following URL:

  `http://www.isi.com/Support/MATRIXx`

  This is the preferred method, as it is the most traceable; your problem report will be automatically entered into our support database.

- Send e-mail to `mx_support@isi.com`. We can serve you better if you mail us details on your configuration and the circumstances under which your problem occurred. We provide an ASCII file that you can use as a template for your email to support; it can be found in:

  MATRIXX`/version/v6support.txt`

  where

  `MATRIXX` is your MATRIX$_X$ product version directory, which is under `ISIHOME`. To determine this directory location, from the Xmath Commands window, type one of the following commands.

  On the PC:

  ```
  oscmd("echo %MATRIXX%");
  ```

  On UNIX:

  ```
  oscmd("echo $MATRIXX");
  ```

  If you have Xmath running you can use the following Xmath function call to copy the template to the current working directory:

  ```
  copyfile("$MATRIXX/version/v6support.txt")
  ```

- Call 800-958-8885 (where 1-800 service is available) or 408-542-1930. Telephone support hours are 7:00 a.m. through 5:30 p.m. PST, Monday through Friday. We can respond more efficiently if you are ready to provide the information requested in *MATRIXX*`/version/v6support.txt` at the time you call.

Using the ISI FTP Site

If your problem involves scripts or model file(s), Technical Support may ask you to FTP your files to us for further examination.

1. Connect to the ISI FTP site:

   ```
   ftp ftp.isi.com
   ```

2. Log on as `anonymous`, and supply your e-mail address as the password.

3. Change to the `/incoming` directory:

   ```
   cd /incoming
   ```

4. Use `put` or `mput` to specify the file(s) you are transferring. When the transfer is complete, `quit`.

5. Send an email message to Technical Support that states the Call ID (if available), the *exact* name(s) of the file(s) you put in `/incoming`, and the approximate time you made the transfer; alternatively, call 800-958-8885 (where 1-800 service is available) or 408-542-1930 and provide this information. It will be a minimum of 15 to 20 minutes before the transferred file(s) will pass through the firewall.

1 Introduction

This chapter provides an overview of the rapid prototyping concept, the automatic code generation process, and the nature of the real-time generated code.

1.1 Rapid Prototyping Concept

Conventional real-time system development usually takes place in stages, with separate tools for control design, software engineering, data acquisition, and testing. The MATRIX$_X$® product family integrates tools for each stage of system development into a single environment. This allows a design to move easily from one stage to the next, making it possible to create a working prototype early in the design process.

Within the MATRIX$_X$ SystemBuild™ and Xmath® products, you can build, simulate, analyze, test, and debug a model. You can then use AutoCode™ to generate real-time code in a high-level language (C or Ada) for the model. The generated application code can be evaluated on the host with SystemBuild simulation or run on the RealSim™ controller for hardware in-the-loop testing. The generated application code can be cross-compiled and linked for implementation on an embedded processor. You can also use DocumentIt™ to generate documentation. Figure 1-1 shows AutoCode in the MATRIX$_X$ product line.

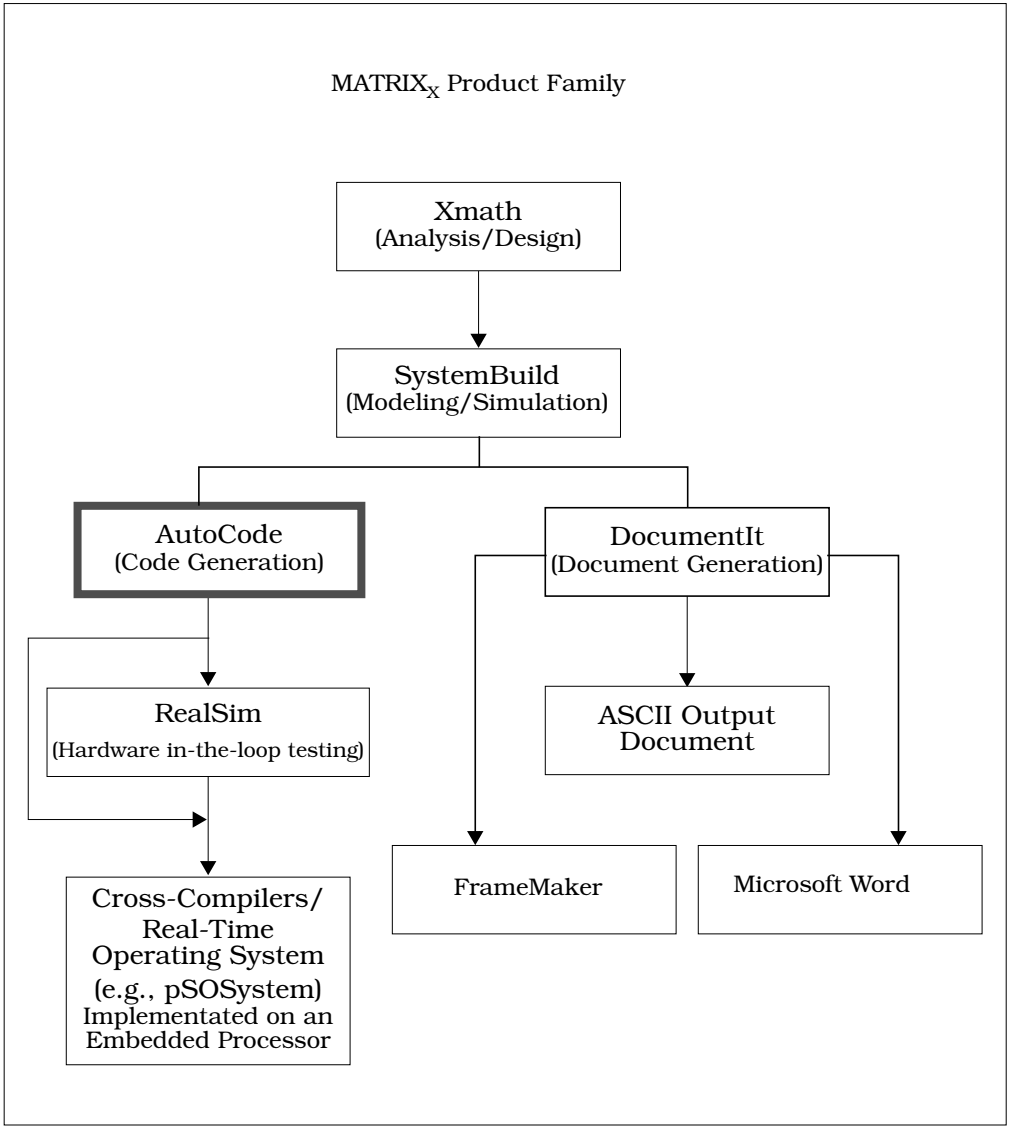


FIGURE 1-1 AutoCode in the MATRIX$_X$ Product Line

1.2 Automatic Code Generation Process

As an integral part of the Integrated System Rapid Prototyping concept, AutoCode lets you generate high-level language code from a SystemBuild block diagram model quickly and automatically. A typical sequence for using AutoCode is as follows (this sequence corresponds to the sequence shown in Figure 1-2 on page 1-4):

1. Build the Model and Validate Through Simulation

   You can quickly develop the continuous-time plant model and corresponding discrete-time controller SuperBlocks using SystemBuild block diagrams. The SystemBuild model is built up from a large palette of blocks that combine to describe the way that the model works and how it should be controlled. You can then analyze and simulate the plant and controller in Xmath; if you find any errors, you can easily amend the model and simulate it again until you are satisfied with its performance. You can perform parametric studies in simulation and pass the values from the Xmath workspace into the generated code.

2. Customize Code Generation

   You can tailor your generated code using the template programming language (TPL), provided in AutoCode. This programming language lets you customize the code for a wide variety of specialized purposes. Run-time parameterization can be programmed into the generated program. Also, configuration information can be entered when the real-time code is generated.

3. Generate the Real-Time Code

   You can invoke AutoCode from inside SystemBuild, from the Xmath Commands window, or from the operating system command line. AutoCode processes discrete-time and continuous-time SuperBlocks to generate high-level language code in C or Ada.

4. Compile and Link

   You can customize the environment in which the generated code runs by editing standalone Input/Output utilities files included with AutoCode. Additionally, you can further enhance the functionality of your model by adding UserCode Blocks. Compile and link the generated code with these standalone files and any UserCode Blocks that you will be using to produce a standalone real-time application program for simulation on the host. Refer to Section 2.4 on page 2-9.

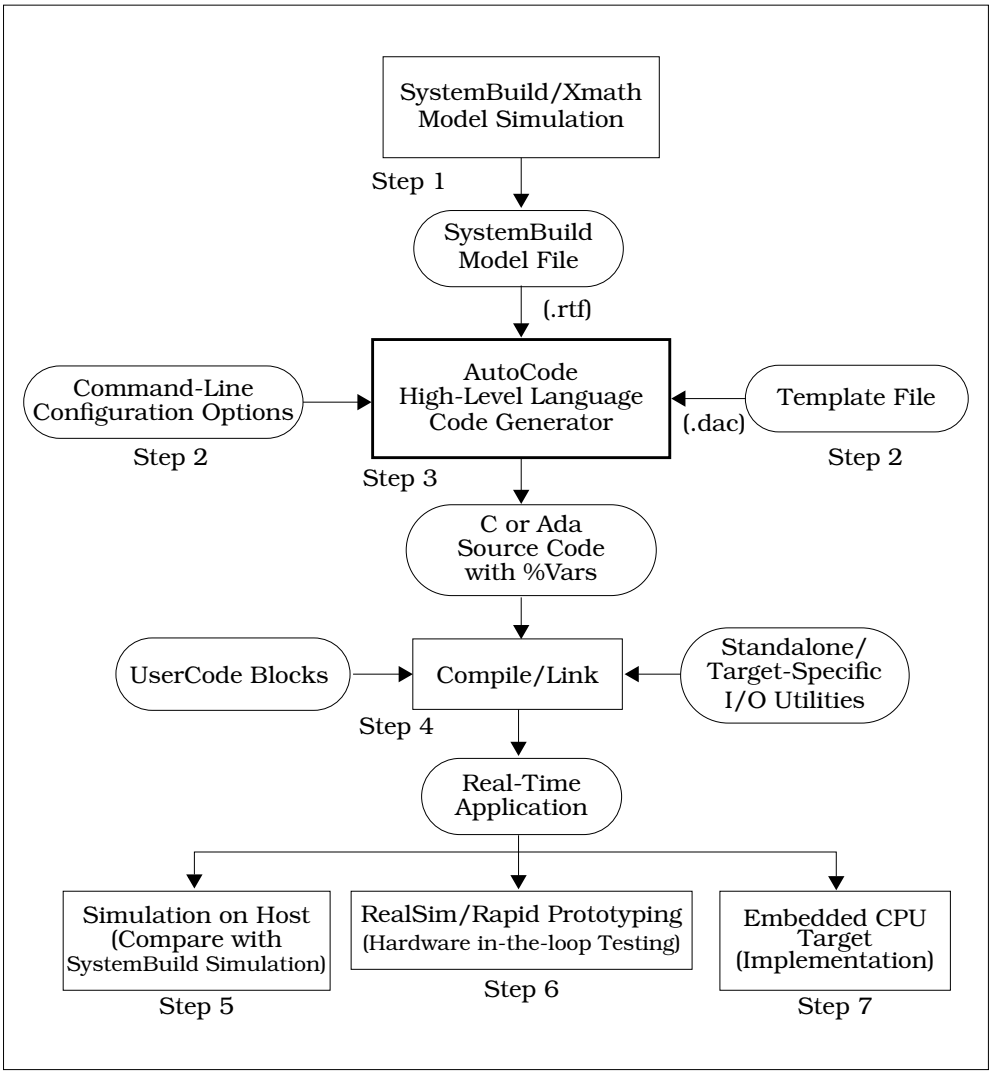


FIGURE 1-2 AutoCode Automatic Code Generation Process

5. Validate the Generated Code Through Simulation Comparison

   Now you can test and simulate the generated code on the host and feed the results back to Xmath for comparison with the SystemBuild simulation data. These steps are described in Section 2.4 on page 2-9.

6. Test with Real or Prototyped Hardware

   You can use a rapid prototyping tool such as RealSim to implement a real-time controller, or to perform real-time hardware-in-the-loop testing with actual or emulated hardware. RealSim customers are provided with special templates and target-specific utilities to generate and link code specific to RealSim.

7. Implement the Finished Code on the Target

   After you have completed all needed testing and simulation to optimize the functionality and performance of your application, the perfected code can be implemented on the target processor.

1.3 Profile of the Generated Program

If no user-originated changes are made to the template program, the generated application program consists of calls to a time-critical application manager/scheduler, a re-entrant dispatcher, one or more pre-emptible subsystems, input/output functions, a timer interrupt handler, and a background function (see Figure 1-3 on page 1-6). The calls to the modules of the generated program are in the template program, which lets you modify the way the generated program is structured and expand it as needed. Under control of the default template file, the application program is assembled from components taken from a variety of files, with different provisions for user modification, as explained in the following:

- Manager/Scheduler

  The manager/scheduler is a time-critical routine that performs external input/output functions for the application program, takes care of various housekeeping tasks, and generates a dispatch list of the subsystems that are ready to be executed.

  Tailoring: This routine can be customized by modifying or rewriting the supplied real-time scheduler. Under control of the template file, any number or variety of scheduler programs can be used.

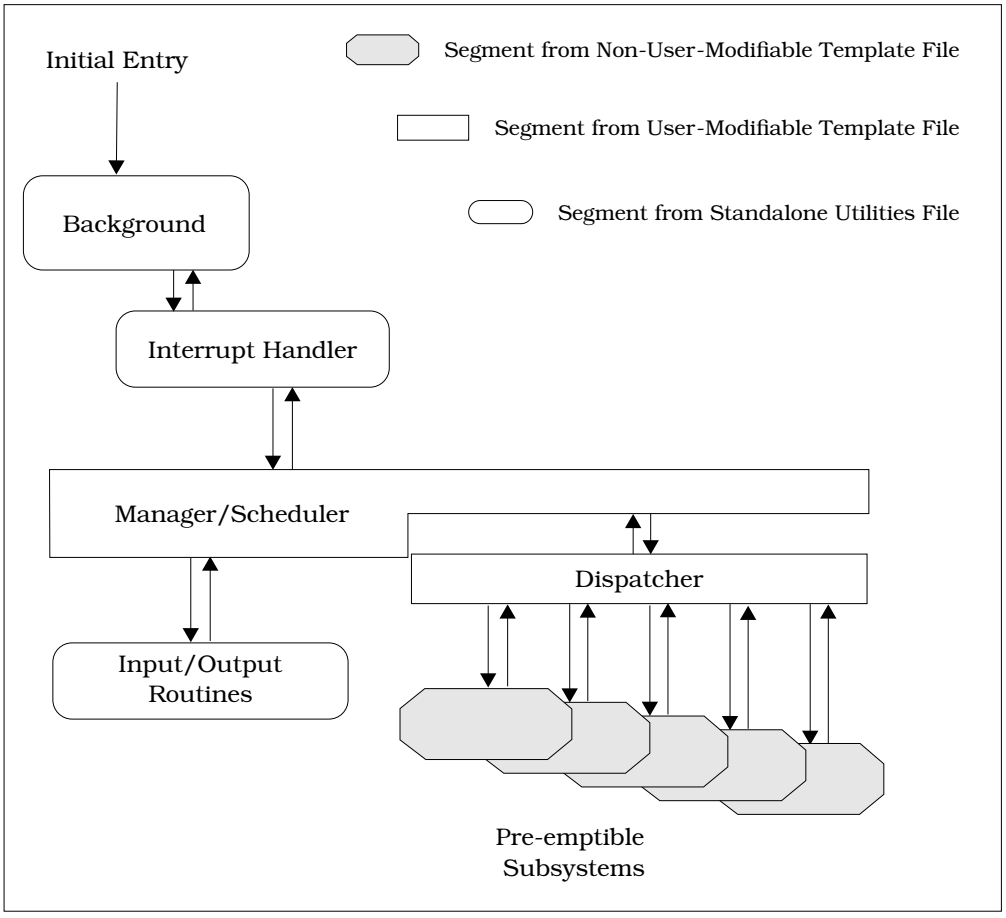


FIGURE 1-3 Components of the Generated Application Program

- Dispatcher

  The dispatcher dispatches the subsystems that are ready to be executed from the dispatch list in a prioritized order. Highest priority subsystems get dispatched first.

  Tailoring: Dispatcher logic can be customized by modifying the template file.

- Subsystems

  The subsystems contain the code implementing block algorithms generated from the SystemBuild model. They implement real-time activities by accepting inputs and posting outputs at times derived from the sampling rates of the SuperBlocks in the SystemBuild model, under control of the manager/scheduler.

  Tailoring: Customization of block code is supported only via the BlockScript and UserCode blocks. The block code for other block types is proprietary. Refer to Section 6.4 on page 6-2 for details regarding BlockScript.

- I/O Routines

  The main function of I/O routines is to provide input data to the AutoCode real-time application on every scheduler cycle and to obtain the computed outputs from it. The supplied I/O routines read inputs from a $MATRIX_X$ formatted ASCII file and write outputs to a file in the same format.

  Tailoring: The input/output routines are part of the standalone utilities file and are intended to be user modified. Any variety of I/O routines can be user-written and invoked as needed under control of the template program.

- Timer Interrupt Handler

  The timer interrupt handler calls (or invokes) the manager/scheduler at a specified time interval. This is not needed for standalone simulation on a host.

  Tailoring: This routine is intended to be user-defined, such that it invokes the scheduler on every minor cycle.

- Background Function

  The background function performs idle time non-time-critical tasks such as self-diagnosis or updating a display; its essential qualification is to be interruptible. For standalone simulation, it merely calls the scheduler for user-specified simulation cycles as specified by time vector.

  Tailoring: This is part of the standalone utilities file that can be user-modified or completely rewritten.

For a detailed explanation of the flow of control in the generated program, see Chapter 3, *Managing and Scheduling Applications*.

1.4 AutoCode-Generated Reusable Procedures

In the earlier sections of this chapter, we looked at the process of generating a real-time scheduled application program and its nature. However, in some cases, you might not want to generate scheduler and related data structures. Rather, you might want to generate merely algorithmic procedures (subroutines). AutoCode provides an option to let you generate only algorithmic procedures from Procedure SuperBlocks, without the scheduler and its data structures.

Following are some uses of generating reusable algorithmic procedures:

- Linking them to your own real-time scheduler (executive) or simulator
- Linking them to the SystemBuild simulator for algorithmic verification and speeding simulation by using integer math in the procedure

The AutoCode generated procedures link to the SystemBuild simulator via a User-Code Block in the SystemBuild model. The procedure describing how to link them is provided in Chapters 2 and 3 of the *AutoCode Reference*.

1.5 Using MATRIX$_X$ Help

MATRIX$_X$ Version 6.X provides a hypertext markup language (HTML) help system. The MATRIX$_X$ Help system is a self-contained system with multiple hypertext links from one component to another. This help system, augmented by online manuals, covers most MATRIX$_X$ topics except for installation. For installation information, see your online or printed manuals.

The MATRIX$_X$ Help system requires Netscape Communicator 4.03 (included on the MATRIX$_X$ CD) or later. On UNIX systems, an OEM version of Navigator is automatically included in the MATRIX$_X$ installation. On PC systems, Netscape Communicator must be installed independently using the Netscape installation procedure included on the MATRIX$_X$ CD.

Additional Netscape Information

For more information on Netscape products, see Netscape's home page at `http://home.netscape.com`.

2 Using AutoCode

This chapter explains how to generate real-time code by invoking AutoCode from SystemBuild, the Xmath Commands window, or the operating system command line. This chapter also discusses generated code applications.

2.1 How to Generate Real-Time Code

Using AutoCode, you can generate C or Ada high-level language code from:

- SystemBuild, which lets you automatically generate a real-time file (`.rtf`) and then source code from a model, using a Graphical User Interface. For ease of use, this is the recommended method of code generation.
- Xmath, which lets you automatically generate an `.rtf` file and then source code from a model, using an Xmath command.
- The operating system command line, which lets you generate source code from an already-existing `.rtf` file, using the `autostar` command from the operating system prompt.

For Xmath Commands window or operating system command-line options, see Appendix A, *AutoCode Options*.

2.1.1 Generating Code from Within SystemBuild

To use AutoCode while inside SystemBuild, select a SuperBlock in the Catalog Browser, and then select Tools→AutoCode to open the dialog. Instructions for using this dialog are in the MATRIX$_X$ Help.

Depending on the template file and command-line options used, the code generated can be either C code or Ada code.

2.1.2 Generating Code from Xmath

The `autocode` command lets you process a model to generate C or Ada code. Two possible syntaxes are supported:

```
autocode, {model = name1, file = name2, ...
language = name3, tpldac = name4, ...
rtf = name5, vars, typecheck}

autocode model, {options}
```

where `name1` identifies the model to be processed for code generation. The model will be either:

- A string (must appear in "quotes"), which must be the name of a SuperBlock that exists in the current SystemBuild Catalog. This SuperBlock is analyzed and processed to generate code.
- A variable (not in quotes). Variables should be assigned to a string, the string must be the name of a SuperBlock in the current catalog; it is analyzed and processed to generate code.

Whenever a file name or other string is included in a command string, it must be enclosed in quotes, but a variable name must *not* be in quotes.

Keywords for the second type of syntax are the same as for the first syntax, except `model`. See Appendix A for the Xmath command options.

Examples:

```
autocode "topSB"
```

The system generates a real-time file named `topSB.rtf`. It loads this file and processes it to produce C code. The output file name is `topSB.c`.

```
autocode "topSB", {tpldac="mytemplet",!vars}

autocode, {model = "topSB", tpldac = "mytemplet", !vars}
```

Either syntax processes the SuperBlock `topSB` in the current catalog to produce C code, using the direct access template file `mytemplet` and no Xmath %variables. The output file name is `topSB.c`.

2.1.3 Generating Code from the Operating System

If a model file already exists, it is also possible to execute AutoCode from the operating system prompt. The file intended for processing must be a real-time file (`.rtf`). At the operating system prompt, execute the command:

```
% autostar {options} model_file.rtf
```

Many of the options are the same as the fields in the Generate Real-Time Code dialog. See Appendix A for the operating system command-line options.

AutoCode runs, creating a high-level language file. When the operating system prompt returns, the process is complete.

Examples:

```
% autostar -h
```

shows a help display.

```
% autostar -l c SysBld_file.rtf
```

processes the model file `SysBld_file.rtf` to produce a C code file named `SysBld_file.c`. All default settings are accepted. It assumes a direct access template file named `c_sim.dac` exists in your working directory.

```
% autostar -l a -t ada_rt.tpl -sd 6 -o CodeFile.a MyModel.rtf
```

processes the ASCII template file `ada_rt.tpl` and produces a direct access template file `ada_rt.dac`. The model file `MyModel.rtf` is then processed, producing ada code in file `CodeFile.a`, which contains numeric literals encoded using a maximum 6 significant digits of precision.

```
% autostar -l c -t c_sim.tpl
```

compiles the template file `c_sim.tpl` to produce a dac file named `c_sim.dac`.

2.1.4 Limitations/Restrictions

- SystemBuild models processed by AutoCode cannot contain algebraic loops.
- AutoCode models cannot accept data that includes complex numbers.
- The input time vector must start at 0.0.
- AutoCode does not support the following blocks:
  - The Zero Crossing Block

  - The MathScript Block
  - The HDL CoSim Block
  - The Implicit UserCode Block

- Macro and Inline Procedure SuperBlocks are not supported within Conditional SuperBlocks.
- When a user input Interactive Animation icon is encountered, code is generated, which assigns a constant value equal to the initial value of the icon.
- Only block comments are inserted in the generated-for-display Interactive Animation icons.
- AutoCode does not generate parameterized code for the following blocks even though parameters within these blocks can be parameterized using the %variable notation. This is because dynamic systems are transformed to optimize performance and the mapping between the Xmath value and the block variable is lost. Variable space will be allocated and initialized but the values are hard-coded.
  - State Space Block
  - Num, Den Block
  - Gain, Zeros, Poles Block
  - Gain, Damps, Freqs Block

The AutoCode software lets you generate ANSI C or Ada code automatically from SystemBuild models.

You can generate code from the Catalog Browser in SystemBuild or use the `autocode` Xmath command. The generated code represents a complete implementation of the model. The generated code can be targeted for and run on other computers or an actual controller. The default target is a standalone simulation that you can execute on your computer; you can then load the results of the simulation back into Xmath for analysis.

2.2 Generating Non-Customized Code

With Xmath running on your PC, generate code for the sample Discrete Cruise System model by taking these steps:

1. Make sure you are in a directory where you want to save your code. If not, enter the following command from the Xmath Commands window:

   `set directory ="`*`your_working_directory`*`"`

2. From the Xmath Commands window, type the following command to load the model:

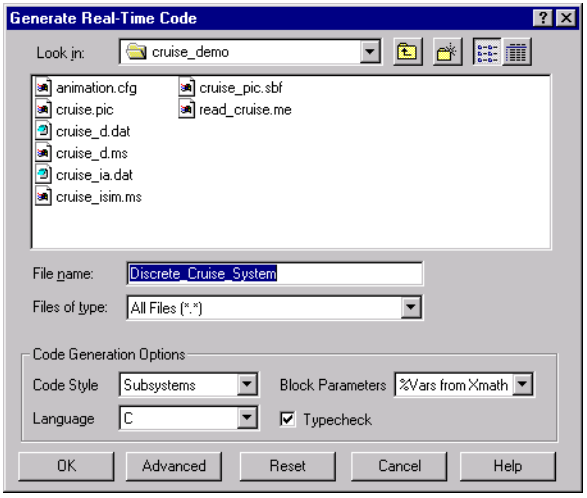
   `load "$SYSBLD\demo\cruise_demo\cruise_d.cat";`

   NOTE: The Xmath Commands window is the only place that can recognize environment variables. For loading with other methods, you must know that full pathname of the SystemBuild directory. Note also that Xmath commands use *`$env_var`* whereas commands that go directly to the operating system, such as `oscmd`, use the operating system convention (for example, *`%env_var%`* for Windows).

3. From the SystemBuild Catalog Browser, select the Discrete Cruise System SuperBlock.

   NOTE: You must generate code from a top level SuperBlock.

4. From the Catalog Browser, select Tools→AutoCode to bring up the Generate Real-Time Code dialog.



5. Enter a name in the **File name** field or accept the default, Discrete_`Cruise_System`.

6. Click **OK** to start the code generation process.

7. Raise the Xmath Commands window to monitor the progress of the code generation.

8. Once the code generation is complete, look for a statement similar to the following in the Xmath log area:

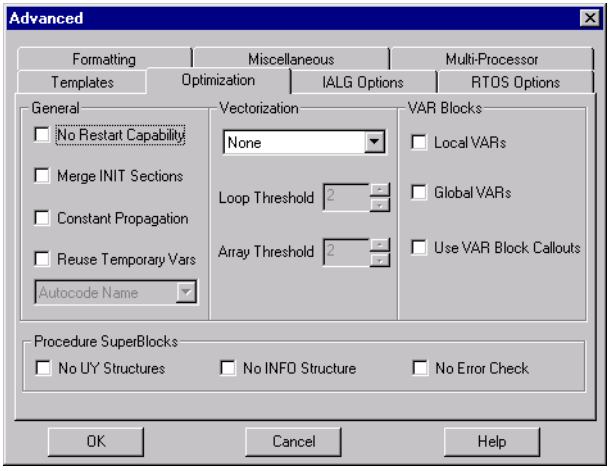
```
Output generated in d:\user\test\Discrete_Cruise_System.c.

Code generation complete.
```

9. (Optional) Display the output file in the Xmath output area by entering a `type` command similar to the following in the Xmath Commands window:

```
oscmd ("type d:\user\test\Discrete_Cruise_System.c")
```

2.3 Generating Customized Code

To customize your AutoCode output, click **Advanced** on the Generate Real-Time Code dialog; this brings up the Advanced dialog.



You can use the Advanced dialog from the AutoCode Code Generation dialog or use keywords with the `autocode` Xmath command to customize the generated code as follows:

- The Templates tab lets you control the formatting of the output of AutoCode to meet a variety of software needs; you can modify the overall architecture of generated code, customize the scheduler, modify data structures and external I/O calls, add user code, and so forth. Using the Template Programming Language (TPL), you can tailor any part of the code except the hierarchy logic and the elementary blocks. Numerous templates are available, including one to customize the generated code for the pSOSystem real-time operating system. For more information on templates, see the *Template Programming Language User's Guide*.
- The Formatting tab lets you set the form of the generated code. You can specify settings for the number of significant digits, maximum variable name length, maximum number of characters per line, and other formatting settings.
- The IALG Options (Integration Algorithms) tab lets you select an algorithm such as Euler, Runge Kutta, or Kutta-Merson.
- The Multi-Processor tab lets you specify a processor, startup, background, interrupt, skew, priority, or map file.

- The Optimization tab (shown in Section 2.3 on page 2-7) lets you make general, vectorization, and VAR block settings that affect code size and efficiency (see the *Autocode Reference* for details).
- The Miscellaneous tab (shown in Figure 2-1) lets you select an options file, the type of scheduler, output scope control, and various other settings. For information about how to set `epsilon`, for example, see Appendix A, *AutoCode Options*.
- The RTOS (real-time operating system) tab lets you specify a configuration file and set additional options.

Once you have customized your settings, you click **OK** in the Advanced dialog; then you generate code by clicking **OK** in the Generate Real-Time Code dialog.

For information about `autocode` keywords, see Appendix A, *AutoCode Options* and the MATRIX$_X$ help.

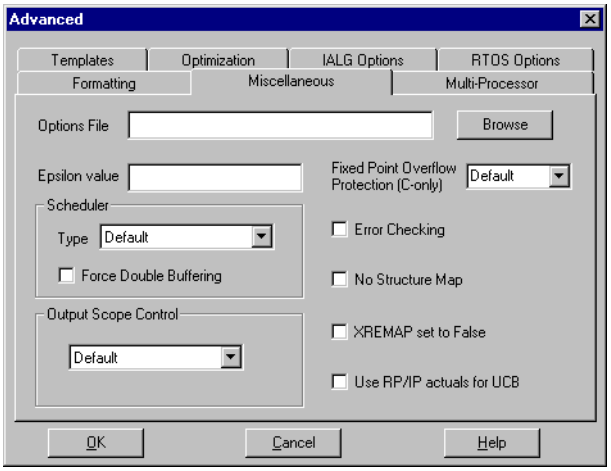


FIGURE 2-1 Advanced Dialog, Miscellaneous Tab

2.4 Applications of AutoCode-Generated Code

In Section 2.1 you learned how to generate real-time code using AutoCode from SystemBuild, the Xmath Commands window, and the operating system command line. These are applications of the AutoCode-generated code:

- Standalone simulation on the host machine
- Rapid Prototyping
- Real-time simulation
- Embedded real-time control

2.4.1 Standalone Simulation

AutoCode lets you execute code as a standalone on the host or PC machine; use the same input vectors for SystemBuild simulation; test the generated application with these vectors; and load the output vectors back to Xmath for comparison and analysis.

NOTE: Use the `csi` option so generated code will match sim results for continuous systems.†

When a system has been modeled in SystemBuild and its source code has been generated using AutoCode, you can compare the outputs from the generated code against those obtained from simulating the block diagram. The Standalone Library is provided in your `src` distribution directory to support this function.

The Standalone Library is a collection of subroutines that performs the operations of the target-specific utilities to allow testing of the generated code in a traditional non-real-time host or PC environment. The Standalone Library provides such services as reading in an input data file and producing an output file.

The role of the Standalone Library in testing the generated application code against the simulations is illustrated in Figure 2-2 on page 2-10.

---

† For standalone AutoCode, results for generated code will not match sim unless the `csi` option is not zero. Typically, set `csi` to 0.01, the time vector for standalone sim. Then, results will match.

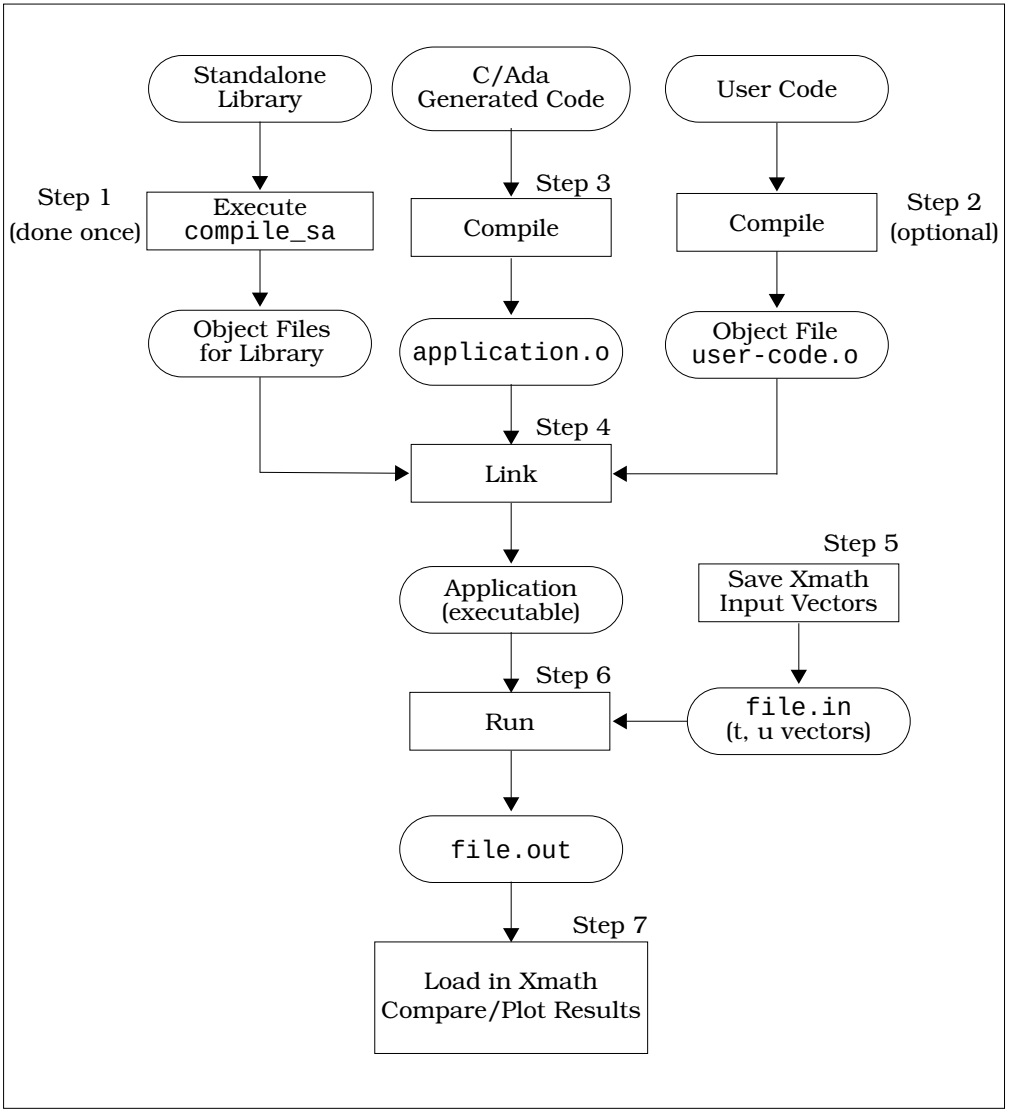


FIGURE 2-2 Compiling, Linking, and Running the Generated Program

Standalone Simulation for UNIX

The steps required to test the generated code in a host environment are:

1. Make sure that the Standalone Library files are in your local working directory. Copy all the `$CASE/ACC/src/sa_*.*` files to your local working directory. This must be the original version, not modified for your target installation. Compile the Standalone Library files. The `compile_c_sa.sh` is provided to compile the Standalone Library files. This needs to be done only once in a given system, and need not be repeated for each model. This step produces object files for the Standalone Library.

   NOTE: Please refer to any comments within the compile script for additional options and variations of the Standalone Library.

   For Ada, the `compile_ada_sa.sh` creates a library named `salib` and compiles the Standalone Library into it.

2. If you have any user code that you want to call from an application, then compile it into an object using your compiler. This is a C example for compiling `user_code.c` on a Sun (UNIX) workstation:

   ```
   % cc -c -DSUN user_code.c
   ```

   This creates an object file called `user_code.o`. Here, the option `-DSUN` defines symbol "SUN" which is required to indicate the operating system.

   NOTE: Refer to the Language-Specific Reference for C in the *AutoCode Reference* for the list of platform defines to be used for specific operating systems.

   Similarly, an Ada example for compiling `user_code.a` on a Sun (UNIX) workstation using the Verdix Ada compiler is as follows:

   ```
   % a.mklib -f mylib salib

   % ada user_code.a -L mylib
   ```

   where `mylib` is the library name of your choice and `salib` is the library created by `compile_ada_sa.sh`.

3. Compile and link the generated source code with the Standalone Library. A C example for compiling and linking code generated in `model.c` on a Sun workstation is:

   ```
   % cc -DSUN -o executable model.c user_code.o sa_*.o -lm
   ```

NOTE: The `user_code.o` is an optional file (see step 2 above).

An Ada example for compiling and linking code generated in `model.a` on a Sun workstation is:

```
% ada model.a -L mylib

% a.ld model -L mylib
```

where `model` is the name of the main program.

4. Create an input file, `file.in`, containing (column) vectors `t` (time) and `u` (inputs), in $MATRIX_X$ ASCII format, using Xmath as follows:

```
t = [0: ...]';
u = [ ... ];
save t u file = "file.in" {matrixx, ascii};
```

Time vector `t` should always start with the first element value as zero. `t` and `u` vectors should have the same number of rows.

5. Run your executable from the OS command line and specify the $MATRIX_X$ ASCII filename as `file.in` and the output file name as `file.out`, as shown below.

```
% executable
Enter xmath {matrixx,ascii} formatted input filename: file.in
Enter output filename: file.out
```

6. Load the output from the generated code `file.out` back into Xmath and compare it to the simulation. (See Section 2.4.2 on page 2-14 for the simulation options. Use the `[te,ye]` extended-time simulation feature to obtain all the discrete time points.) The file contains two items, the output matrix `yrt` and the time vector `ytime`. The generated code and simulation results should match very closely. For more details on each subroutine in the Standalone Library, refer to the comments in the source file.

   After you have completed all needed testing and simulation to optimize the functionality and performance of your application, the perfected code can be implemented onto the target processor.

Standalone Simulation for Windows

The steps required to test the generated code from a Command Prompt in a Windows environment are:

1. Make sure that the environment variables are setup according to the *Microsoft Visual C++ Getting Started* manual. In Windows 98 or Windows 95, this is done via the `autoexec.bat` file (using Set envvar = ...). In Windows NT, this is done

from the Environment tab of System Properties in Start→Settings→Control Panel.

The `path` should have: C:\MSDEV\BIN.

The `include` should have: C:\MSDEV\INCLUDE;C:\MSDEV\MFC\INCLUDE.

The `lib` should have: C:\MSDEV\LIB;C:\MSDEV\MFC\LIB.

2. Copy all the `c:\isi\case\acc\src\SA*.h` files to your local (working) directory. Also, you should copy the file `compile_c_sa.bat` from the same directory to your working directory.

3. Execute the batchfile (`compile_c_sa.bat`, now in your local directory) to compile the utility files to `sa_*.obj` files.

   NOTE: Please refer to any comments within the compile script for additional options and variations of the Standalone Library.

4. If you have any user code that you want to call from an application, then compile it into an object using your compiler. This is a C example for compiling `user_code.c`:

   ```
   CL -Op -O1 -W1 -c -DMSWIN32 -I. -Fouser_code.obj user_code.c
   ```

   NOTE: This is case sensitive. Also, note that there is no space between the `-Fo` and the `user_code.obj`.

   This creates an object file called `user_code.obj`.

5. Compile and link the source code. From a Command Window, use the following (intuitive) command:

   ```
   CL -Op -O1 -W1 -DMSWIN32 -I. -Femodelname.exe modelname.c
      user_code.obj sa_*.obj
   ```

   NOTE: This is case sensitive. Also, note that there is no space between the `-Fe` and the `modelname.exe`. The `user_code.obj` is an optional file (see step 4 above).

6. Create an input file, `something.in`, containing (column) vectors `t` (time) and `u` (inputs), in $MATRIX_X$ ASCII format, using Xmath as follows (for a unit step single input system):

   ```
   t=[0:0.1:1]';u=ones(t);save "something.in" t u {ascii, matrixx}
   ```

   Time vector `t` should always start with the first element value as zero. `t` and `u` vectors should have the same number of rows.

7. To run the model type (at the Command Prompt):

   modelname

   You will be prompted for the input file name (`something.in`) containing the time vector and the inputs that were previously saved from Xmath and the output file name (`something.out`).

   To load the results back into Xmath, load `something.out`, and to plot it for comparison to SystemBuild simulations, select the vector to be incremented by 2 so that you plot every other point. For the simulation options, see Section 2.4.2 . Use the `[te,ye]` extended-time simulation feature to obtain all of the discrete time points.

   The file contains two items: the output matrix `yrt` and the time vector `ytime`. The generated code and simulation results should match very closely. For more details on each subroutine in the Standalone Library, refer to the comments in the source file.

   After you have completed all needed testing and simulation to optimize the functionality and performance of your application, the perfected code can be implemented onto the target processor.

2.4.2 Simulation Options

`Simulation Note:` For best results when using the Standalone Library to compare the generated code with simulation results, you first need to set up the simulator to imitate the generated code's real-time behavior. This is done through simulator options.

The appropriate Xmath command is:

```
setsbdefault,{actiming,extend,typecheck}
```

The SystemBuild simulator provides the `actiming` keyword in order to match AutoCode results for discrete systems. The simulator accomplishes this by matching AutoCode's scheduler cycle, system initialization, and execution and posting times for each subsystem.

Three simulation keyword values are forced so that the initialization and posting of outputs match AutoCode.

`cdelay = 1` The output posting is always delayed one minor cycle.

initmode = 0 This keyword setting disables the initialization that is normally performed at simulation time. 0 = Outputs of continuous subsystems only are computed based on initial conditions and inputs. Outputs of discrete subsystems are set to $\sqrt{\varepsilon}$.

dtout = 0 No extra output time points are specified. This keyword forces the outputs of the simulation to be posted only at the minor cycle of the simulation scheduler, which is defined by the least common multiple of the sampling intervals and timing requirements of the subsystems.

This command can be included in your set-up file. It is a good idea to use these options whenever your discrete controller model is intended for eventual implementation on a digital computer. Analyze your top-level SuperBlock from the SystemBuild menu. For more information, see the MATRIX$_X$ Help or the *SystemBuild User's Guide*.

Assuming that you have executed the `setsbdefaults` statement shown above, you can execute the simulation with the command:

```
[te,ye] = sim(model,t,u,{sim keywords});
```

where:

`model` is a text string enclosed in double quotes, which is the name of the top-level SuperBlock in the SystemBuild Editor.

`t` is the required time vector.

`u` is an input data matrix.

2.4.3 Rapid Prototyping

AutoCode provides the means for fast implementation of SystemBuild block diagrams without lengthy manual coding. You can speed up design iterations by simply editing the block diagrams and generating new code. For more information on rapid prototyping, see the *RealSim User's Guide*.

2.4.4 Real-Time Simulation

AutoCode lets you create and execute code and perform hardware-in-the-loop simulations for an entire system using RealSim. For details regarding real-time simulation, see the *RealSim User's Guide*.

2.4.5 Implement Embedded Real-Time Control

AutoCode lets you generate code for real-time controllers. You can cross-compile, link, and download onto a wide variety of target processors.

3 Managing and Scheduling Applications

This chapter details the management of the application control flow via the real-time scheduler. Topics include scheduler operation sequence, subsystem properties, subsystem interruption, and examples of scheduler operation.

3.1 Real-Time Application Scheduler

AutoCode builds the scheduler as part of the real-time application program by means of the template file. The scheduler performs overall direction and control of inserting inputs, scheduling tasks, posting outputs, and dispatching the tasks that perform the work of the real-time system. Although you can tailor the scheduler as well as other parts of the code, the intention of this program is to provide a generic real-time scheduler, combining high performance with deterministic, prioritized, pre-emptive scheduling of application tasks that have different timing requirements.

The application scheduler operates on the principle of rate-monotonic scheduling, deriving priorities for the tasks from the repetition rate for periodic subsystems and the timing requirement for triggered subsystems. The algorithm assigns higher priority to the faster sample rate or timing requirement subsystems and lower priority to slower ones (see Figure 3-1 on page 3-2). The rate-monotonic algorithm maximizes the number of tasks that get to complete their operations in a given time. Using the rate-monotonic algorithm, all periodic tasks complete their operations if CPU task utilization does not exceed about 70%.

For consistent and deterministic operation in a real-time environment, the task subroutines are scheduled and dispatched as encapsulated objects, which accept inputs and post outputs strictly under control of the scheduler. For this reason, all external input and output operations are handled by the scheduler directly and inter-task data transfer is performed via input sample and hold. The scheduler is re-

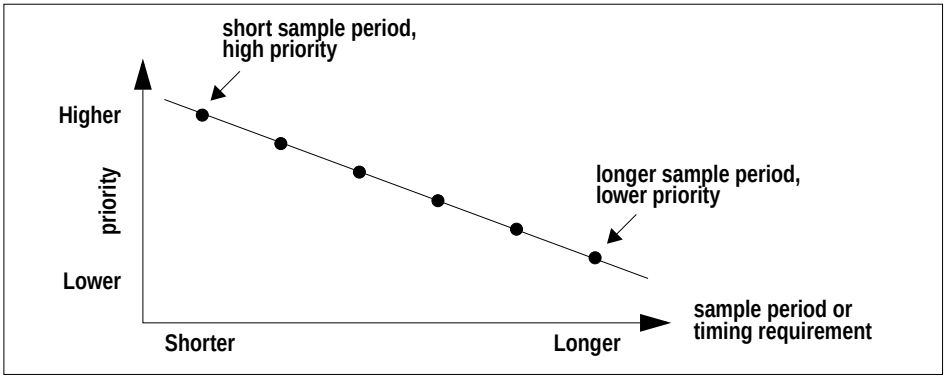


FIGURE 3-1 Rate-Monotonic Scheduling Algorithm

entrant except for the critical section, which must not be interrupted. The scheduler can be called externally by means of an interrupt handler (for real-time applications) or by a background task (for simulation).

3.1.1 Subsystems

The term *subsystem* refers to the entities that are scheduled and dispatched for execution by the generated scheduler. The terms subsystem and task can be used interchangeably. By definition, a subsystem is an independently-scheduled program object, consisting of a single computational thread, which accepts inputs and posts outputs under the control of the scheduler at scheduler-specified times and which can be pre-empted.

Subsystems are constructed by AutoCode from all SuperBlocks in a system that have the same computational timing requirements or attributes (sample rates, skew, timing requirements, enable signals, and triggers). AutoCode has four types of subsystem:

Continuous subsystem	Dispatched every time the dispatcher is invoked.
Free-running periodic subsystem	Executed repetitively at a fixed frequency.
Enabled periodic subsystem	Executed repetitively, but only while its enabling signal remains active.
Triggered subsystem	Executed as and when its trigger is detected.

Throughout this discussion, the behavior of the scheduler and of the subsystems is explained in terms of interrupts or scheduler interruptions. These interruptions are implementation-dependent, involving a hardware timer interrupt, a wakeup call, or some other method of invoking the scheduler. The operation of the generated code is the same, regardless of which method of invoking the scheduler is used.

3.1.2 Flow of Control in the Generated Program

On start-up, initialization software (part of the standalone utilities) establishes a wakeup interrupt timing, time lines, priority queues, and initial conditions for the pre-emptible subsystems, and the manager/scheduler enters a ready state. As illustrated in Figure 3-2, initial entry is to the background, which waits for the first interrupt or other wakeup action.

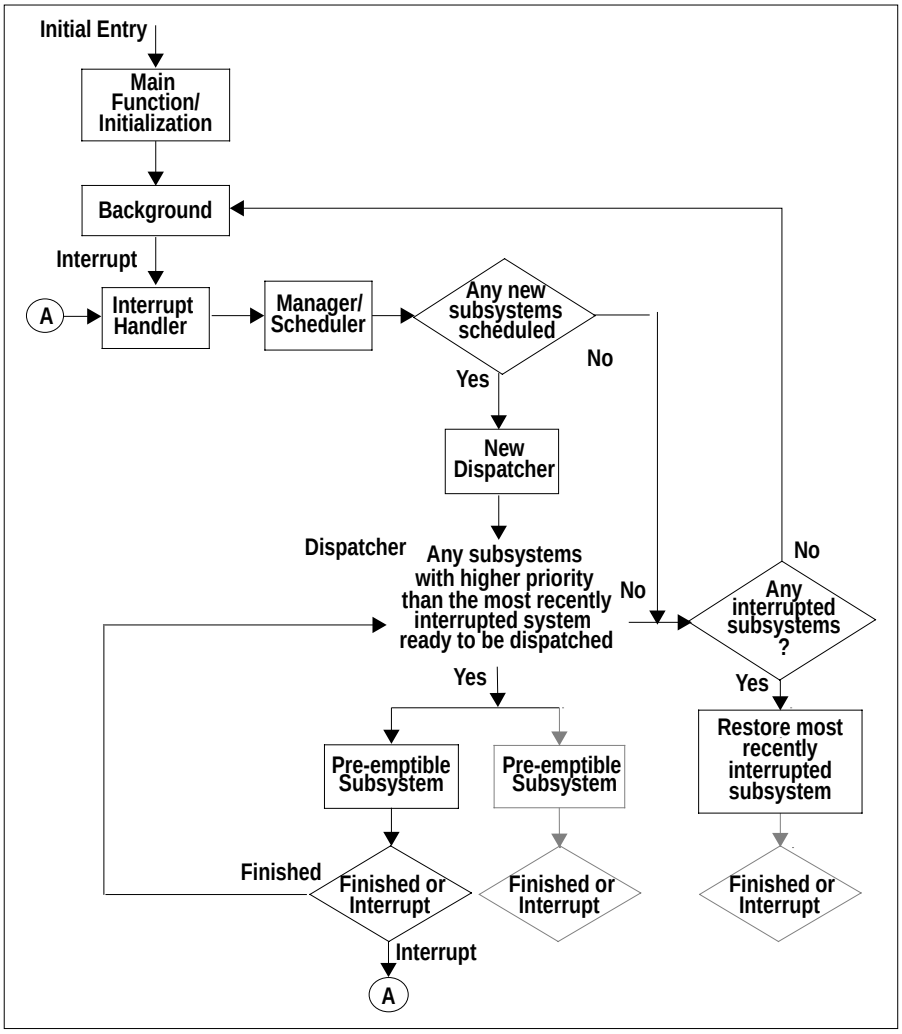


FIGURE 3-2 Flow of Control in the Generated Program

When the wakeup is received, the interrupt handler saves the interrupted context, if necessary, and passes control to the manager/scheduler. The scheduler checks external inputs and establishes a list of subsystems to be dispatched. It then posts any external outputs and performs certain housekeeping before passing the dispatch list to the dispatcher.

The dispatcher is bascially a big switch that passes control to the subsystem in the dispatch list that has highest priority. It always checks to see if there are any previously dispatched, but interrupted, subsystems with higher priority before dispatching a newly scheduled subsystem from its dispatch list. If there are, the most recently interrupted subsystem (which should be of highest priority among previously interrupted subsystems and newly scheduled subsystems) is restored by the interrupt handler and allowed to continue.

When the subsystem is finished, it passes control back to the dispatcher, which dispatches or restores the next-highest-priority subsystem, and so on. If all the currently dispatched subsystems and previously interrupted subsystems finish before a new timer interrupt is received, the interrupt handler returns control to the background. However, if a subsystem is still processing when the next timer interrupt is received, control passes to the interrupt handler, which again passes control to the manager/scheduler, which executes again.

In the case of Ada code with tasks representing subsystems, the dispatcher simultaneously dispatches all tasks that are ready. The Ada tasks have priorities associated with them which determine the CPU availability for each task.

If the manager/scheduler has nothing to schedule at the next timer interrupt, the scheduler passes control back to the interrupt handler. The interrupt handler then restores whatever was running at the time of the interrupt. If any subsystems or the dispatcher were interrupted part-way through their execution, the interrupt handler passes control back to whatever was running (subsystem or dispatcher) at the time of the interrupt. If no such dispatchers or subsystems remain, control returns to the background.

3.2 Sequence of Scheduler Operations

Figure 3-3 on page 3-6 illustrates the sequence of operations of the real-time application scheduler. The scheduler is represented as a bubble diagram (although it is not strictly a finite state machine), because during the "dispatch subsystems" phase (Bubble 9 in Figure 3-3), operations can be interrupted. During the critical section, however (Bubbles 1-8), it operates in the manner of a state machine. In the discussion that follows, the term scheduler is sometimes used to refer to the critical section and dispatcher refers to the interruptible section.

Because the first eight steps in the scheduler's operation are non-interruptible, critical steps, it is an Integrated Systems policy to optimize critical code for maximum performance and to execute in the same amount of time on every cycle. This minimizes output jitter and avoids performance problems.

1. Read External Inputs

   Bubble 1: On entry or re-entry, the scheduler collects the system external inputs so they can be used by the scheduled subsystems without a minor cycle delay. By definition, the minor cycle time of the application is the minimum scheduler cycle, a timing interval created from the sampling rates and timing requirements of all of the SuperBlocks in the system. The scheduler executes exactly once during each minor cycle.

2. Check Triggers and Enables

   Bubble 2: The scheduler prepares for scheduling the subsystems by first determining which triggered and enabled subsystems are eligible to execute during this minor cycle. This step is not performed for systems that have no enabled or triggered subsystems. For enabled subsystems, the scheduler checks the subsystem state. If the state is blocked (that is, ready to run but waiting for an enable signal), the subsystem is ready to execute. If the enable signal is set, the scheduler queues it for execution.

   However, if the subsystem is in an idle state and the enable signal is true, the subsystem scheduler must determine whether the correct time has arrived for this subsystem to execute. If it is not yet time, the subsystem waits in the idle state until it is time to execute. See Section 3.3.2 on page 3-11 for further details.

   For triggered subsystems, if the trigger signal is true, the scheduler checks the subsystem state and proceeds appropriately for the subsystem type. At this stage, when a triggering signal is received and the triggered subsystem is in a blocked state waiting for a trigger, the scheduler queues it for execution. If the

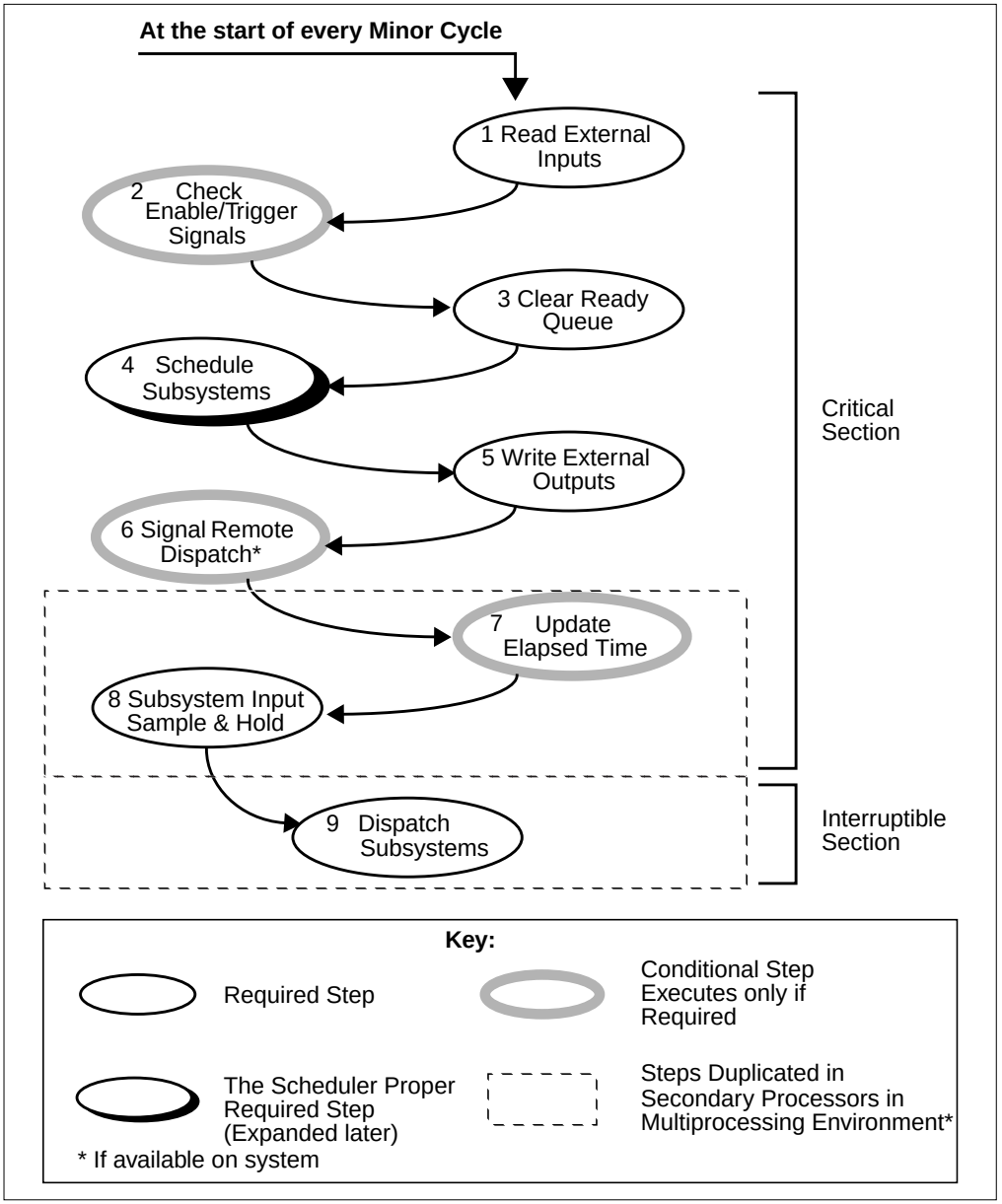


FIGURE 3-3 Scheduler Operation

triggered subsystem is in an idle state, the subsystem is also queued for execution, but the outputs are posted or not posted, depending on the type of the subsystem. See Section 3.3.3 on page 3-14 for further details.

3. Clear the Ready Queue

   Bubble 3: The scheduler clears the ready queue for all the subsystems. The ready queue is established by the scheduler in Bubble 4 and used in Bubble 8 to determine which subsystems are to have their input sample-and-holds updated.

4. Schedule the Subsystems

   Bubble 4: The scheduling algorithm is performed. For scheduling each of the subsystems, the scheduler checks for timing overflow (when a subsystem is still running, even though it is time to start running its next cycle; that is, the duration of the subsystem's execution has lasted longer than its cycle time). For each subsystem, the scheduler also checks all the criteria that determine whether the subsystem is to be dispatched.

   These criteria are:

   - Continuous

     The continuous task is dispatched (through the integrator) every time the dispatcher is invoked.

     An element of the scheduler is the Integrator (see Figure 3-4 on page 3-9). It performs continuous, fixed-step integration of states and implicitly dispatches the continuous subsystem to perform the state and output updates. The integrator/continuous task pair is by default treated as the fastest task to be dispatched by the scheduler. You can also specify the rate with the `-csi` option, but it must be at least as fast as the fastest periodic task in the model. For details regarding continuous code generation, see Chapter 5.

   - Free-Running Periodic

     The appropriate time has arrived, as determined by the time line table and the elapsed time counter.

   - Enabled

     The correct time has arrived, the enable signal is still true, and the subsystem was in the *Idle State* in the previous minor cycle (the state in which the enable signal is true, but the correct time has not yet arrived).

or:

The enable signal has just become true and the subsystem was in the *Blocked State* in the previous minor cycle (the state in which the enable signal is false).

- Triggered

  The trigger signal has transitioned from false to true since the start of the last minor cycle. This condition is also checked for in Bubble 2. If the triggered block type is Asynchronous, and if the subsystem is to be dispatched by the Scheduler (that is, if the triggering signal is not an External Input), the subsystem triggers if the triggering signal has transitioned either from false to true or from true to false since the start of the last minor cycle. If the triggering signal for the Asynchronous subsystem is an External Input, the subsystem is dispatched separately from the Scheduler; see *Properties of Asynchronous Subsystems* on page 3-17 for details.

  Based on these criteria, the scheduler builds up the ready queue and adds the subsystems that are to execute to the dispatch list. The difference between the dispatch list and the ready queue is that subsystems that were on the dispatch list, but not dispatched on the previous cycle, are brought forward on the dispatch list for dispatching on this cycle or later. By contrast, the ready queue is cleared and built up again on every cycle.

  As indicated in Bubble 4 and explained in more detail in the section that follows, the scheduler uses the computational attributes of the subsystems to establish the priority for dispatching the subsystems. The computational attributes include the sample rate and the type of the subsystem. The priority sequence established using the computational attributes runs from fastest sample rate or timing requirement to slowest. For a tie in sample rate or timing requirement, the priority for execution is based on the type of subsystem, from highest to lowest:

  Continuous

  Free-Running Periodic

  Enabled Periodic

  Triggered Asynchronous

  Triggered As-Soon-As-Finished

  Triggered At-Timing-Requirement

Triggered At-Next-Trigger

Using the above priority rules, the subsystems are assigned their task IDs in the sequence 1, 2, 3, ... NTASKS, with the highest priority subsystem getting the lowest ID number (that is, 1) and the lowest priority subsystem getting an ID of NTASKS. NTASKS is the number of tasks in the system and is also the name of the variable in the code which represents that number. The task IDs are assigned at code generation time.

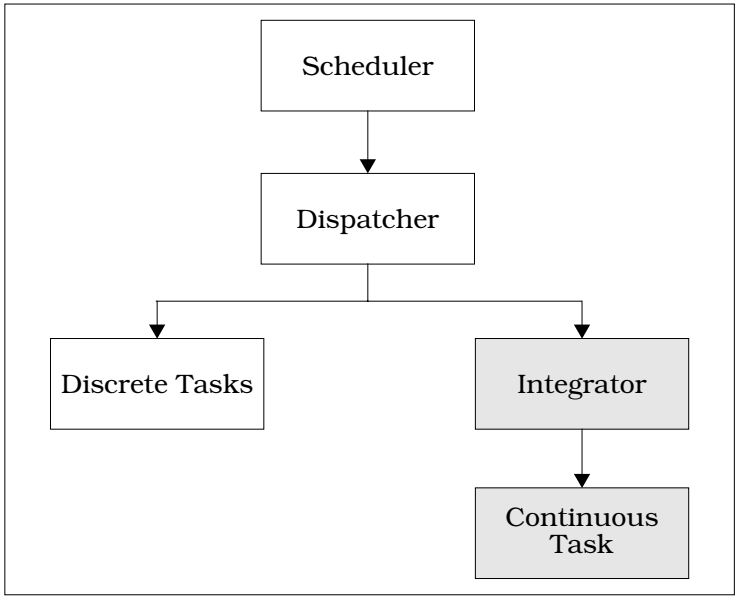


FIGURE 3-4 Scheduler Architecture

5. Write External Outputs and Signal Remote Dispatch

   Bubbles 5 & 6: The scheduler calls the external output routine to post all subsystem outputs at every minor cycle. In multiprocessor implementations only, the dispatch list and a remote dispatch signal are posted to the secondary processors (Bubble 6) to signal the availability of the dispatch list and mark the start of subsystem execution.

6. Update Elapsed Time and Sample & Hold

   Bubbles 7 & 8: In multiprocessor implementations, every secondary processor's scheduler subsystem may or may not need to perform an elapsed time update

(Bubble 7), but will be required to perform a "sample and hold" subsystem input (where the scheduler reads the inputs and latches them for use by the subsystem, Bubble 8) and a subsystem dispatch (Bubble 9). The scheduler updates the elapsed time counter (Bubble 7), if required. Bubble 8: the scheduler in each processor consults the ready queue to perform a sample-and-hold on the subsystem inputs for the subsystems that have been added to the dispatch list. For determinacy reasons, subsystems remaining in the dispatch list from a prior cycle will not have their inputs sampled again. This step is the end of the critical section.

7. Dispatch Subsystems

   Bubble 9: The dispatcher is re-entrant and it can be interrupted at any point in its operations. This re-entrant step accepts the dispatch list and queues the subroutines for execution. This step also includes the execution of subroutines, which can be interrupted/pre-empted at any time.

3.3 Properties of Scheduled Subsystems

A scheduled subsystem is viewed as a finite state machine that is represented as a State Transition Diagram (STD). A finite state machine always exists in exactly one of its defined states, where it remains until some change forces it to transition to another state. No more than one transition can take place on each cycle of the subsystem in which the STD resides. These STDs have the same timing conventions as other state machines in SystemBuild. The STDs are illustrated in Figure 3-5, Figure 3-6 on page 3-12, and Figure 3-8 on page 3-15; associated timing diagrams are shown in Figure 3-7 on page 3-13 and Figure 3-10 on page 3-17.

3.3.1 Free-Running Periodic Subsystems

Figure 3-5 shows the operation of a free-running periodic subsystem. A free-running subsystem is always enabled and it exists in its idle state until the scheduler decides it is time for the subsystem to run (i.e., that its sample time has arrived). At that time, the subsystem posts its outputs and enters the running state. This also applies on start-up, when the subsystem theoretically has not yet executed and has no outputs to post; the simulation user can control the way that this start-up output is generated by using the `sim...{initmode}` command (see the *SystemBuild User's Guide*).

A subsystem accepts its inputs just before starting; this event is associated with the transition before the running state is entered. With specified exceptions, the subsystem posts its previous outputs at that same time (that is, just before running).

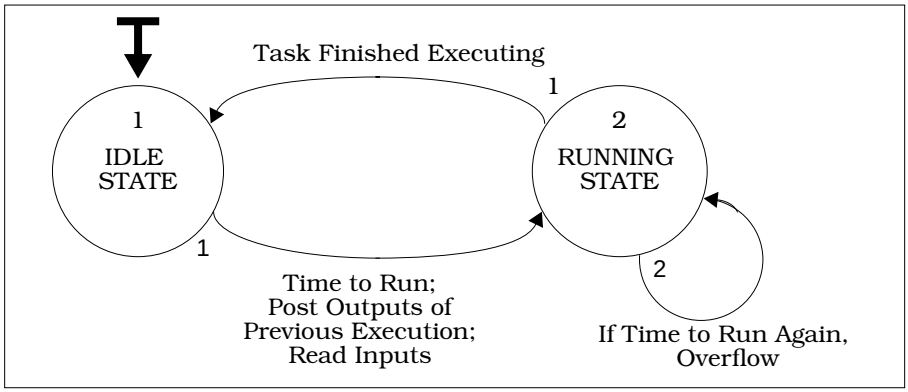


FIGURE 3-5 Free-Running Periodic Subsystem as a State Machine

The exceptions are associated with enabled and triggered subsystems and are shown in the State Transition Diagrams.

When a subsystem is running (that is, manipulating its outputs), the outputs of the previous cycle are latched (double-buffered) and can be read. Note the overflow condition that occurs when a subsystem is in the running state and it becomes time for it to start running again. This means that it did not finish its execution in time. This is a fatal error in systems that must operate in a real-time environment. When the subsystem finishes, it performs the appropriate housekeeping subsystems and then returns to the idle state; but for reasons of determinacy, it does not post its new outputs until it is started up again.

3.3.2 Enabled Periodic Subsystems

Figure 3-6 on page 3-12 shows the state diagram for an enabled periodic subsystem, illustrating the blocked state during which it is disabled.

In simulation, these subsystems can be scheduled to run only on a predefined time line established by the subsystem's sample rate and when the subsystem is enabled at the same time. If a disabled subsystem is enabled after its synchronous start time in the time line, it has to wait until its next major cycle (the repetition time of the enabled periodic subsystem) to run. The blocked state is used in the generated code to eliminate latencies that would occur if a disabled subsystem were to receive its enable signal several minor cycles before its synchronous start time in the time line (its major cycle).

When the enable goes true while the subsystem is in the blocked state, the subsystem is scheduled to run at the next minor cycle; if it has priority over other contending subsystems, it is executed immediately, at the same minor cycle of the real-

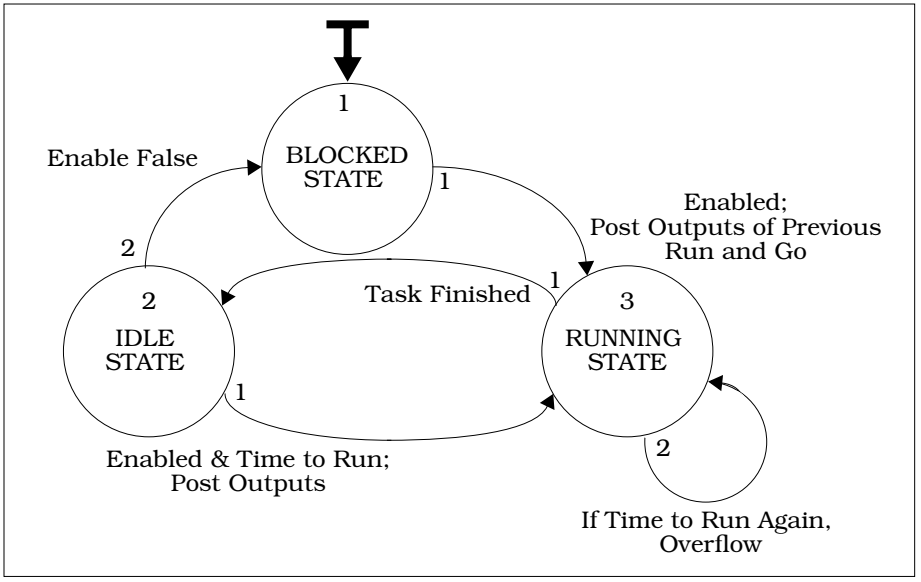


FIGURE 3-6 Enabled Periodic Subsystem as a State Machine

time scheduler. The behavior of the scheduler can differ, however, depending on whether the enable signal is generated internal to the system (as the output of another subsystem) or presented as an external input to the system. See the timing diagram (Figure 3-7 on page 3-13) for these distinctions.

In the example shown in Figure 3-7, the major cycle is three times as long as the minor cycle of the system. When an external enable signal is detected at the beginning of a major cycle, (as shown at point A); the enabled subsystem is dispatched for execution immediately.

In Figure 3-7, item B shows the case in the generated code concerning enables that are presented as external inputs, asynchronous to the original time line. The scheduler attempts to process these inputs as quickly as possible. This is also shown in Bubble 2 of Figure 3-3 on page 3-6, where external inputs are gathered immediately upon initialization of the scheduler, so they can be processed without delay. In Figure 3-7 at B, in the generated code the subsystem is queued for execution at the minor cycle following the instant when the enable signal was detected true. Note the timing window indicated at C. During that interval, it does not matter whether the external enable is presented synchronously with the minor cycle or before it; both the simulation and the generated code operate identically.

At D, the signal appears at the start of a major cycle and is applied immediately. At E, the signal appears one minor cycle after the start of a major cycle and incurs a two-minor-cycle delay.

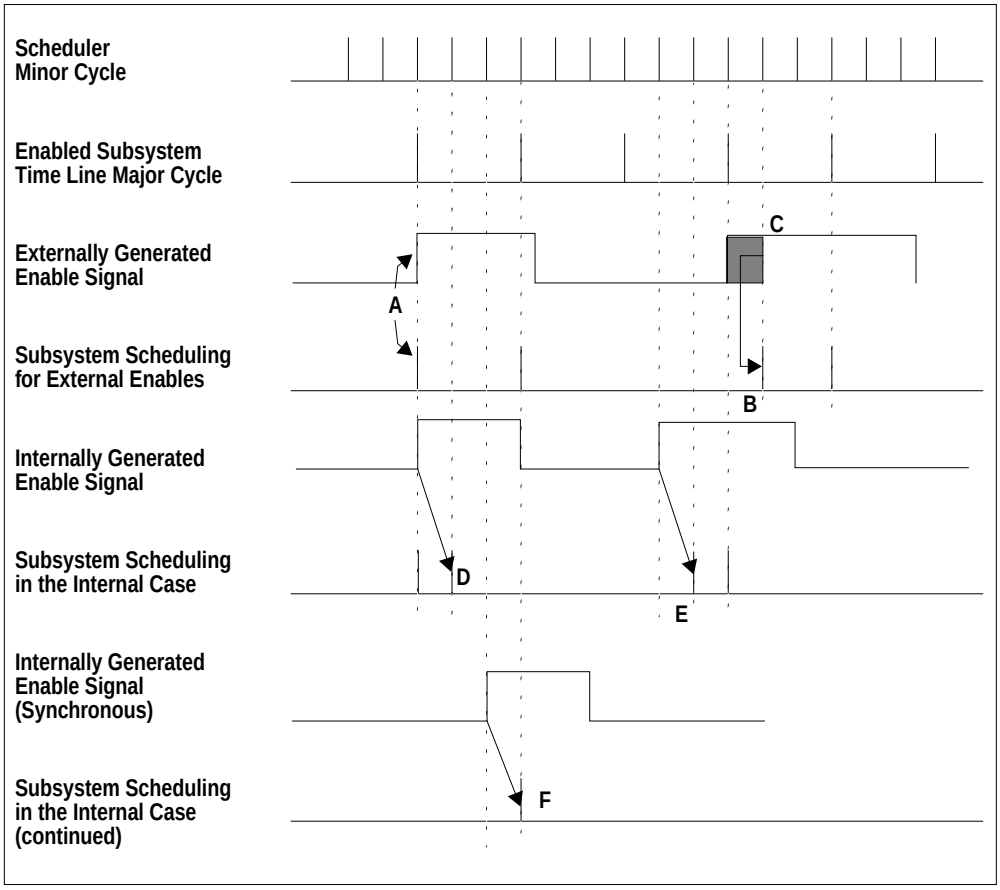


FIGURE 3-7 Enabled Subsystem Timing

At F, the enable occurs one minor cycle before the beginning of the major cycle and consequently, the delay is one minor cycle. By contrast, at points D, E, and F, illustrating the operation of generated code, the internally generated enable signal is also synchronized through the application scheduler. It is always delayed exactly one minor cycle before being applied. The reason for this delay is implied in Bubble 8 of Figure 3-3. Here, input sample-and-holds are performed after the determination of which subsystems are to be dispatched for execution on this (the first) minor cycle, so the application of the synchronous enable in generated code is always deferred for one minor cycle.

3.3.3 Triggered Subsystems

Figure 3-8 on page 3-15 shows the State Transition Diagrams for triggered, asynchronous subsystems. The types of triggered subsystems differ mainly in the manner in which they post their outputs. Just like enabled subsystems, certain types of triggered subsystems have a blocked state, from which they can be invoked immediately when the trigger is detected. (By default, the trigger is defined as the positive-going edge of the triggering signal, but provision is made in the template files for changing it to detect a negative-going edge.) The properties of the types of triggered subsystems are illustrated as State Transition Diagrams in Figure 3-8 on page 3-15, and Figure 3-9 on page 3-16 (for aysnchronous), and in a consolidated timing diagram in Figure 3-10 on page 3-17. These properties are summarized as follows:

At Timing Requirement (ATR)	Specify a timing requirement (number of system minor cycles) in the SuperBlock block form. The outputs are always posted exactly that number of cycles after the subsystem is triggered for execution. ATR is used especially in systems where determinacy must be guaranteed.
At Next Trigger (ANT)	The subsystem only posts its outputs when it is next triggered for execution, however long that may be. ANT is used for modeling certain kinds of variable rate, but repetitive activities, such as a shaft that rotates at a variable speed. This type of subsystem has no blocked state.
As Soon As Finished (SAF)	The outputs are posted at the beginning of the minor cycle after the subsystem finishes running. SAF is used for maximum performance, but might compromise determinacy.
Asynchronous (ASYNC)	If dispatched by the scheduler, the outputs are posted at the beginning of the next minor cycle after the subsystem finishes running. This will occur if the triggering signal is an output of another subsystem. If dispatched as a result of an aysnchronous interrupt, the outputs are posted as part of the interrupt handler, and are available to scheduled systems immediately.

In Figure 3-10 on page 3-17, a timing requirement equal to four scheduler minor cycles has been established. When the trigger signal is received, the subsystem is started at the next scheduler minor cycle (point A1). In this example, although the

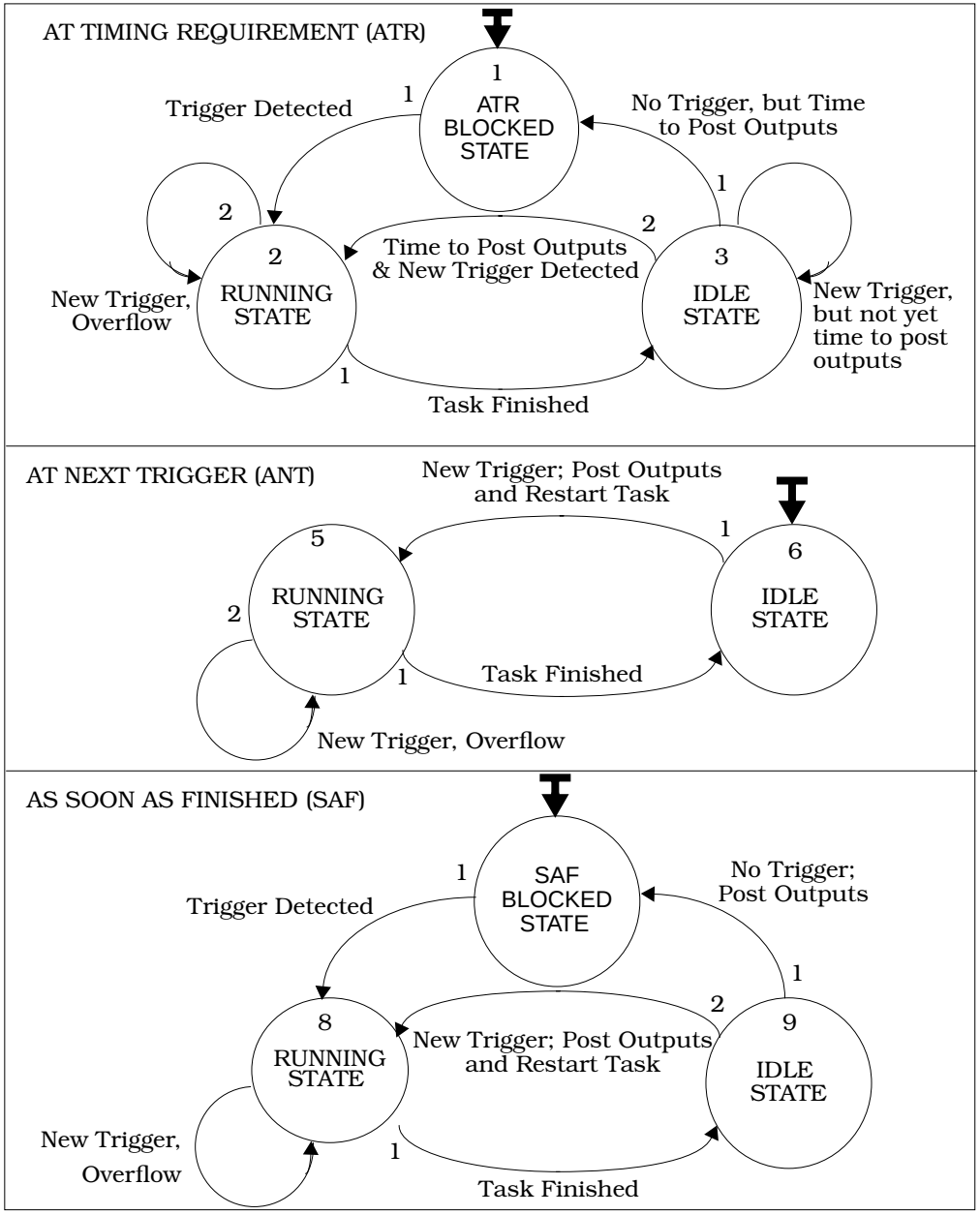


FIGURE 3-8 Triggered Subsystems as State Machines

timing requirement is four (point A2), the subsystem actually completes execution in slightly less than three cycles. This illustrates the kinds of output posting:

- ASYNC - The output is seen at point B, three cycles after the subsystem was started, assuming this subsystem was dispatched by the scheduler.
- SAF - The output is seen at point B, three cycles after the subsystem was started.
- ATR - The output becomes available at point C, exactly four cycles after start-up.
- ANT - The output is not available until point D, when the subsystem is next triggered for execution.

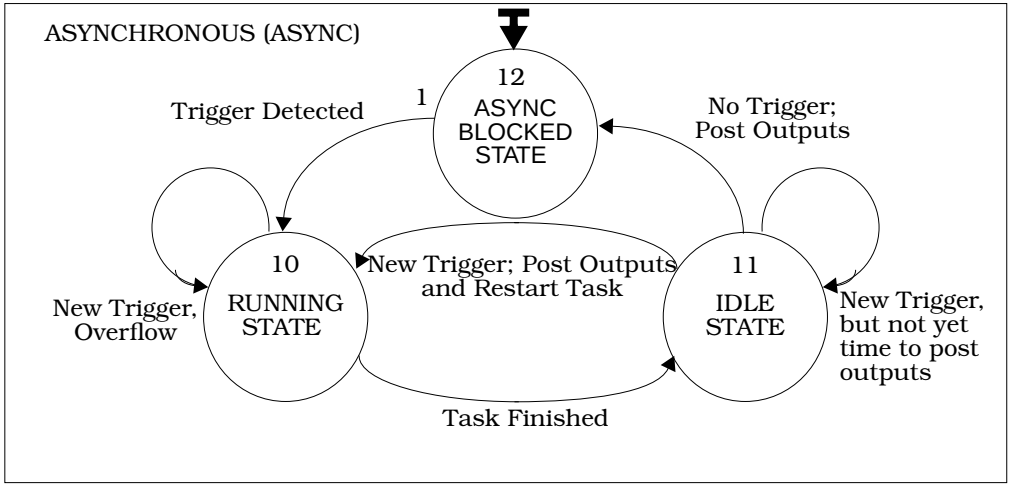


FIGURE 3-9 ASYNCHRONOUS Triggered Subsystems as State Machines

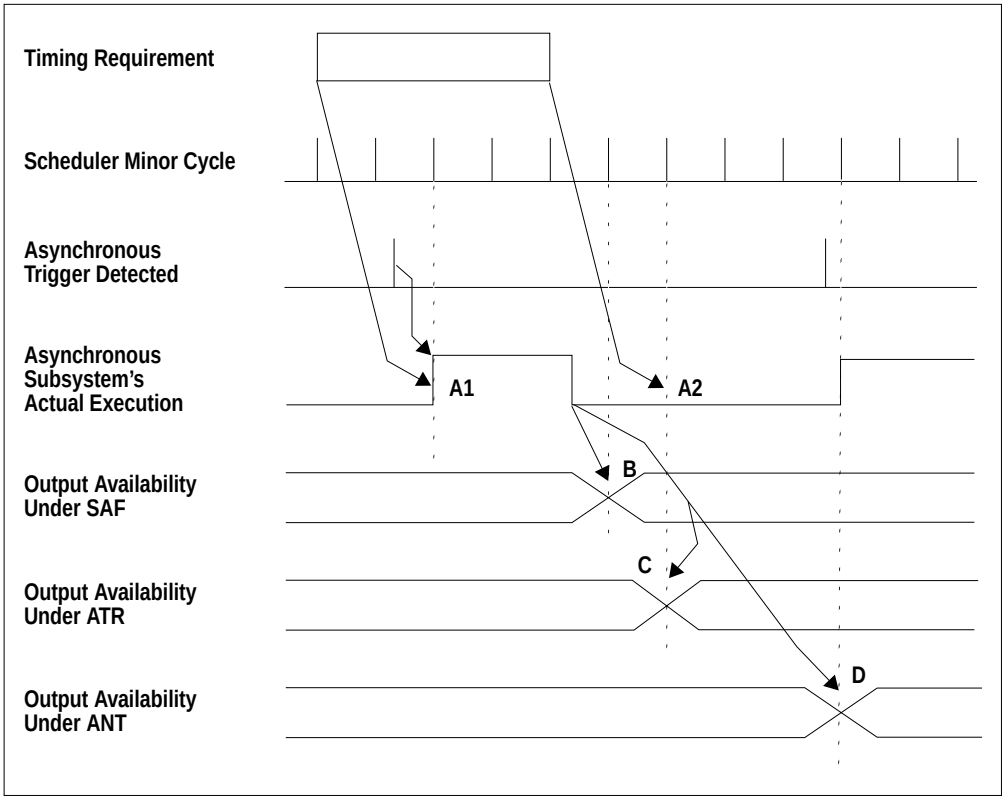


FIGURE 3-10 Timing of Triggered Subsystems

3.4 Properties of Asynchronous Subsystems

Asynchronous subsystems (procedures or asynchronous trigger subsystems) are so called because of their noncyclic or unscheduled execution in a real-time application. These subsystems should be viewed as special purpose entities and should be used accordingly.

The asynchronous subsystems are not directly managed or scheduled by the application scheduler as are the synchronous subsystems. Scheduled subsystems can execute only at the start of a scheduler minor cycle and not instantaneously (for example, at the arrival of an external event). This adds latency to the execution of cer-

tain subsystems, which absolutely cannot wait until the next minor cycle for execution. Asynchronous subsystems are designed to solve this problem.

The kinds of asynchronous subsystems (see Figure 3-11 on page 3-19) that you can generate in an AutoCode application are:

- Start-up procedure
- Asynchronous subsystem
- Interrupt procedure
- Background procedure

3.4.1 Start-up Procedure

This procedure is defined in SystemBuild using the Start-up Procedure SuperBlock. The purpose of this start-up procedure is to initialize the application data at start-up time. This data includes variable block data and %variables represented by variable blocks. It is only via the Startup SuperBlock that you can initialize the %variables in SystemBuild at run time. Usually, the start-up procedure is called at the system initialization phase. Refer to the *SystemBuild User's Guide* for more details on the Startup SuperBlock description.

3.4.2 Asynchronous Trigger Subsystems

Subsystems formed by collections of Asynchronous Triggered SuperBlocks (ATSBs) with the same triggering signal are handled differently depending upon the source of the triggering signal.

If the triggering signal is internal to the model (that is, if the triggering signal is not an external input), these subsystems function similarly to the Triggered - Soon As Finished subsystems with the following exceptions:

- The ATSB subsystems have higher priority than the SAF triggered subsystems (that is, they are executed before other triggered subsystems).
- The triggering signal to ATSB subsystems is double-edged; the ATSB subsystem will be scheduled if its triggering signal transitions from low to high or from high to low during the previous scheduler cycle. This is to maintain compatibility with the use of these subsystems in simulation.

If the triggering signal is external to the model (that is, the triggering signal is an external input), then the ATSB subsystem is handled specially in the template. In this case, two pieces of code are generated; the regular (nonreentrant) triggered subsystem code, and a small (reentrant) wrapper that is designed to function as an interrupt service routine (ISR). This wrapper is wholly generated in the template,

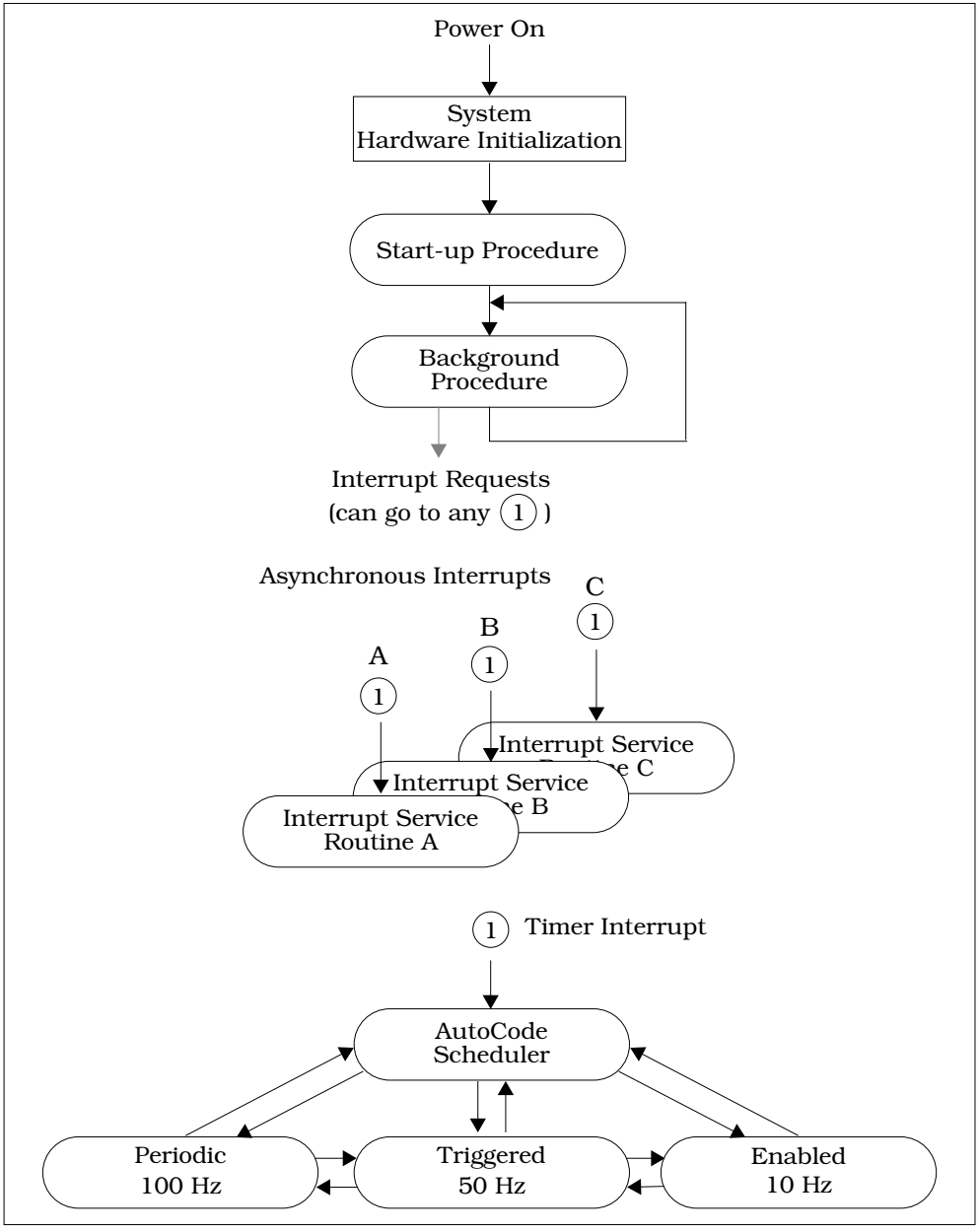


FIGURE 3-11 AutoCode Real-Time Application Execution Sequence

and can be customized for the user's application. In the templates provided by Integrated Systems, the wrapper for the ATSB subsystem does the following:

- Checks for reentrancy, and reports an error if the wrapper is reentered. Note that the wrapper could instead queue calls to the subsystem code, but under no circumstances should the subsystem code be reentered.
- Gathers the subsystem inputs from external inputs and outputs from other subsystems.
- Invokes the ATSB subsystem.
- Posts the ATSB's subsystem outputs, updating the external output structures as necessary.

In the default template, the ATSB wrapper is a procedure requiring neither inputs nor return values.

Note that ATSB subsystems differ from Interrupt Procedure SuperBlocks (see *Asynchronous Trigger Subsystems*) in the following ways:

- The inputs and outputs to these subsystems can be represented in the model like those of other standard SuperBlocks without incurring processing overhead at interrupt time.
- These subsystems are simulatable by the SystemBuild simulator.
- States are supported in ATSB subsystems, but not in Interrupt Procedure SuperBlocks.
- Calls to Procedure SuperBlocks are supported on ATSB subsystems.
- These subsystems are not reentrant, even though the wrapper is reentrant, and thus can be designed in such a way to support asynchronous interrupts that may occur before the processing of the ATSB subsystem code is complete.

For more information on Asynchronous Trigger SuperBlocks, see the *SystemBuild User's Guide*.

3.4.3 Interrupt Procedure

This procedure is defined in SystemBuild using the Interrupt Procedure SuperBlock. The purpose of this SuperBlock is to model the Interrupt Service Routine (ISR). This ISR model in SystemBuild is generated as an interrupt procedure by AutoCode and can be executed on arrival of a specific interrupt signal. Using variable blocks, implicit communication can be established to any other subsystem in the application.

The execution of the interrupt procedure is done directly on arrival of an external interrupt or an event; the AutoCode scheduler does not manage, monitor, or schedule this procedure. However, since the AutoCode scheduler, as supplied, employs rate monotonic scheduling principle, it is possible that interrupt procedure execution will overflow the synchronous task execution. Interrupt procedure users should keep the interrupt procedure execution time to a minimum and should keep enough time buffer in each scheduler cycle for possible execution of interrupt procedures. Please refer to the *SystemBuild User's Guide* for more details on the Interrupt Procedure SuperBlock description.

3.4.4 Background Procedure

This procedure is defined in SystemBuild using the Background Procedure SuperBlock. The purpose of the background procedure is to represent the logic executed when the system is idle, in the background mode of operation. Typically, the background procedure should be executed as the lowest priority task and only if no other tasks need to be performed in the system. Please refer to the *SystemBuild User's Guide* for more details on the Background Procedure SuperBlock description.

3.5 Reentrancy and Preemption: The Dispatcher

The generated application program is interruptible except at the critical section of the scheduler. The scheduler is automatically created by AutoCode, using the template file, to provide input/output calls, scheduling, error handling, and dispatching services for the generated application program. All the services, except for dispatching of the subsystems, are performed in the critical section. The critical section is kept as brief as possible; one of several reasons for this minimization is to allow maximum time for the subsystems to execute.

The subsystems operate under different constraints as compared to those of the time-critical scheduler. A subsystem cannot execute more frequently than the scheduler does and in many cases, it will run far less often. However, the subsystem may require considerable time for each execution pass. Consequently, it can be interrupted repeatedly by scheduler execution. Thus, the subsystem code must be completely interruptible. It must be able to be interrupted by the scheduler and thus to be pre-empted by higher priority subsystems. It must also be able to restart at any point in its operations.

The only timing requirement of the subsystem is that it must finish executing before the next time it is to be queued for execution. A subsystem being ready to run and simultaneously not finished running, defines the condition called "subsystem timing overflow." This is a catastrophic error in any system that requires deterministic operation.

The scheduler can add subsystems to the dispatch list at any cycle of the scheduler's operations. However, the dispatcher only removes the subroutine from the list when the subsystem has begun its operations. Determinacy and proper operation of a pre-empted subsystem both demand that the inputs receive a sample-and-hold to a subsystem only once per execution, when first queued. As a result, a second list, the ready queue, is employed to determine which subsystems will have their inputs sampled and held this cycle. The ready queue is cleared and set up by the application scheduler once every minor cycle.

When the ready queue is cleared, the scheduler determines which subsystems are to be queued for dispatch this cycle and places the subsystems into the ready queue and the dispatch list. Note that the scheduler never removes a subsystem from the dispatch list; only the dispatcher has that duty. The following examples explore these ideas graphically, following the scheduler and dispatcher through a few cycles of operation, showing how the subsystem states, the ready queue, and the dispatch list change over time.

Other examples show the operation of a scheduler with more complex timing requirements ("pseudo-rate scheduler") and its special conditions and error conditions.

3.6 Scheduler Examples

The examples presented in the subsections that follow describe scheduler operation both when sampling rates and timing requirements for all subsystems are common multiples and when they are not common multiples, thus requiring a pseudo-rate for the scheduler. Operating with skew is also discussed.

3.6.1 Dispatching and Pre-emption Example

The illustrations, starting with Figure 3-12, show a system with five subsystems: three periodic, one enabled, and one triggered. The subsystem list identifies which subsystems are periodic, enabled, or triggered, and shows them in priority order.

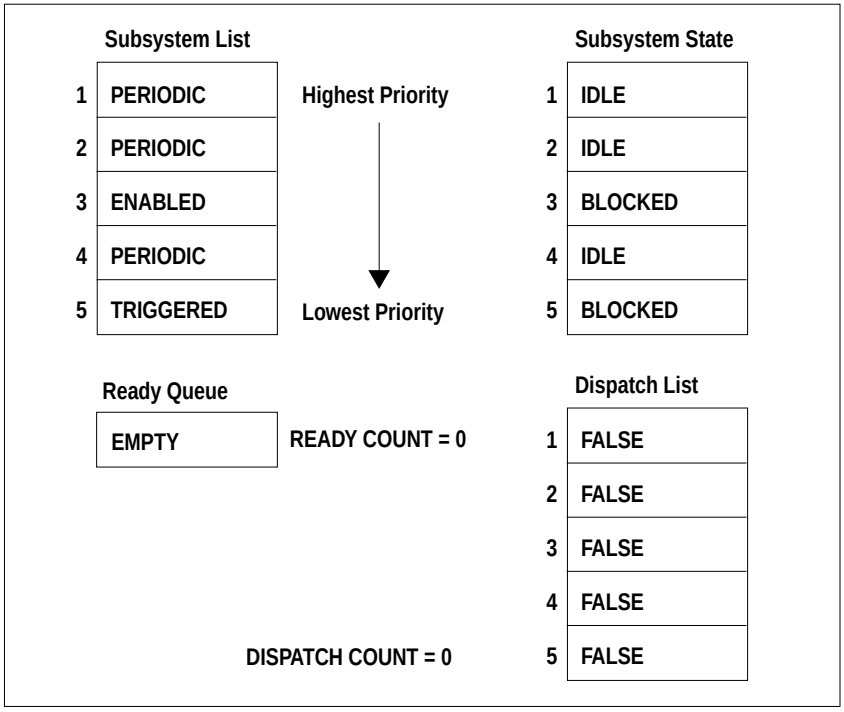


FIGURE 3-12 Scheduler Data Structures at Initialization: 1

The subsystem state list identifies the state of the subsystem. The ready queue is used by the scheduler to determine which subsystems are to have their inputs sampled and held. When the scheduler determines that a subsystem is to be dispatched for execution, the subsystem is placed into this list. When a subsystem is dispatched for execution, it is removed from the list by the dispatcher. In Figure 3-12, the subsystems are idle or blocked, each awaiting its condition to start running.

In Figure 3-13, the time has arrived for subsystem 1 to execute and the enable signal for subsystem 3 has been received. The numbers of the two subsystems are entered into the ready queue in reverse order of priority and the ready count is set to equal the number of subsystems that are ready (that is, 2). The corresponding entries in the dispatch list are marked true. The dispatch count (which is the pointer to the lowest priority subsystem that is ready for dispatch) is set to 3. The immediate effect of this is that the inputs for subsystems 1 and 3 are sampled and held, and control is passed to the dispatcher.

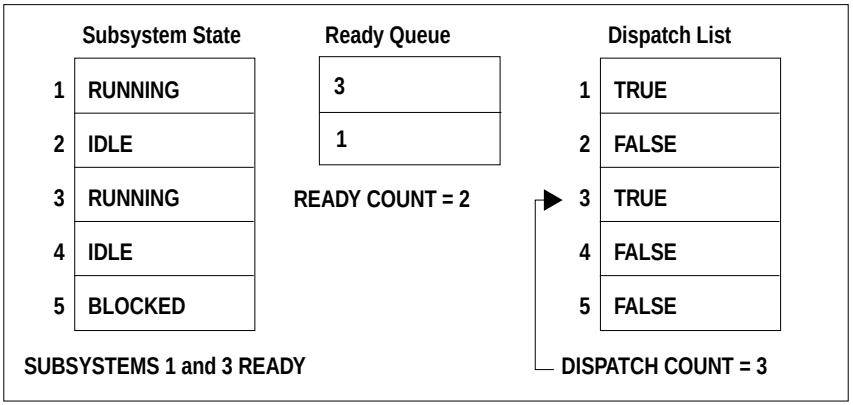


FIGURE 3-13 Scheduler Data Structures: 2

The action of the dispatcher is shown in Figure 3-14, where subsystem 1 (the highest priority subsystem) is actually executing and subsystem 3 is waiting to be dispatched. The dispatch count is still equal to 3, pointing to the subsystem of lowest priority that is still either running or in the dispatch list. When subsystem 1 completes its operations (is no longer running), the system moves to the state shown in Figure 3-15 on page 3-26.

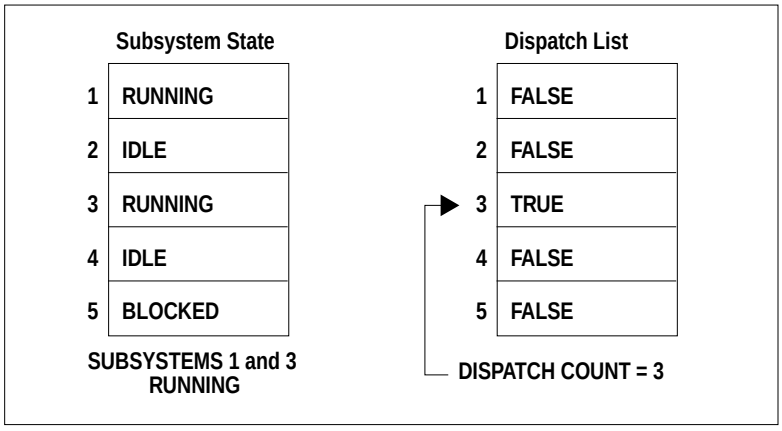


FIGURE 3-14 Scheduler Data Structures: 3

In Figure 3-15, subsystem 1 is marked idle in the subsystem state table and subsystem 3 is dispatched and running. All the entries in the dispatch list are marked false, indicating that no subsystems currently need to be dispatched. However, the dispatch count pointer is still pointing to subsystem 3, indicating that it is not finished executing. Now, while subsystem 3 is still working, a scheduler interruption occurs.

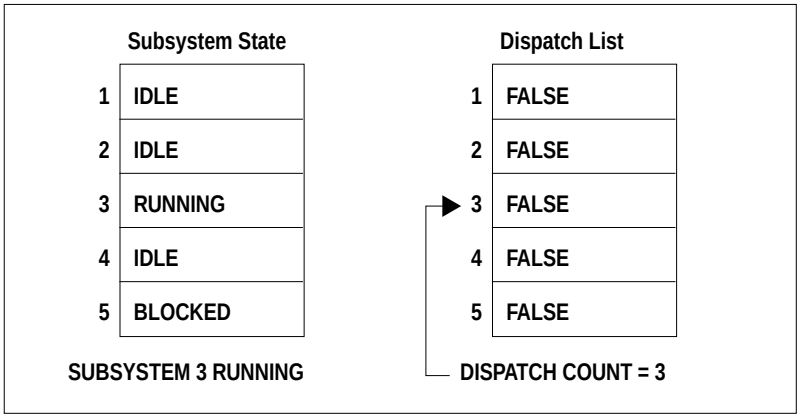


FIGURE 3-15 Scheduler Data Structures: 4

The scheduler is re-entered and, as a part of the scheduling loop, subsystems 1, 2, and 5 are all marked ready (see Figure 3-16). Subsystems 1, 2, and 5 have been entered into the subsystem state list as running and the interrupted subsystem 3 remains in a running state. Therefore, subsystems 1, 2, and 5 are placed in the ready queue to have their inputs sampled and held. Note that subsystem 3 cannot be placed in the ready queue, because it has already received its inputs for this operations cycle. Now, subsystem 1 and then subsystem 2 will be dispatched for execution.

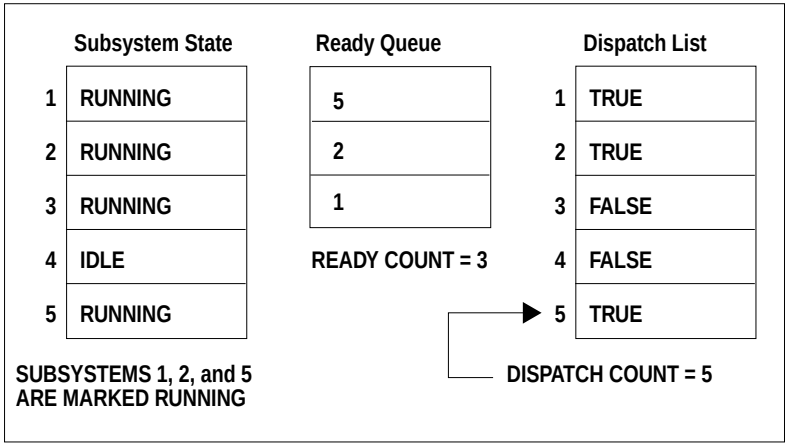


FIGURE 3-16 Scheduler Data Structures: 5

When subsystems 1 and 2 are both finished (see Figure 3-17), the dispatcher notes that subsystem 3 is in a running state, but is not in the dispatch list. The dispatcher checks its records and determines that subsystem 3 was actually in the process of executing when the next scheduler cycle occurred. Thus, the subsystem needs to have its context restored by the interrupt handler, not by the real-time application scheduler. Therefore, the scheduler dispatcher performs an exit, which passes control back to the operating system interrupt handler. The interrupted subsystem is restored from there.

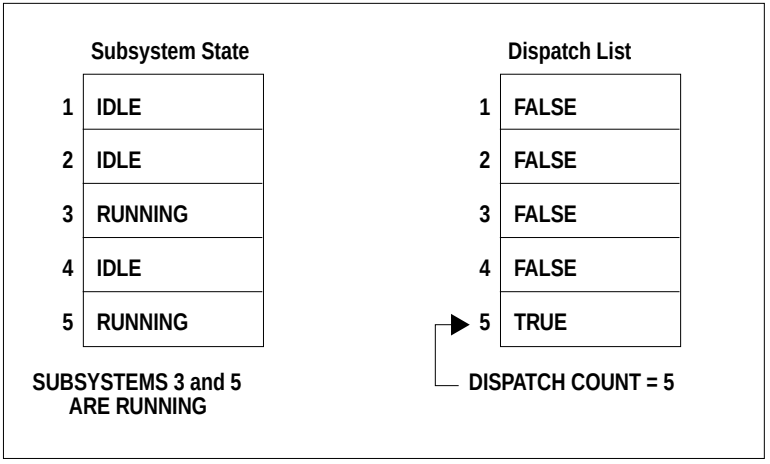


FIGURE 3-17 Scheduler Data Structures: 6

When subsystem 3 is finally finished executing, as shown in Figure 3-18, subsystem 5 is dispatched and executes.

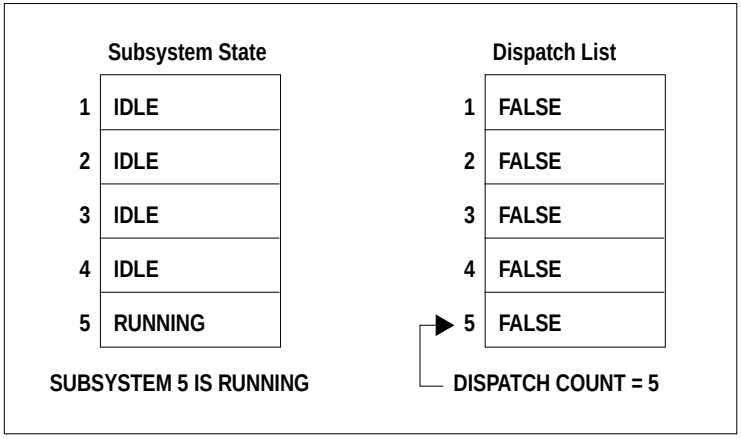


FIGURE 3-18 Scheduler Data Structures: 7

Finally, when subsystem 5 is also finished, the situation is as represented in Figure 3-19. At this point, no subsystems are running, nothing is to be dispatched, and the next interruption for scheduler operation is sometime in the future. The dispatcher responds by passing control to the operating system interrupt handler, which restores and passes control to the background subsystem.

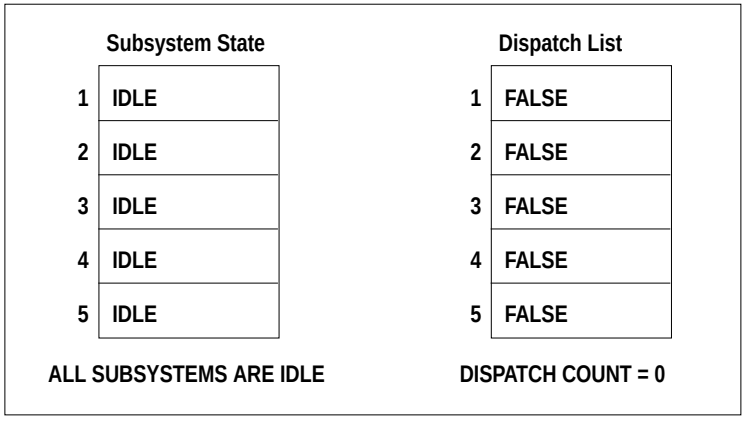


FIGURE 3-19 Scheduler Data Structures: 8

This background subsystem consists of interruptible code that does not return, but waits to be interrupted. It might be nothing but a loop that waits to be interrupted, or it might perform any of a range of low-priority program tasks, such as self-diagnosis and updating displays. For example, some Interactive Animation displays are updated by the background subsystem.

The operation of the scheduler and the subsystems in this example are shown in the form of a timing diagram in Figure 3-20. The numbers in circles correspond to the numbers at the end of each figure title associated with each of the figures for this example.

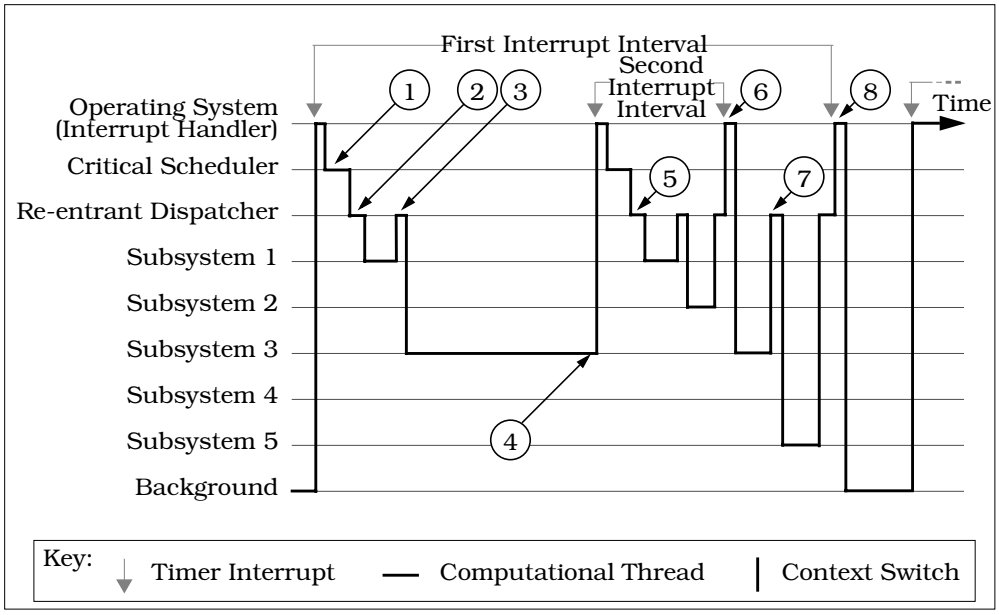


FIGURE 3-20 Dispatcher Example as a Timing Diagram

3.6.2 Pseudo-Rate Scheduler

The previous example assumed that the repetition rate of the scheduler (the scheduler minor cycle) was the same as the sampling rate of the fastest subsystem, subsystem 1. This correspondence holds true only if the sampling rates and timing requirements of all of the subsystems are common multiples. Thus, in the example, the sampling rate of subsystem 1 might be 1 unit, the sampling rate of subsystem 2 might be 2 units, and that of subsystem 4 might be 3 or more units.

However, if the sampling rates or timing requirements of the subsystems are not even multiples, AutoCode establishes a "pseudo-rate" for the scheduler, based on the least common multiple of the rates of the subsystems. The simplest case of a pseudo-rate is shown in Figure 3-21, where there are two free-running subsystems: subsystem 1 with a sampling rate of 2 and subsystem 2 with a rate of 3. At time step 1, the scheduler places both subsystems into the dispatch list and they are both executed. But at step 2, the time to execute has arrived for neither of them and nothing runs; the same thing is true at step 6. However, the scheduler still must complete its cycle of operations, that is, the first 7 steps of Figure 3-3 on page 3-6, even though there is nothing ready to be scheduled. The cycles thus wasted might have a negative impact on system performance. For this reason, use sampling rates that are all even multiples in systems where performance is an issue.

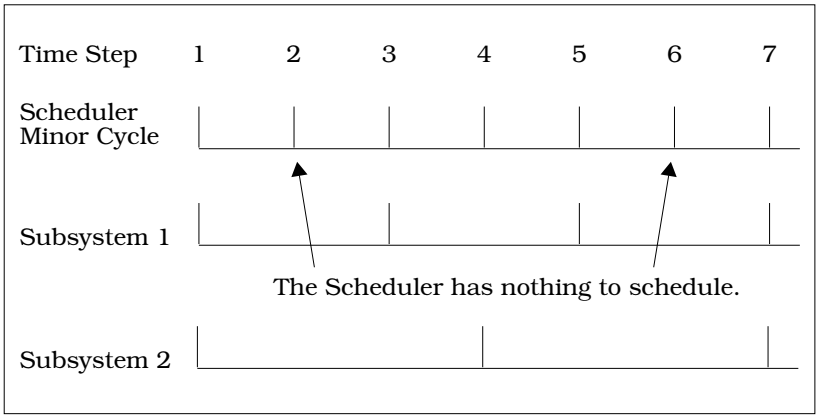


FIGURE 3-21 Dispatcher Operation with a Pseudo-Rate Scheduler

What happens at the beginning of step 2 depends on whether or not the subsystems had completed running before the end of the last cycle. If a subsystem (subsystem 2, presumably) had still been running when the scheduler interruption occurred, at the time the scheduler completed its cycle with nothing in the dispatch list, the dispatcher would pass control back to the interrupt handler. The interrupt handler would invoke the operating system to restore the interrupted subsystem and pass control to it. If all subsystems had finished operation, then it would have been the background subsystem that was interrupted. The operating system would restore this subsystem instead and pass control to the background.

3.6.3 Operating with Skew

Skew, or `First Sample` in the SuperBlock block form, is a method for controlling the operation of the subsystems on the time line. The sampling interval lets you specify the periodicity of the subsystem, and the first sample lets you establish an offset from the beginning of the minor cycle on which the subsystem first becomes eligible to execute. One of the uses for skew is to force a slower-rate subsystem to execute before a faster-rate one, when either could run on the same cycle.

Refer to Figure 3-22. At A, subsystem 1 is running at twice the sampling interval of subsystem 2. Therefore, subsystem 1 has priority. It is assumed that both subsystems receive external input data and that each subsystem posts outputs to the other. At time step 1, when the system is started, both subsystems receive as internal, sampled-and-held inputs whatever initial states you might have defined. But at time step 3, the outputs of subsystem 2 from its first major cycle are latched as inputs to subsystem 1 and the system is running in sync. The outputs from subsystem 1's operations during its secondary cycle are latched and fed to subsystem 2 to serve as its inputs at time step 3. From that time forward, the outputs of every major cycle of subsystem 2's operations are presented as inputs for the next (odd-numbered) cycle of subsystem 1's operations (steps 3, 5, etc.). Those same inputs are still visible to subsystem 1 on its next even-numbered cycle (steps 4, 6, etc.).

And subsystem 1's outputs from the even-numbered (minor) cycle serve as inputs to subsystem 2 on the odd-numbered (minor) cycle (latched at steps 3, 5, etc.).

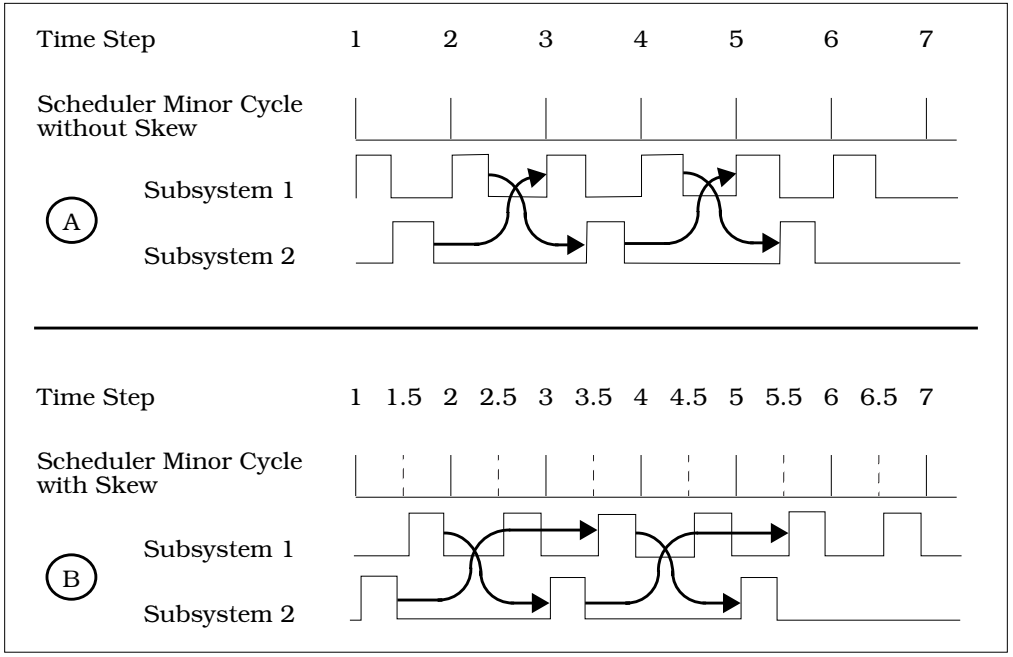


FIGURE 3-22 Operation of Skew

Subsystem 2 never sees the outputs of subsystem 1 from its odd-numbered cycle, for there is no sample-and-hold performed between the end of subsystem 1's odd-numbered step and the beginning of subsystem 2's operations. In the bottom part of the figure, at B, a skew of 0.5 has been added to the timing properties of subsystem 1 and no other change has been made. But the system will operate quite differently from A. First, the scheduler minor cycle is now a pseudo-rate, introduced so that the start-up of subsystem 1 can be scheduled and dispatched correctly. Also, characteristic of pseudo-rate schedulers, there are steps where the scheduler has no scheduling to do (steps 2, 4, 6, etc.). Even so, the critical part of the scheduler must go through its full cycle of operations. This, of course, has an impact on overall system performance.

Observe that even though subsystem 1 would have priority, subsystem 2 starts executing before subsystem 1 in this example, because subsystem 2's time to run arrives before that of subsystem 1. Thus, at step 1, subsystem 2 starts up with

initial states as its inputs and posts its outputs at time step 3 to be latched and made inputs to subsystem 1 at time step 3.5.

Now, as at A, subsystem 1 will run through two full cycles of its operations before subsystem 2 can run again. When subsystem 1 starts at timestep 1.5, its outputs are posted at timestep 2.5, and when it starts at timestep 2.5, its outputs are posted at timestep 3.5, etc. Unlike example A, the outputs from subsystem 1's odd numbered cycles are visible to subsystem 2 at time steps 3, 5, etc., and the outputs of subsystem 2 posted at time steps 3, 5, etc. are presented to subsystem 1 at time steps 3.5, 5.5, etc. Consequently, the subsystems are synchronized differently in the two examples and the two systems can be expected to behave in very different ways at the micro-level.

3.7 Scheduler Errors

The exact method for controlling the scheduler minor cycle interruptions is implementation-dependent. In the absence of standardization of the hardware and software for this and other functions within an embedded system, Integrated Systems has not attempted to furnish a timing simulator, choosing instead to emphasize functional simulation. This is one major reason for our emphasizing the development of rapid prototyping or test bed systems, which can help you evaluate the performance aspects of systems where simulation cannot easily reach. However, we can postulate two kinds of timing problems that could be detected. The application scheduler traps both: scheduler overflow and subsystem overflow.

3.7.1 Scheduler or Subsystem Overflow

- Scheduler Overflow - If the non-interruptible critical section of the scheduler is running and an interruption for the scheduler occurs, then the scheduler is receiving more interrupts than it can handle. To prevent this, the length of the scheduler minor cycle must be increased, the interrupt timer rate must be decreased, or a faster processor must be obtained.
- Subsystem Overflow - Might be intermittent or rare, might or might not be a catastrophe in the context of a given system, and flexible means for dealing with it are provided. Subsystem overflow is defined as a subsystem ready to run and

still not finished running. Figure 3-23 shows a graphical representation of this condition.

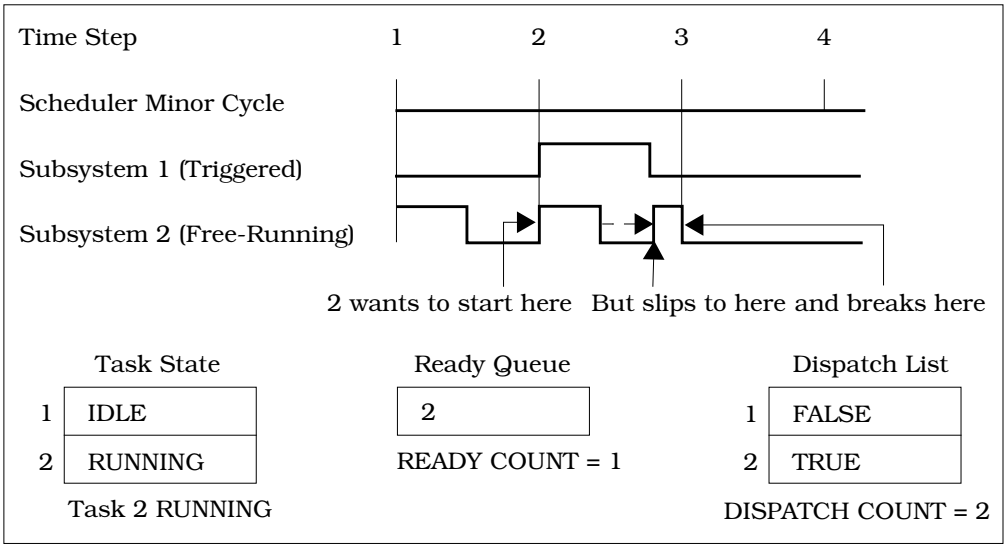


FIGURE 3-23 Subsystem Overflow Example

In Figure 3-23, subsystem 2 is free-running with an intermediate sampling rate. Subsystem 1, with a shorter timing requirement and therefore higher priority than subsystem 2, is triggered and runs only occasionally. When it does run, at step 2, it takes a considerable amount of time. When it finishes and subsystem 2 starts, there is not enough time for subsystem 2 to finish before the scheduler's interruption for step 3 is received. This would be acceptable, because subsystem 2 is required to be interruptible. However, at step 3, the scheduler notes that it is time for subsystem 2 to run again and enters it into the ready queue and the dispatch list. The scheduler also notes that subsystem 2 is not finished and that is where the problem begins. The scheduler cannot post sample-and-hold inputs for a subsystem that is not finished, resulting in a subsystem overflow. For some systems, the subsystem overflow is not critical.

3.7.2 Examples Where Overflow is Irrelevant or Cannot Happen

- A triggered subsystem with output posting As Soon As Finished cannot overflow.
- The background cannot overflow.

- In a subsystem where timing is not really critical, you might wish to disable the overflow indication, or to give it a slower sampling rate. Provision is made for customizing handling of the subsystem overflow error in the template files.

Most blocks in SystemBuild operate in a largely synchronous manner, executing once each time the subsystem is dispatched and contributing little to the generation of intermittent overflows. Even so, several conditions can contribute to the generation of subsystem overflows:

- A heavy load of triggered asynchronous events (see Figure 3-23 on page 3-36).
- While blocks execute a user-defined number of times in a given subsystem. If the number of iterations is variable, the amount of time to execute the subsystem becomes nondeterministic.
- If/else blocks have different logic depending on which branch is selected. If some branches have significantly more processing than others, the amount of time to execute the subsystem becomes less deterministic.
- User-supplied I/O drivers, which have variable execution time, such as a pulse-width-modulation driver that immediately returns if the duty cycle has not changed. Integrated Systems avoids this practice in its implementation systems to the greatest degree possible.
- Your system might be over-extended. The code that is generated for SystemBuild blocks is optimized for performance, but any system can be overloaded by too many tasks doing too much work in a given cycle. Naturally, if this kind of overload occurs, the situation is likely to be catastrophic and reasonably easy to detect. But on a heavily loaded system, minor perturbations such as triggered subsystems or heavily loaded if/else constructs could cause an occasional overflow, which would be hard to debug.

4 Code Generation for Discrete Systems

This chapter introduces features of the generated code for discrete systems. This includes scheduler architecture as it relates to discrete code generation.

4.1 Introduction

A *discrete system* is a model that does not contain any Continuous SuperBlocks. The general categories are single-rate, multi-rate, and procedural discrete systems.

Single-rate discrete system — This system contains SuperBlocks that use the exact same timing attributes.

Multi-rate discrete system — This system contains SuperBlocks with different timing attributes.

Procedural discrete system — This system contains only procedure SuperBlocks and therefore has no timing attributes.

A table describing all the options that control code generation can be found within Appendix A, *AutoCode Options*. For more information about the structure and content of the generated code, see the *AutoCode Reference*.

4.2 How to Generate Code for Discrete Systems

The minimum options required to generate code for a discrete subsystem are: choice of language and top-level SuperBlock or real-time file (`.rtf`). This is shown in Chapter 2, *Using AutoCode*. Additional options are specified along with the required options. A separate options file can be used as a replacement to specify options directly to AutoCode.

4.3 Introduction to Vectorized Code

A vectorized discrete system is a discrete system that uses array variables in the generated code to implement vectors for the purposes of bundling signals together to enable loops within the generated code. The resultant vectorized code is more efficient in terms of code size and performance as compared to the nonvectorized equivalent. There is no need to generate vectorized code unless you are interested in gaining performance improvements and reducing code size on your target hardware. In other words, there is no difference in numerical results between vectorized and non-vectorized code, but *how* those results are obtained is significantly different.

By default, AutoCode generates nonvectorized code. You must specify an option to produce vectorized code (see the `-Ov` entry in Table A-1 on page A-1). That option controls two variations of vectorization which are summarized below.

Maximal Vectorization — This option directs AutoCode to create vectors everywhere possible. Traceability in the generated code is reduced as only one name can be used to represent many signals from the same block.

Label-based Vectorization — This option directs AutoCode to selectively create vectors for only those signals that have a vector name or label as specified in the diagram. This variation lets you specify exactly what signals are to be generated as a vector and exactly what the name of the array is within the generated code. Any signal that does not have a vector name or label will be generated as a scalar variable.

4.4 Introduction to Optimized Code

AutoCode produces generated code that is nearly one-to-one compared to the blocks used in the diagram. However, performance constraints of target hardware require optimization of the generated code. One would expect the target compiler to optimize the code, but many target compilers provide minimal optimization capabilities. Therefore, AutoCode can be directed to perform some optimizations that favor better executable code. Of course, there is a price to be paid; traceability back to the model's diagram is significantly reduced when optimized.

By default, AutoCode does not perform any special optimizations. You must specify which type of optimizations you desire. Some of the optimizations are summarized below.

Variable reuse — Reuse local variables within the code as the outputs of more than one block.

No restart — Generate code that cannot be restarted on the target unless the object code is reloaded.

Variable Block read propagation — Directly reference the Variable Block variable for read operations.

Constant propagation — Blocks that compute a constant are eliminated and the constant value is used directly.

4.5 Introduction to Procedural Code

Procedural code is the generated code for only the Procedure SuperBlocks within a model. The code is typically used for two purposes: 1) to subsequently treat the generated code as a module that is plugged into a much larger code stream; 2) the first step toward linking generated code back into the SystemBuild Simulator to improve its performance as a UserCode Block.

4.6 Sample Generated Code

The following section contains sample generated code. The code was generated with maximal vectorization and the no-restart optimization. Examples have been edited to eliminate the scheduler and other code not relevant for this example.

4.6.1 Sample C Code

EXAMPLE 4-1: SAMPLE_MODEL.c

```
/*************************************************************************
|                   AutoCode/C (TM) Code Generator V6.X                   |
|            INTEGRATED SYSTEMS INC., SUNNYVALE, CALIFORNIA               |
*************************************************************************
rtf filename           : SAMPLE_MODEL.rtf
Filename               : SAMPLE_MODEL.c
Dac filename           : c_sim.dac
Generated on           : Mon Mar 17 18:26:36 1999
Dac file created on    : Thu Mar  6 12:09:32 1999
--
--   Number of External Inputs : 4
--   Number of External Outputs: 8
--
--   Scheduler Frequency:    10.0
--
--   SUBSYSTEM  FREQUENCY  TIME_SKEW  OUTPUT_TIME  TASK_TYPE
--   ---------  ---------  ---------  -----------  ---------
--   1          10.0000    0.00000    0.00000      PERIODIC
*/
```

```
#include <stdio.h>
#include <math.h>
#include "sa_sys.h"
#include "sa_defn.h"
#include "sa_types.h"
#include "sa_math.h"
#include "sa_user.h"
#include "sa_utils.h"
#include "sa_time.h"
#include "sa_fuzzy.h"

/******* System Ext I/O type declarations. *******/
struct _Subsys_1_out {
   RT_FLOAT limited_values_1[4];
   RT_FLOAT limited_values_1_1[4];
};

struct _Sys_ExtIn {
   RT_FLOAT SAMPLE_MODEL_1[4];
};

/******* System Ext I/O type definitions. *******/
struct _Subsys_1_out subsys_1_out = {{-EPSILON, -EPSILON, -EPSILON, -EPSILON},
   {-EPSILON, -EPSILON, -EPSILON, -EPSILON}};
struct _Sys_ExtIn sys_extin;

/******** Procedures' declarations ********/

/******* Procedure: value_added *******/

/***** Inputs type declaration. *****/
struct _value_added_u {
   RT_FLOAT gainfactor_1[4];
};

/***** Outputs type declaration. *****/
struct _value_added_y {
   RT_FLOAT limited_values_1[4];
};

/***** Info type declaration. *****/

struct _value_added_info {
   RT_INTEGER iinfo[5];
   RT_FLOAT RP[16];
};

/******** Procedures' definitions ********/
```

```
/******* Procedure: value_added *******/



void value_added(U, Y, I)
   struct _value_added_u *U;
   struct _value_added_y *Y;
   struct _value_added_info *I;
{
      RT_INTEGER *iinfo = &I->iinfo[0];

      /***** Parameters. *****/
      RT_FLOAT *R_P = &I->RP[0];

      /***** Algorithmic Local Variables. *****/
      RT_INTEGER ilower;
      RT_INTEGER iupper;
      RT_FLOAT uval;
      RT_INTEGER i;
      RT_INTEGER k;
      RT_FLOAT alpha;

      /***** Output Update. *****/
      /* ----------------------------  Linear Interp */
      /* {value_added..2} */
      for (i=1; i<=4; i++) {
         if (U->gainfactor_1[-1+i] < R_P[-2+2*i]) {
            ilower = 1;
            iupper = 0;
         }
         else if (U->gainfactor_1[-1+i] >= R_P[-1+2*i]) {
            ilower = 0;
            iupper = 1;
         }
         else {
            ilower = (RT_INTEGER)((U->gainfactor_1[-1+i] - R_P[-2+2*i])/(R_P[
         -1+2*i] - R_P[-2+2*i]));
            iupper = ilower + 1;
         }
         alpha = (U->gainfactor_1[-1+i] - R_P[-2+ilower+2*i])/(R_P[-2+iupper+2*
         i] - R_P[-2+ilower+2*i]);
         Y->limited_values_1[-1+i] = (1.0 - alpha)*R_P[6+ilower+2*i] + alpha*
         R_P[6+iupper+2*i];
      }

      iinfo[1] = 0;

EXEC_ERROR: return;
}

/******** Tasks declarations ********/
```

```
/******** Tasks code ********/



/******* Subsystem 1  *******/

void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
      static RT_INTEGER iinfo[4] = {0, 1, 1, 1};

      /***** Parameters. *****/
      static RT_FLOAT R_P[8] = {4.3, 5.2, 3.5, 2.3, -4.3, -5.2, -3.5, -2.3};

      /***** Local Block Outputs. *****/

      RT_FLOAT gainfactor_1[4];
      RT_FLOAT inverse_factor_1[4];

      /***** Algorithmic Local Variables. *****/

      RT_INTEGER i;
      static struct _value_added_u value_added_4_u;
      static struct _value_added_y value_added_4_y;
      static struct _value_added_info value_added_4_i = {{0, 1, 1, 1, 1},
         {-10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -1.5, 1.5, -1.5,
         1.5, -1.5, 1.5, -1.5, 1.5}};
      static struct _value_added_u value_added_14_u;
      static struct _value_added_y value_added_14_y;
      static struct _value_added_info value_added_14_i = {{0, 1, 1, 1, 1},
         {-10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -1.5, 1.5, -1.5,
         1.5, -1.5, 1.5, -1.5, 1.5}};

      /***** Output Update. *****/
      /* ---------------------------  Gain Block */
      /* {SAMPLE_MODEL.gf1.1} */
      for (i=1; i<=4; i++) {
         gainfactor_1[-1+i] = R_P[-1+i]*U->SAMPLE_MODEL_1[-1+i];
      }

      /* ---------------------------  Procedure SuperBlock */
      /* {value_added.4} */
      {
         RT_INTEGER k = 0;
         for( k=0;k<4;k++ ) {
            value_added_4_u.gainfactor_1[k] = gainfactor_1[k];
         }
      }
      value_added(&value_added_4_u, &value_added_4_y, &value_added_4_i);
      {
         RT_INTEGER k = 0;
```

```
         for( k=0;k<4;k++ ) {
            Y->limited_values_1[k] = value_added_4_y.limited_values_1[k];
         }
      }
      iinfo[0] = value_added_4_i.iinfo[0];
      if( iinfo[0] != 0 ) {
         value_added_4_i.iinfo[0] = 0; goto EXEC_ERROR;
      }
      /* ----------------------------  Gain Block */
      /* {SAMPLE_MODEL.gf2.2} */
      for (i=1; i<=4; i++) {
         inverse_factor_1[-1+i] = R_P[3+i]*U->SAMPLE_MODEL_1[-1+i];
      }
      /* ----------------------------  Procedure SuperBlock */
      /* {value_added.14} */
      {
         RT_INTEGER k = 0;
         for( k=0;k<4;k++ ) {
            value_added_14_u.gainfactor_1[k] = inverse_factor_1[k];
         }
      }
      value_added(&value_added_14_u, &value_added_14_y, &value_added_14_i);
      {
         RT_INTEGER k = 0;
         for( k=0;k<4;k++ ) {
            Y->limited_values_1_1[k] = value_added_14_y.limited_values_1[k];
         }
      }
      iinfo[0] = value_added_14_i.iinfo[0];
      if( iinfo[0] != 0 ) {
         value_added_14_i.iinfo[0] = 0; goto EXEC_ERROR;
      }

      if(iinfo[1]) {
         SUBSYS_INIT[1] = FALSE;
         iinfo[1] = 0;
      }
      return;

EXEC_ERROR: ERROR_FLAG[1] = iinfo[0];
      iinfo[0]=0;
}
```

4.6.2 Sample Ada Code

EXAMPLE 4-2: SAMPLE_MODEL.a

```
--------------------------------------------------------------------------------
--                  AutoCode/Ada (TM) Code Generator V6.X                     -
--           INTEGRATED SYSTEMS INC., SUNNYVALE, CALIFORNIA                   -
--------------------------------------------------------------------------------
-- rtf filename           : SAMPLE_MODEL.rtf
-- Filename               : SAMPLE_MODEL.a
-- Dac filename           : ada_rt.dac
-- Generated on           : Mon Mar 17 18:27:44 1999
-- Dac file created on    : Mon Mar 10 17:03:32 1999
--
--   Number of External Inputs : 4
--   Number of External Outputs: 8
--
--   Scheduler Frequency:    10.0
--
--   SUBSYSTEM  FREQUENCY  TIME_SKEW  OUTPUT_TIME  TASK_TYPE
--   ---------  ---------  ---------  -----------  ---------
--   1          10.0000    0.00000    0.00000      PERIODIC
--------------------------------------------------------------------------------

----------------------
---  System Data   ---
----------------------
with SYSTEM;
with UNCHECKED_CONVERSION;
with SA_TYPES;                      use SA_TYPES;
with SA_DEFN;                       use SA_DEFN;
with SA_TIME;                       use SA_TIME;
package SYSTEM_DATA is
    NUMIN           : constant RT_INTEGER := 4;
    NUMOUT          : constant RT_INTEGER := 8;
    ExtIn           : RT_FLOAT_AY(0..NUMIN);
    ExtOut          : RT_FLOAT_AY(0..NUMOUT) := (others => -EPSILON);
    SUBSYS_PREINIT  : RT_BOOLEAN_AY(0..NTASKS);

   -------- System Ext I/O type declarations. --------
   type Subsys_1_out_t is record
      limited_values_1 : RT_FLOAT_AY(0..3);
      limited_values_1_1 : RT_FLOAT_AY(0..3);
   end record;

   type Sys_ExtIn_t is record
      SAMPLE_MODEL_1 : RT_FLOAT_AY(0..3);
   end record;
```

```
   -------- System Ext I/O type definitions. --------
   subsys_1_out : Subsys_1_out_t := ((-EPSILON, -EPSILON, -EPSILON, -EPSILON),
      (-EPSILON, -EPSILON, -EPSILON, -EPSILON));
   sys_extin : Sys_ExtIn_t;

end SYSTEM_DATA;

-------- Procedures package declarations --------
with SYSTEM;
with UNCHECKED_CONVERSION;
with SA_TYPES;                          use SA_TYPES;
with SYSTEM_DATA;                       use SYSTEM_DATA;
package value_added_pkg  is

   ------ Inputs type declaration. ------
   type value_added_u_t is record
      gainfactor_1 : RT_FLOAT_AY(0..3);
   end record;

   ------ Outputs type declaration. ------
   type value_added_y_t is record
      limited_values_1 : RT_FLOAT_AY(0..3);
   end record;

   ------ Info type declaration. ------
   type value_added_info_t is record
      iinfo : RT_INTEGER_AY(0..4);
      RP : RT_FLOAT_AY(0..15);
   end record;

   type value_added_u_t_P is access value_added_u_t;
   function ptr_of is new UNCHECKED_CONVERSION
            (SOURCE => SYSTEM.ADDRESS, TARGET => value_added_u_t_P);
   type value_added_y_t_P is access value_added_y_t;
   function ptr_of is new UNCHECKED_CONVERSION
            (SOURCE => SYSTEM.ADDRESS, TARGET => value_added_y_t_P);
   type value_added_info_t_P is access value_added_info_t;
   function ptr_of is new UNCHECKED_CONVERSION
            (SOURCE => SYSTEM.ADDRESS, TARGET => value_added_info_t_P);

   -------- Procedure: value_added --------
   procedure value_added(U : value_added_u_t_P;
      Y : value_added_y_t_P;
      I : value_added_info_t_P
   );
end value_added_pkg;

-------- Subsystems' declarations --------
with SYSTEM;
with UNCHECKED_CONVERSION;
with SA_TYPES;                          use SA_TYPES;
with SYSTEM_DATA;                       use SYSTEM_DATA;
```

```
with value_added_pkg;                  use value_added_pkg;
package SUBSYSTEMS is

   -------- Subsystem 1 Package --------
   package subsys_1_pkg is
         type Sys_ExtIn_t_P is access Sys_ExtIn_t;
      function ptr_of is new UNCHECKED_CONVERSION
               (SOURCE => SYSTEM.ADDRESS, TARGET => Sys_ExtIn_t_P);
      type Subsys_1_out_t_P is access Subsys_1_out_t;
      function ptr_of is new UNCHECKED_CONVERSION
               (SOURCE => SYSTEM.ADDRESS, TARGET => Subsys_1_out_t_P);
      U : Sys_ExtIn_t_P := ptr_of(sys_extin'address);
      Y : Subsys_1_out_t_P := ptr_of(subsys_1_out'address);

      procedure subsys_1;
   end subsys_1_pkg;
end SUBSYSTEMS;

with SA_DEFN;                          use SA_DEFN;
with SA_TYPES;                         use SA_TYPES;
with SYSTEM_DATA;                      use SYSTEM_DATA;
package body SUBSYSTEMS is
    package body subsys_1_pkg is separate;
end SUBSYSTEMS;

with SA_TYPES;      use SA_TYPES;
with SYSTEM_DATA;   use SYSTEM_DATA;
with SA_UTILITIES;  use SA_UTILITIES;
separate (SUBSYSTEMS)
package body subsys_1_pkg is

   SUBSYS_ID : constant := 1;

   -------- Tasks code --------
   iinfo : RT_INTEGER_AY(0..3) := (0, 1, 1, 1);

   ------ Parameters. ------
   R_P : RT_FLOAT_AY(0..7) := (4.3, 5.2, 3.5, 2.3, -4.3, -5.2, -3.5, -2.3);
   value_added_4_u : value_added_u_t;
   value_added_4_y : value_added_y_t;
   value_added_4_i : value_added_info_t := ((0, 1, 1, 1, 1), (-10.5, 20.5,
      -10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -1.5, 1.5, -1.5, 1.5, -1.5, 1.5,
      -1.5, 1.5));
   value_added_14_u : value_added_u_t;
   value_added_14_y : value_added_y_t;
   value_added_14_i : value_added_info_t := ((0, 1, 1, 1, 1), (-10.5, 20.5,
      -10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -1.5, 1.5, -1.5, 1.5, -1.5, 1.5,
      -1.5, 1.5));

   procedure subsys_1 is

      ------ Local Block Outputs. ------
```

```
      gainfactor_1 : RT_FLOAT_AY(0..3);
      inverse_factor_1 : RT_FLOAT_AY(0..3);

      ------ Algorithmic Local Variables. ------
      i_2 : RT_INTEGER;

   begin
      ------ Output Update. ------
      -- ----------------------------  Gain Block --
      -- {SAMPLE_MODEL.gf1.1} --
      for i_2 in RT_INTEGER range 1..4 loop
         gainfactor_1(-1+i_2) := R_P(-1+i_2)*U.SAMPLE_MODEL_1(-1+i_2);
      end loop;
      -- ----------------------------  Procedure Super Block --
      -- {value_added.4} --
      value_added_4_u.gainfactor_1(0..3) := gainfactor_1(0..3);
      value_added(ptr_of(value_added_4_u'address), ptr_of(
         value_added_4_y'address), ptr_of(value_added_4_i'address));
      Y.limited_values_1(0..3) := value_added_4_y.limited_values_1(0..3);
      iinfo(0) := value_added_4_i.iinfo(0);
      if iinfo(0) /= 0 then
         value_added_4_i.iinfo(0) := 0; raise EXEC_ERROR;
      end if;
      -- ----------------------------  Gain Block --
      -- {SAMPLE_MODEL.gf2.2} --
      for i_2 in RT_INTEGER range 1..4 loop
         inverse_factor_1(-1+i_2) := R_P(3+i_2)*U.SAMPLE_MODEL_1(-1+i_2);
      end loop;
      -- ----------------------------  Procedure Super Block --
      -- {value_added.14} --
      value_added_14_u.gainfactor_1(0..3) := inverse_factor_1(0..3);
      value_added(ptr_of(value_added_14_u'address), ptr_of(
         value_added_14_y'address), ptr_of(value_added_14_i'address));
      Y.limited_values_1_1(0..3) := value_added_14_y.limited_values_1(0..3);
      iinfo(0) := value_added_14_i.iinfo(0);
      if iinfo(0) /= 0 then
         value_added_14_i.iinfo(0) := 0; raise EXEC_ERROR;
      end if;

   if iinfo(1) > 0 then
      iinfo(1) := 0;
      SUBSYS_INIT(1) := false;
   end if;

   exception
      when EXEC_ERROR =>
          ERROR_FLAG(1) := iinfo(0); iinfo(0) := 0;
      when NUMERIC_ERROR | CONSTRAINT_ERROR =>
          ERROR_FLAG(1) := MATH_ERROR;
      when OTHERS =>
          ERROR_FLAG(1) := UNKNOWN_ERROR;
   end subsys_1;
end subsys_1_pkg;
```

```
-------- Procedures package bodies --------
with SA_TYPES;                          use SA_TYPES;
with SA_DEFN;                           use SA_DEFN;
with SYSTEM_DATA;                       use SYSTEM_DATA;
package body value_added_pkg is
-------- Procedure: value_added --------
   procedure value_added(U : value_added_u_t_P;
      Y : value_added_y_t_P;
      I : value_added_info_t_P
   ) is
      iinfo : RT_INTEGER_AY_5_P := ptr_of(I.iinfo'address);

      ------ Parameters. ------
      R_P : RT_FLOAT_AY_16_P := ptr_of(I.RP'address);

      ------ Algorithmic Local Variables. ------
      ilower : RT_INTEGER;
      iupper : RT_INTEGER;
      uval : RT_FLOAT;
      i_1 : RT_INTEGER;
      k_1 : RT_INTEGER;
      alpha_1 : RT_FLOAT;

   begin
      ------ Output Update. ------
      -- ----------------------------  Linear Interp --
      -- {value_added..2} --
      for i_1 in RT_INTEGER range 1..4 loop
         if U.gainfactor_1(-1+i_1) < R_P(-2+2*i_1) then
            ilower := 1;
            iupper := 0;
         elsif U.gainfactor_1(-1+i_1) >= R_P(-1+2*i_1) then
            ilower := 0;
            iupper := 1;
         else
            ilower := ITRUNCATE((U.gainfactor_1(-1+i_1) - R_P(-2+2*i_1))/(R_P(
         -1+2*i_1) - R_P(-2+2*i_1)));
            iupper := ilower + 1;
         end if;
         alpha_1 := (U.gainfactor_1(-1+i_1) - R_P(-2+ilower+2*i_1))/(R_P(
         -2+iupper+2*i_1) - R_P(-2+ilower+2*i_1));
         Y.limited_values_1(-1+i_1) := (1.0 - alpha_1)*R_P(6+ilower+2*i_1) +
         alpha_1*R_P(6+iupper+2*i_1);
      end loop;

   iinfo(1) := 0;

   exception
      when EXEC_ERROR =>
          null;
      when NUMERIC_ERROR | CONSTRAINT_ERROR =>
          iinfo(0) := MATH_ERROR;
```

```
      when OTHERS =>
          iinfo(0) := UNKNOWN_ERROR;
   end value_added;
end value_added_pkg;
```

5 Code Generation for Continuous Systems

This chapter discusses the scheduler architecture as it relates to continuous code generation. Topics include fixed-step integrators, user-defined integrators, and how to generate code for continuous and hybrid systems.

5.1 Introduction

AutoCode supports code generation for continuous or hybrid (continuous and discrete) systems. The AutoCode scheduler supports continuous subsystems in the same manner in which it supports discrete subsystems.

For continuous subsystems, at each minor cycle, the scheduler:

- Schedules the continuous subsystem to run.
- Posts continuous subsystem's outputs.
- Performs sample and hold on the continuous subsystem's inputs.
- Dispatches the continuous subsystem if ready.
- Handles vectorization and optimization the same as for discrete systems.

An element of the scheduler is the Integrator (see Figure 5-1 on page 5-2). It performs continuous, fixed-step integration of states and implicitly dispatches the continuous subsystem to perform the state and output updates. The integrator/continuous task pair is, by default, treated as the fastest task to be dispatched by the scheduler.

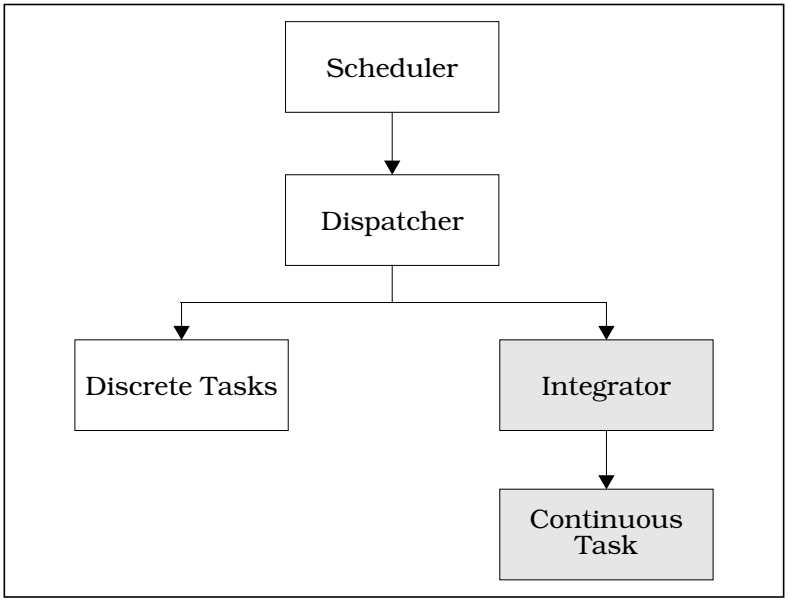


FIGURE 5-1 Scheduler Architecture

5.2 Integrators

AutoCode supplies four fixed-step integrators:

- First order Runge-Kutta (Euler)
- Second order Runge-Kutta (Modified Euler)
- Fourth order Runge-Kutta (Simpson's 2nd rule)
- Kutta-Merson

All of these integrators are located in the templates directory in the integrator template file *language*`_intgr.tpl`. There is also the capability to insert a user-supplied integrator. Instructions for using your own integrator are provided in Section 5.4.2 on page 5-4.

5.3 Limitations

When using continuous code generation, keep these limitations in mind:

- Only fixed-step integrators are supported.
- There is a slight mismatch of sim and continuous application outputs (i.e., the subsystem external inputs at `time t` and at `time t+h`, where `h` is the integration step, are assumed to be unchanged inside AutoCode integrator algorithms).
- Continuous task states and derivatives are always of float data type.
- Algebraic loops are not supported.
- Only sim initialization mode 0 (initmode 0; see sim help for details) is supported.
- You cannot generate procedures-only continuous code (procedure around a top-level continuous hierarchy).

5.4 How to Generate Code for Continuous or Hybrid Systems

As described in Section 2.1 on page 2-1, using AutoCode, you can generate C high-level language code from SystemBuild, the Xmath Commands window, or from the operating system command line. The subsections that follow discuss each of these methods of code generation in terms of those options that are unique to generating code for continuous or hybrid systems.

You need both `c_sim.tpl` and `c_intgr.tpl` template files for C or `ada_rt.tpl` and `ada_intgr.tpl` for Ada (supplied in the `templates` directory). The `c_sim.tpl` and `ada_rt.tpl` template files include continuous subsystems-related parameters and the integrator template file. The integrator template file contains the code for the four integrators and a stubbed routine, `usrintegrator`, which provides the means for user-defined integrator implementation.

5.4.1 Generating Code for Continuous Systems from SystemBuild

To use AutoCode while inside SystemBuild, select Tools→AutoCode on the Catalog Browser to open the dialog. Instructions for using this dialog are in the MATRIX$_X$ Help.

Depending on the template file used, the code generated can be either C code or Ada code.

5.4.2 Xmath Command-Line Options for Continuous Code Generation

The method for generating code for a continuous or hybrid system using the Xmath command line follows the procedure described in Section 2.1.2 on page 2-2. Two command line options that are unique to continuous code generation are `ialg` and `csi`.† Although not for exclusive use in continuous code generation, the `minsf` option is useful for increasing the rate of a continuous task. See Table 5-1 for a summary of these options.

TABLE 5-1 Xmath Command Options for Continuous Code Generation

Option	Description
ialg	Specifies the integrator selection 0 = user-defined integrator 1 = first order Runge-Kutta integrator 2 = second order Runge-Kutta integrator (default) 3 = fourth order Runge-Kutta integrator 4 = Kutta-Merson integrator
csi	Specifies the continuous task sample interval
minsf	Specifies the minimum AutoCode scheduler frequency in seconds (0.0 is the default)

As indicated in Table 5-1, `ialg` specifies the selected integrator. The option takes an integer argument of 0, 1, 2, 3, or 4. The default integrator is the second order Runge-Kutta.

When using 0 (user-defined integrator) for this command-line option, the integrator function should be implemented inside the function `usrintegrator` located in `c_intgr.tpl` for C or `ada_intgr.tpl` for Ada.

Because the integrator is invoked at each scheduler interval and the continuous task is dispatched via the integrator, an implicit frequency (that of the scheduler) is associated with the continuous task. If the system is all continuous, the scheduler cycle is 1 Hz. For hybrid systems, the implicit frequency of the continuous task is always the least common multiple of all of the frequencies of the discrete tasks. For continuous only modes, the implicit frequency of the single continuous defaults to

† For standalone AutoCode, results for generated code will not match sim unless the `csi` option is not zero. Typically, set `csi` to 0.01, the time vector for standalone sim. Then, results will match.

1 Hz. The command-line option `csi` specifies the sample interval for a continuous task. This option is useful for adjusting the rate of the continuous task.

The command-line option `minsf` specifies the minimum AutoCode scheduler frequency. This option is useful for increasing the rate of a continuous task. The real-time scheduler frequency is set to the larger value of the frequency determined by the block diagram application and the value specified by the `minsf` option. The default value for this option is `0.0`, which allows the application to set its own scheduler frequency. Deviation from this default should be approached with caution, as a consistent scheduler frequency should normally be based on a least common multiple of the application timing requirements.

For example, to generate code for a model with a continuous subsystem, using the fourth order Runge-Kutta integrator method, and minimum scheduler frequency of 300.0 Hz in file `model.c`, use the following Xmath command:

```
autocode, model="model", {ialg=3, minsf=300.0}
```

In this case, the `autocode` command automatically generates the file `model.c` in the directory from which Xmath was invoked.

5.4.3 OS Command-Line Options for Continuous Code Generation

The method for generating code for a continuous or hybrid system using the operating system command line follows the procedure described in Section 2.1.3 on page 2-3. Two command-line options that are unique to continuous code generation are `-i` and `-csi`. Although not for exclusive use in continuous code generation, the `-minsf` option may be useful for increasing the rate of a continuous task. See Table 5-2 for a summary of these options.

TABLE 5-2 Operating System Command Options for Continuous Code Generation

Option	Description
-i	Specifies the integrator selection 0 = user-defined integrator 1 = first order Runge-Kutta integrator 2 = second order Runge-Kutta integrator (default) 3 = fourth order Runge-Kutta integrator 4 = Kutta-Merson integrator

TABLE 5-2 Operating System Command Options for Continuous Code Generation

Option	Description
-csi	Specifies the continuous task sample interval
-minsf	Specifies the minimum AutoCode scheduler frequency in seconds (0.0 is the default)

As indicated in Table 5-2, `-i` specifies the selected integrator. The option takes an integer argument of 0, 1, 2, 3, or 4. The default integrator is the second order Runge-Kutta.

When using 0 (user-defined integrator) for this command-line option, the integrator function should be implemented inside the function `usrintegrator()` located in `c_intgr.tpl` for C or `ada_intgr.tpl` for Ada.

Because the integrator is invoked at each scheduler interval and the continuous task is dispatched via the integrator, an implicit frequency (that of the scheduler) is associated with the continuous task. For hybrid systems, the implicit frequency of the continuous task is always the least common multiple of all the frequencies of the discrete tasks. The command-line option `-csi` specifies the sample interval for a continuous task.

The command-line option `-minsf` specifies the minimum AutoCode scheduler frequency. This option is useful for increasing the rate of a continuous task. The real-time scheduler frequency is set to the larger value of the frequency determined by the block diagram application and the value specified by the `minsf` option. The default value for this option is `0.0`, which allows the application to set its own scheduler frequency. Deviation from this default should be approached with caution, as a consistent scheduler frequency should normally be based on a least common multiple of the application timing requirements.

To generate code for a model with a continuous subsystem, using the fourth order Runge-Kutta integrator method, and minimum scheduler frequency of 300.0 Hz, use the operating system command shown in Example 5-1 (C) or Example 5-2 (Ada):

EXAMPLE 5-1: Sample Operating System Command for C

```
% autostar -l c -i 3 -minsf 300.0 -o model.c model.rtf
```

EXAMPLE 5-2: Sample operating system command for Ada

```
% autostar -l a -i 3 -minsf 300.0 -o model.a model.rtf
```

5.5 Sample Generated C Code

The following example is a file that lists the generated model and default integrator (Runge-Kutta 2) code for the block diagram model located in the file `mws_demo.dat` in the `classical_demo` directory located in the SystemBuild `demo` distribution directory. The block diagram is shown in Figure 5-2.

The sample generated code (Example 5-3) has been edited for brevity, showing only the most important features.

As code can change slightly from one release to the next, please refer to the current example in your demo directory for an exact code listing.

NOTE: If you need to review the steps required to create an executable, refer to Section 2.4.1 on page 2-9.

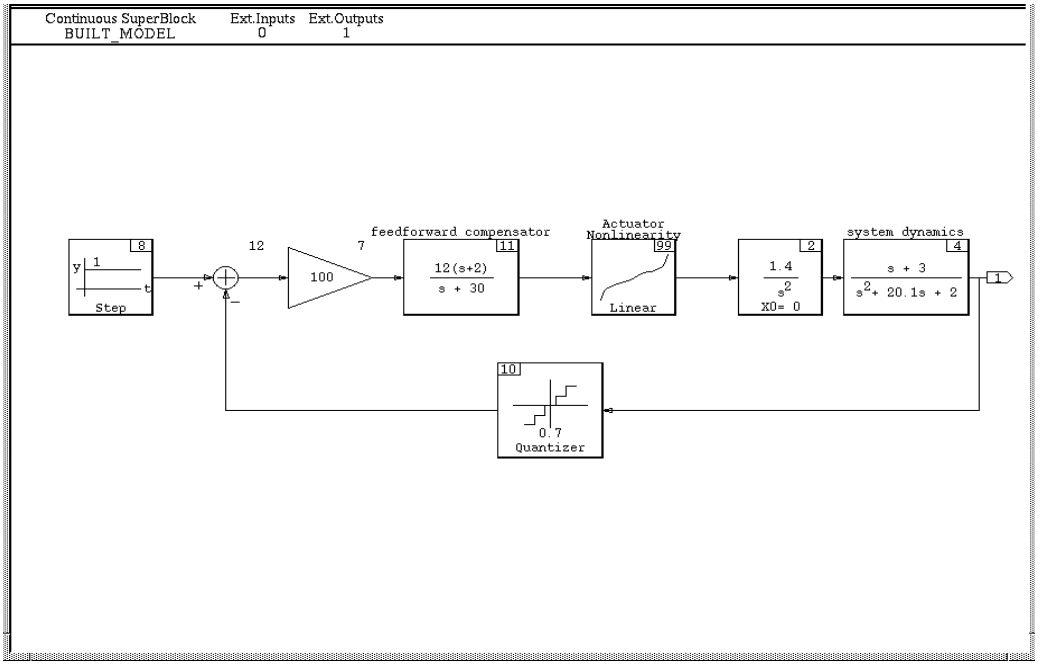


FIGURE 5-2 Built_Model SuperBlock

EXAMPLE 5-3: File `built_model.c`

```
/*************************************************************
| AutoGen/C (TM) Code Generator V6.X |
| INTEGRATED SYSTEMS INC., SUNNYVALE, CALIFORNIA |
*************************************************************
Modelname : built_model
Filename : built_model.c
Generated on : Wed Aug 4 16:43:28 1999
Dac file created on : Tue Jul 27 20:48:17 1999
*/

#include <stdio.h>
#include "sa_intgr.h"
...
/*** System Data ***/

#define SCHEDULER_FREQ 300.0
#define NTASKS 1
```

```
#define NUMIN 0
#define NUMOUT 1
#define IALG 2

enum TASK_STATE_TYPE { IDLE, RUNNING, BLOCKED, UNALLOCATED };

enum SUBSYSTEM_TYPE { CONTINUOUS, PERIODIC, ENABLED_PERIODIC,
                  TRIGGERED_ANT,TRIGGERED_ATR, TRIGGERED_SAF, NONE };
...
/******* Global declarations. *******/
...
/******* System Ext I/O structs. declarations.*******/
...
/******* System Ext I/O structs. definitions.*******/
...
/**Continuous Subsystem states and info structs. declarations.**/
struct _Subsys_1_states {
 RT_FLOAT system_dynamics_S1;
 RT_FLOAT system_dynamics_S2;
 RT_FLOAT BUILT_MODEL_2_S1;
 RT_FLOAT BUILT_MODEL_2_S2;
 RT_FLOAT feedforward_compensator_S1;
};
struct _Subsys_1_info {
 RT_INTEGER iinfo[5];
 RT_FLOAT rinfo[5];
};

/***Continuous Subsystem states and info structs. definitions.***/
struct _Subsys_1_states subsys_1_states[2] = {0., 0., 0., 0.,  0.,0.,
0., 0.,
   0., 0.};

struct _Subsys_1_info subsys_1_info = {0, 1, 1, 1, 0, 1., 0.};
...
/******* Task's declarations. *******/

/******* (Continuous) Subsystem 1 *******/
extern void subsys_1();

/******* Task's code. *******/

/******* (Continuous) Subsystem 1 *******/
void subsys_1(Y, S, I)
 struct _Subsys_1_out *Y;
 struct _Subsys_1_states *S;
 struct _Subsys_1_info *I;
{
 RT_INTEGER *iinfo = &I->iinfo[0];
 RT_FLOAT *rinfo = &I->rinfo[0];
 RT_INTEGER INIT = iinfo[1];
 RT_INTEGER STATES = iinfo[2];
 RT_INTEGER OUTPUTS = iinfo[3];
```

```
 RT_INTEGER CALLER = iinfo[4];
 const RT_DURATION TIME = rinfo[0];

 /***** Current and Next States Pointers. *****/

 struct _Subsys_1_states *X = &S[0];
 struct _Subsys_1_states *XD = &S[1];

 /***** Parameters. *****/
...

/***** Local Block Outputs. *****/
...

if(OUTPUTS) { /* Output Update. */
 /*---------------------------- Num - Den Coeffs. */
 /* {BUILT_MODEL.system dynamics.4} */
 Y->system_output = 0.5*X->system_dynamics_S1;
 Y->system_output = Y->system_output + 1.5*X->system_dynamics_S2;
 /*---------------------------- Nth Order Integrator */
 /* {BUILT_MODEL..2} */
 BUILT_MODEL_2_1 = 1.4*X->BUILT_MODEL_2_S1;
 /*---------------------------- Step Function */
 BUILT_MODEL_8_1= 1;
 /*---------------------------- Quantization */
 /* {BUILT_MODEL..10} */
 BUILT_MODEL_10_1 = 0.7*ROUND(Y->system_output/0.7);
 sgn = SGN(Y->system_output);
 ushift = fabs(Y->system_output) + 0.35;
 remain = fmod(ushift,0.7);
 alpha = remain - (1.0 - RELTOL)*0.7;
 if (alpha > 0.0) {
    BUILT_MODEL_10_1 = BUILT_MODEL_10_1 + sgn*alpha/RELTOL;
}
 ...
}
if(STATES) { /* State Update. */
 /*---------------------------- Num - Den Coeffs. */
 /* {BUILT_MODEL.system dynamics.4} */
 XD->system_dynamics_S1 = 0.0;
 XD->system_dynamics_S2 = 0.0;
 XD->system_dynamics_S1 = XD->system_dynamics_S1 - 20.1
 *X->system_dynamics_S1;
 XD->system_dynamics_S1 = XD->system_dynamics_S1 - 2.0
 *X->system_dynamics_S2;
 XD->system_dynamics_S2 = XD->system_dynamics_S2 +
    X->system_dynamics_S1;
 XD->system_dynamics_S1 = XD->system_dynamics_S1 +
    2.0*BUILT_MODEL_2_1;
 /*---------------------------- Nth Order Integrator */
 /* {BUILT_MODEL..2} */
 XD->BUILT_MODEL_2_S1 = X->BUILT_MODEL_2_S2;
 XD->BUILT_MODEL_2_S2 = Actuator_Nonlinearity_1;
...
```

```
}
 INIT = 0;
 iinfo[1] = 0;
 return;
ERROR: ERROR_FLAG[1] = iinfo[0];
 iinfo[0]=0;
}
/* The function rungekutta2 employs the second-order Runge-Kutta
method with Kutta's coefficients to integrate a system of n
simultaneous first order ordinary differential equations dxdt[j] =
dx[j]/dt, (j=1,2,...,n), across one step of length h in the
independent variable t, subject to initial conditions x[j],
(j=1,2,...,n). Each dxdt[j], the derivative of x[j], must be computed
two times per integration step by calling the state derivatives and
output equations function (sybsys_1()).  savex(j) is used to save
the initial value of x(j) and phi(j) is the increment function for
the j(th) equation. As written, n may be no larger than 5. (Modified
Euler)
*/
void rungekutta2(n,x,dxdt,t,h)
    RT_INTEGER n;
    RT_FLOAT *x,*dxdt,t,h;
{
    RT_FLOAT phi[5];
    RT_FLOAT savex[5];
    RT_INTEGER j, retval;
    RT_FLOAT hh = t;

    ss1_rinfo[0] = hh; ss1_rinfo[1] = 0.0;
    ss1_iinfo[2]=1; ss1_iinfo[3]=1; ss1_iinfo[4]=8;
    subsys_1(&subsys_1_out, subsys_1_states, &subsys_1_info);

    for (j=0; j<n; j++) {
         savex[j] = x[j];
         phi[j] = dxdt[j];
         x[j] = savex[j] + h*dxdt[j];
    }
    hh=t+h;

    ss1_rinfo[0] =  hh; ss1_rinfo[1] = 0.0;
    ss1_iinfo[2]=1; ss1_iinfo[3]=1; ss1_iinfo[4]=0;
    subsys_1(&subsys_1_out, subsys_1_states, &subsys_1_info);

    for (j=0; j<n; j++) x[j] = savex[j] + (phi[j] + dxdt[j])*h/2.0;
}

/*---------------*
 *-- SCHEDULER --*
 *---------------*/
...
void Init_Scheduler()
{
 ...
```

```
}
void SCHEDULER()
{

    register RT_INTEGER NTSK;
    register RT_INTEGER J;
    RT_INTEGER ITSK;
    RT_INTEGER I;

    TIME_COUNT = TIME_COUNT + 1;

    /*** System Input ***/
    ...
    /*** Task Scheduling ***/

    for( NTSK=NTASKS; NTSK>=1; NTSK-- ){

    switch( TASK_STATE[NTSK] ){
     case IDLE :

          switch( TCB[NTSK].TASK_TYPE ){
             case CONTINUOUS :
             case PERIODIC :
                 if( TCB[NTSK].START == 0 ){
                     Queue_Task(NTSK);
                     Update_Outputs(NTSK);
                 TCB[NTSK].START = TCB[NTSK].SCHEDULING_COUNT;
                 }else{
                     TCB[NTSK].START = TCB[NTSK].START - 1;
                 }
                 break;

          case ENABLED_PERIODIC :
          ...
                 }
                 break;

      case RUNNING :
...
}
 }

 /*** System Output ***/
...
/*** Update elapsed time ***/

 ELAPSED_TIME = ((RT_DURATION)TIME_COUNT)*SCHEDULER_INTERVAL;

 /*** Task Input Sample and Hold ***/
...
```

```
/*** Signal End of Critical Section ***/
...

/*** Task Dispatching ***/

 while( ITSK < CURRENT_PRIORITY && ITSK <= DISPATCH_COUNT ){
   Disable;
   if( DISPATCH[ITSK] ){
       LEVEL++;
       PRIORITY[LEVEL] = CURRENT_PRIORITY;
       CURRENT_PRIORITY = ITSK;
       DISPATCH[ITSK] = FALSE;
       Enable;
       switch (ITSK){
         case 1:
         subsys_1 (&subsys_1_out, &subsys_1_states,
            &subsys_1_info);
         rungekutta2(5,
         (RT_FLOAT *)(&subsys_1_states[0]),
         (RT_FLOAT *)(&subsys_1_states[1]),
         (RT_FLOAT)SUBSYS_TIME[1],
         0.003);

   break;

   default : break;
 }
 ...
}
 ..
}
```

5.6 Sample Generated Ada Code

The following example is a file that lists the generated model and default integrator (Runge-Kutta 2) code for the block diagram model located in the file `mws_demo.dat` in the `classical_demo` directory located in the SystemBuild `demo` distribution directory. The block diagram is shown in Figure 5-2.

AutoCode automatically generates `built_model.a` in the directory from which Xmath was invoked. The sample generated code (Example 5-4) has been edited for brevity, showing only the most important features.

As code can change slightly from one release to the next, be sure to refer to the current example in your demo directory for an exact code listing.

NOTE: If you need to review the steps required to create an executable, refer to Section 2.4.1 on page 2-9.

EXAMPLE 5-4: File `built_model.a`

```
------------------------------------------------------------------------
--                   AutoCode/Ada (TM) Code Generator V6.X
--            INTEGRATED SYSTEMS INC., SUNNYVALE, CALIFORNIA
------------------------------------------------------------------------
-- Modelname              : built_model
-- Filename               : built_model.ada
-- Dac filename           : ada_rt.dac
-- Generated on           : Wed Dec  1 20:59:39 1998
-- Dac file created on    : Wed Dec  1 18:01:31 1998
------------------------------------------------------------------------
...
package SUBSYSTEMS is

   -------- (Continuous) Subsystem 1 Package --------
   package subsys_1_pkg is
      ...

      procedure subsys_1;

      procedure rungekutta2(n :in RT_INTEGER;
                            x :in out RT_FLOAT_AY_5_P;
                            dxdt :RT_FLOAT_AY_5_P;
                            t :in RT_FLOAT;
                            h :in RT_FLOAT);
   end subsys_1_pkg;

end SUBSYSTEMS;
```

```
...
package body Subsys_1_pkg is

   SUBSYS_ID : constant := 1;

   -------- Task's code. --------
   ...
   procedure subsys_1 is
      ...
      ------ Local Block Outputs. ------
      ...
      ------ Algorithmic Local Variables. ------
      ...
   begin
      if iinfo(1) > 0 then
         INIT := TRUE; iinfo(1) := 0;
      end if;
      if iinfo(2) > 0 then
         STATES := TRUE; iinfo(2) := 0;
      end if;
      if iinfo(3) > 0 then
         OUTPUTS := TRUE; iinfo(3) := 0;
      end if;

      ------ Output Update. ------

      if OUTPUTS then
         -- ----------------------------  Num - Den Coeffs. --
         -- {BUILT_MODEL.system dynamics.4} --
         Y.system_output := 0.5*X.system_dynamics_S1 + 1.5*
             X.system_dynamics_S2;

         -- ----------------------------  Nth Order Integrator --
         -- {BUILT_MODEL..2} --
         BUILT_MODEL_2_1 := 1.4*X.BUILT_MODEL_2_S1;
         -- ----------------------------  Step Function --
         -- {BUILT_MODEL..8} --
            BUILT_MODEL_2_1=1;
-- ----------------------------  Quantization --
         -- {BUILT_MODEL..10} --
         BUILT_MODEL_10_1 := 0.7*ROUND(Y.system_output/0.7);
         sgn := SGN(Y.system_output);
         ushift := ABS(Y.system_output) + 0.35;
         remain := ((ushift)MOD(0.7));
         alpha_1 := remain - (1.0 - RELTOL)*0.7;
         if alpha_1 > 0.0 then
           BUILT_MODEL_10_1 := BUILT_MODEL_10_1 + sgn*alpha_1/RELTOL;
         end if;
         ...
```

```
   end if;

   ------ State Update. ------
   if STATES then
      -- ----------------------------  Num - Den Coeffs. --
      -- {BUILT_MODEL.system dynamics.4} --
      XD.system_dynamics_S1 := 0.0;
      XD.system_dynamics_S2 := 0.0;
      XD.system_dynamics_S1 := XD.system_dynamics_S1 -  20.1*
         X.system_dynamics_S1;
      XD.system_dynamics_S1 := XD.system_dynamics_S1 -  2.0*
         X.system_dynamics_S2;
      XD.system_dynamics_S2 := XD.system_dynamics_S2 +
         X.system_dynamics_S1;
      XD.system_dynamics_S1 := XD.system_dynamics_S1 + 2.0*
         BUILT_MODEL_2_1;
      -- ----------------------------  Nth Order Integrator --
      -- {BUILT_MODEL..2} --
      XD.BUILT_MODEL_2_S1 := X.BUILT_MODEL_2_S2;
      XD.BUILT_MODEL_2_S2 := Actuator_Nonlinearity_1;
      ...
   end if;

   INIT := FALSE;
exception
   when EXEC_ERROR =>
       ERROR_FLAG(1) := iinfo(0); iinfo(0) := 0;
   when NUMERIC_ERROR | CONSTRAINT_ERROR =>
       ERROR_FLAG(1) := MATH_ERROR;
   when OTHERS =>
       ERROR_FLAG(1) := UNKNOWN_ERROR;
end subsys_1;

-- The function rungekutta2 employs the second-order Runge-Kutta method
-- with Kutta's coefficients to integrate a system of n simultaneous
-- first order ordinary differential equations dxdt(j) = dx(j)/dt,
-- (j=1,2,...,n), across one step of length h in the independent
-- variable t, subject to initial conditions x(j), (j:=1,2,...,n). Each
-- dxdt(j), the derivative of x(j), must be computed two times per
-- integration step by calling the state derivatives and output
-- equations function (subsys_1()). savex(j) is used to save the
-- initial value of x(j) and phi(j) is the increment function for the
-- j(th) equation. As written, n may be no larger than 5.
-- (Modified Euler)
procedure rungekutta2(n    :in RT_INTEGER;
                      x    :in out RT_FLOAT_AY_5_P;
                      dxdt :in RT_FLOAT_AY_5_P;
                      t    :in RT_FLOAT;
                      h    :in RT_FLOAT) is
```

```
        phi    : RT_FLOAT_AY(0..5);
        savex  : RT_FLOAT_AY(0..5);
        j      : RT_INTEGER;
        retval : RT_INTEGER;
        hh     : RT_FLOAT := t;

    begin

        I.rinfo(0) := hh;   -- TIME
        I.rinfo(1) := h;    -- SAMPLE INTERVAL
        I.rinfo(2) := 0.0;  -- SKEW
        I.rinfo(3) := 0.0;  -- START TIME
        I.iinfo(2):=1; I.iinfo(3):=1; I.iinfo(4):=1;
        subsys_1;

        for j in 0..n loop
             savex(j) := x(j);
             phi(j) := dxdt(j);
             x(j) := savex(j) + h*dxdt(j);
        end loop;
        hh:=t+h;



        I.rinfo(0) := hh;   -- TIME
        I.rinfo(1) := h;    -- SAMPLE INTERVAL
        I.rinfo(2) := 0.0;  -- SKEW
        I.rinfo(3) := 0.0;  -- START TIME
        I.iinfo(2):=1; I.iinfo(3):=1; I.iinfo(4):=1;
        subsys_1;

        for j in 0..n -l loop
           x(j) := savex(j) + (phi(j) + dxdt(j))*h/2.0;
        end loop;
    end rungekutta2;
    ...

end Subsys_1_pkg;
```

5.7 Hints

When dealing with a system containing a single continuous subsystem, AutoCode generates a SCHEDULER_FREQ of 1.0 (that is, the inherent rate of the continuous subsystem is that of the scheduler, 1.0). Additionally, when dealing with a hybrid system, AutoCode, by default, treats the continuous subsystem as the fastest task to be dispatched (again, the inherent rate of the continuous subsystem is that of the scheduler). This value might not reflect the true dynamics of the system. In order to obtain an approximate rate for the continuous task, you need to use sim iteratively (or lin for predominantly linear systems) to arrive at an optimal step size for the integration algorithm, and thus, an approximate sampling interval for the continuous task. For a continuous system (represented by differential equations), the step size is related to its eigenvalues (the eigenvalues vary in time for nonlinear systems). Therefore, AutoCode cannot calculate the average step size.

Typically, a continuous system needs to be sampled 5 to 15 times faster than the smallest time constant in the system (depending on the order of the integration algorithm). This time constant is the reciprocal of the largest eigenvalue in the system and this information can be obtained with lin.

6 Customizing AutoCode and Generated Code

This chapter provides advanced methods for customizing AutoCode and its output real-time code using AutoCode configuration options, templates, BlockScript, and %variables.

6.1 Introduction

You can customize the AutoCode process and the generated output code to suit your specific needs. The different ways you can do this are listed below and described in detail in the sections that follow:

- AutoCode configuration options allow you to specify indentation, coding of significant digits for numeric literals, minimum scheduler frequency, and output file name as described in *AutoCode Configuration Options* on page 6-2.
- Templates allow you to modify the overall architecture of generated code, customize the scheduler, modify data structures and external I/O calls, add user codes described in *Templates* on page 6-2.
- BlockScript enables you to create your custom block algorithm and generate it in-line in the output source file s described in *BlockScript Block* on page 6-2.
- Data parameterization (%variable) allows the numeric literals in the block algorithms to be represented by named variables (%variables) as described in *Data Parameterization* on page 6-3.
- Using existing code libraries and interfaces to hardware are accomplished by using a UserCode Block as described on page 6-4 or Macro Procedure as described on page 6-4.

6.2 AutoCode Configuration Options

You can specify the AutoCode configuration from a SystemBuild form (see Chapter 2), by using Xmath Commands window options or by using operating system command-line options. For information on Xmath Commands window or operating system command-line options, see Appendix A, *AutoCode Options*.

6.3 Templates

Templates serve as the front end to AutoCode. They determine completely what the output code should be for a given model (`.rtf` file) and command-line options. You use the Template Programming Language (tpl) to specify the templates, which are merely TPL programs. We provide templates for both C and Ada code generation that, when compiled, will produce what is called a standalone simulation executable.

Templates and the template programming language (TPL) are described in the *Template Programming Language User's Guide*.

6.4 BlockScript Block

The block algorithms for supplied blocks cannot be modified or customized through AutoCode templates. However, you can create your own block by specifying the algorithm in a BlockScript Block (BSB). A BSB uses a scripting language called BlockScript that is translated into C or Ada code and it is generated along with the other blocks in the system. BlockScript is documented in the *AutoCode Reference* and the *SystemBuild User's Guide*.

Example 6-1 shows a user-defined BlockScript algorithm calculating the average of 5 numbers (using BlockScript while loop).

EXAMPLE 6-1: Example BlockScript Using While Loop

```
Outputs: y;
parameters: p;
Float y, p(5);
Float sum;

sum=0.0;
k=1;
While k<p.size Do
   sum=sum+p(k);
   k=k+1;
```

```
EndWhile;
y=sum/p.size;
```

Resulting generated C Code segment:

```
/*----------  BlockScript */
   /* {gplvar.Thru_Var.3} */
      sum = 0.0;
      k = 1;
      while (k < 5) {
         sum = sum + myvar[-1+k];
         k = k + 1;
      }
      Y->Thru_Var_1 = sum/5;
```

Resulting generated Ada Code segment:

```
   ----------  BlockScript  --
   -- {gplvar.Thru_Var1.3} --
      sum := 0.0;
      k := 1;
      while k < 5 loop
         sum := sum + myvar(-1+k);
         k := k + 1;
      end loop;
      Y.Thru_Var_1 := sum/RT_FLOAT(5);
```

The `parameters: p;` statement causes the Parameters View in the BlockScript form to include a new 5-by-1 `Parameter p`. A %variable `%myvar` has been defined for this parameter. This causes AutoCode to replace all occurrences of `p` in the previous and following scripts to be replaced by `myvar`.

For more information about programming BlockScripts, see the *AutoCode Reference* and the *SystemBuild User's Guide*.

6.5 Data Parameterization

AutoCode users have a choice of generating block data with constant values entered in the Block form or to use Xmath variables (%variables) to represent the data symbolically. While generating code from the SystemBuild menu, this choice is made via the `Block Parameters` option, which can have values of `% Xmath vars` or `Block Defaults`. Example 6-2 shows generated code for a gain block using block default data, and Example 6-3 shows generated code for a gain block using an Xmath variable called `gainvar`, which is initialized to 5.6 in the Xmath partition.

EXAMPLE 6-2: Generated Code for a Gain Block Using Block Default Data

```
y = 2.3 * u;
```

EXAMPLE 6-3: Generated Code for a Gain Block Using Xmath Initialized Variable

```
0    C generated code:
     VAR_FLOAT gainvar = 5.6;
     . . .
     y = gainvar * U->gainvar_1;
```

CAUTION: **Changing of %variables values can in certain cases, such as feedback loops, cause the blocks to be executed in an incorrect order. The result of the application might not match the SystemBuild simulation.**

6.6 UserCode Block

A UserCode Block (UCB) provides a strict interface between an AutoCode-generated system and some other code. The idea is that you can implement a particular functionality more efficiently by supplying code rather than attempting to model it within SystemBuild. Such functionality includes operations dealing with hardware or reusing existing code found in libraries.

A UCB can also be used to increase the performance of the SystemBuild Simulator by linking back code within a UCB directly to the simulator. The code can be either handwritten or discrete procedural code generated by AutoCode.

For more information about the UCB interface and linking back into the Simulator, see the *AutoCode Reference*.

6.7 Macro Procedure Block

A Macro Procedure block provides a C-macro like capability in the generated code. The macro's functionality is to be modeled within SystemBuild so that the simulation matches, but within the generated code, only a macro name will be generated. For example, you can model a function that returns the maximum of 2 numbers, but it is much more efficient to use the MAX macro provided in C. Therefore, the implementation of a macro procedure must be supplied by you in the generated code for the code to compile. Macro procedures provide an inline capability for small code

fragments. See the *AutoCode Reference* for more details about the code generated for a Macro Procedure block.

NOTE: AutoCode supports macro procedure blocks for Ada as well as C. However, there is no standard C-macro like capability in Ada. Therefore, we recommend that you implement the Macro Procedure as a standard procedure and use the INLINE pragma.

6.8 User-Defined Code Comments

AutoCode provides the following tokens for adding comments within generated code:

- `blk_code_cmt` for blocks
- `sb_code_cmt` for SuperBlocks
- `ds_code_cmt` for DataStores

These code-comment tokens are predefined user parameters, which are especially useful if you plan to generate documentation as described in the *DocumentIt User's Guide*. For additional information on user parameters, see the *SystemBuild User's Guide*.

6.8.1 Using a User-Defined Code Comment

To use a code-comment token, take the steps described in the following phases.

Phase One

Before code generation:

1. Create an appropriate user-parameter.
2. Within a block, create a user-parameter with the name `blk_code_cmt_s`.

   NOTE: Within a SuperBlock, create the user-parameter with the name `sb_code_cmt_s` and for a DataStore, create a user-parameter with the name `ds_code_cmt_s`.

3. Repeat this process for each of the blocks, SuperBlocks, and DataStores where you want comments to appear in the generated code.

Phase Two

The second phase occurs during code generation. To insert the comments you created within the user-parameters into the generated code, do one of the following:

- Generate code by selecting the **Enable DocumentIt Block Comments** option from the Formatting Tab of the Advanced Dialog of the AutoCode code generation dialog.
- Use the `docit` AutoCode keyword.
- Use the `-doc` Xmath command option.

After code is generated, the comments placed within a block's `blk_code_cmt_s` user-parameter appear where the block appears in the generated code. For SuperBlocks, only Procedure SuperBlocks have the comments placed within the generated code. Those comments from the `sb_code_cmt_s` user-parameter are placed at the definition of the function that represents the Procedure SuperBlock. For DataStores, the comments within the `ds_code_cmt_s` are placed at the definition of the DataStore.

ISI recommends that the content of the user-parameters be plain-text, rather than Rich Text Format or other formatted text content, because the contents are placed within code.

6.8.2 Limitations

The code-comment tokens have the following limitations:

- Any "basic block" can use the `blk_code_cmt_s` user-parameter, including the SuperBlock Block (that is, a SuperBlock reference).
- Any DataStore can use the `ds_code_cmt_s` user-parameter.
- Any SuperBlock definition can use the `sb_code_cmt_s`, but only for Procedure SuperBlocks (all variations) will the comments appear in the code.

7 Introduction to Software Constructs with AutoCode

This chapter is an introduction to the blocks that implement typical software constructs just as loops and decision statements. This includes UserCode Blocks, Macro Procedure Blocks, and Procedure SuperBlocks.

7.1 Introduction

We call blocks that implement typical software logic (such as loops and decision statements) *software constructs* to differentiate from other blocks that compute a result (that is, functional blocks). In other words, software construct blocks deal with the control flow of the program rather than the data flow of the model, allowing the automatically generated code to more closely mimic handwritten code.

NOTE: Unless otherwise noted, these blocks are not supported in Continuous SuperBlocks.

7.2 Standard Procedure SuperBlocks

Standard Procedure SuperBlocks are included in the discussion of software constructs because a procedure represents good software engineering by creating a modular piece of code that can be reused throughout the model. When procedures are reused, code size can be greatly reduced. Maintenance of your design is made easier as fixes are made only in one place. Testing is more tractable as a procedure defines an encapsulated unit that can be independently tested, validated, and verified. ISI recommends that you use Standard Procedures within your model.

7.3 Variable Blocks

Variable Blocks represent actual variables within the generated code. SystemBuild has two types of Variable Blocks, one that represents a global variable and the other that represents a local variable.

7.3.1 Global

A Global Variable Block is used for a variety of purposes. Traditionally, it has been used to communicate information between Asynchronous Procedure SuperBlocks and subsystems. Also, data shared across multiple processors can be easily accessed. Another usage of Global Variable Blocks is to provide persistent data during execution of the system. Standard Procedures can use Global Variable Blocks as well.

Global Variable Blocks represent global variables in the code. Therefore, Global Variable Blocks are implemented to preserve determinancy. This is only an issue for multi-rate and multi-processor systems. However, even for single-rate systems, overhead and special semantics are associated with a Global Variable Block when data is read and written.

Generally speaking, all reads from a Global Variable Block occur at the beginning of the subsystem for that time point or activation frame, while all writes to the Global Variable Block occur at the end of the subsystem or activation frame. See the *AutoCode Reference* for more details.

7.3.2 Local

A Local Variable Block is similar to a Global Variable Block, except Local Variable Blocks represent local variables in the generated code. Therefore, a Local Variable Block *cannot* hold persistent data, and *cannot* be used to communicate information across processors, subsystems, and procedures. Local Variable Blocks provide efficient communication within a subsystem, and are used with the Iterator and IfThenElse Blocks.

7.4 IfThenElse Block

The IfThenElse Block implements a decision within the generated code, and then executes one sequence of blocks. This type of block is like an `if` statement in C or Ada.

7.5 Iterator Block

The While Block provides a container that defines blocks that will continue to execute until a condition or set of conditions is met. This type of block is like a `while` loop in C and Ada.

7.6 Explicit Block Sequencing

The SystemBuild Analyzer and AutoCode automatically determine the sequence in which blocks are executed. However, there may be algorithms you design that require a set of blocks to be executed before another set of blocks. You control the sequencing of blocks by using the Sequencer Block to divide the diagram into frames; the frame on the left side executes before the frame on the right side. No code is generated for a Sequencer Block.

Explicit sequencing is critical for managing blocks such as Global and Local Variable Blocks, IfThenElse Blocks, and possibly Standard Procedure SuperBlocks.

7.7 Example Model

Figure 7-1 shows a model with software constructs, and Example 7-1 on page 7-4 shows the code generated for this model.

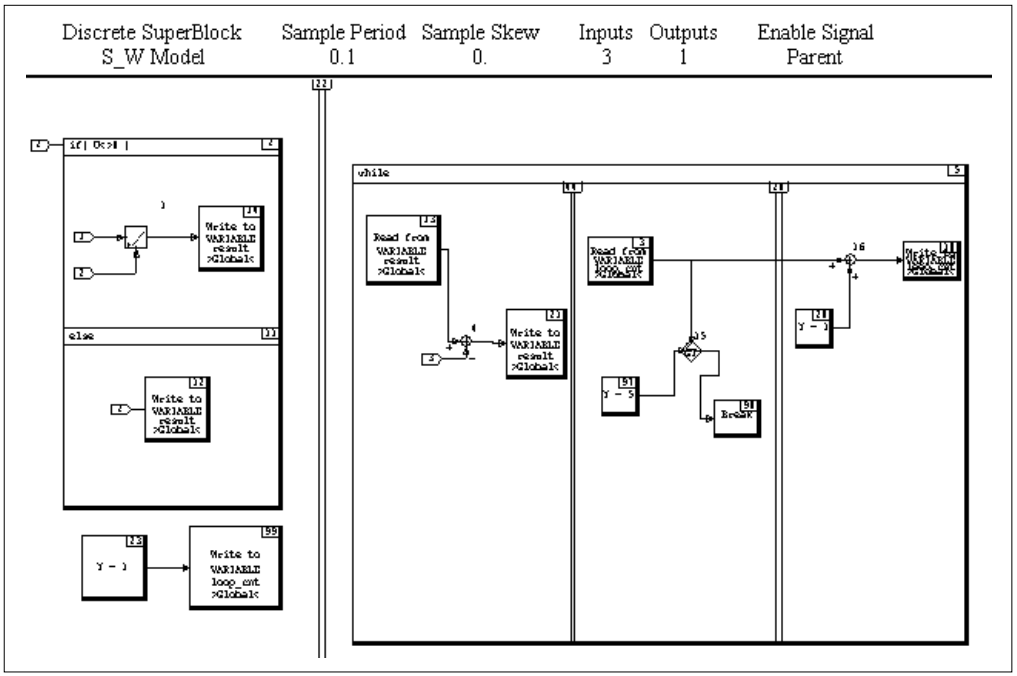


FIGURE 7-1 Sample Model with Software Constructs

The following C code (Example 7-1) was generated from the model in Figure 7-1 on page 7-4 with the Variable Block Read and Constant Propagation optimizations.

EXAMPLE 7-1: Generated Software from Software Constructs Model (excerpt)

```
/*****************************************************************************
|                  AutoCode/C (TM) Code Generator V6.X                       |
|            INTEGRATED SYSTEMS INC., SUNNYVALE, CALIFORNIA                 |
*****************************************************************************
rtf filename           : S_W_Model.rtf
Filename               : S_W_Model.c
Dac filename           : c_sim.dac
Generated on           : Wed Jun  2 19:10:16 1999
Dac file created on    : Thu Mar 25 10:10:09 1999
Options                : -l c
--
--   Number of External Input : 3
--   Number of External Output: 1
--
```

```
--   Scheduler Frequency:    10.0
--
--   SUBSYSTEM  FREQUENCY  TIME_SKEW  OUTPUT_TIME  TASK_TYPE
--   ---------  ---------  ---------  -----------  ---------
--   1          10.0       0.0        0.0          PERIODIC
*/

#include <stdio.h>
#include <math.h>
#include "sa_sys.h"
#include "sa_defn.h"
#include "sa_types.h"
#include "sa_math.h"
#include "sa_matrix.h"
#include "sa_user.h"
#include "sa_utils.h"
#include "sa_time.h"
#include "sa_fuzzy.h"

/*** System Data ***/

/******* Structure to drive disconnected input/output. *******/

struct _DcZero {
   RT_FLOAT dzero;
};

static const struct _DcZero dczero = {0.0};

#define EPSILON                     1.49011611938476562E-08
#define EPS                         (4.0 * EPSILON)
#define ABSTOL                      EPSILON
#define XREMAP                      1

#define SCHEDULER_FREQ               10.0
#define NTASKS                       1
#define NUMIN                        3
#define NUMOUT                       1
#define SCHEDULER_ID                 0
#define PREEMPTABLE                  2

enum TASK_STATE_TYPE { IDLE, RUNNING, BLOCKED, UNALLOCATED };

static RT_INTEGER                    ERROR_FLAG  [NTASKS+1];
static RT_BOOLEAN                    SUBSYS_PREINIT [NTASKS+1];
static RT_BOOLEAN                    SUBSYS_INIT [NTASKS+1];
static enum TASK_STATE_TYPE          TASK_STATE  [NTASKS+1];

/******* System Ext I/O and Sample-Hold type declarations. *******/
struct _Sys_ExtOut {
   RT_FLOAT dzero;
};
```

```
struct _Sys_ExtIn {
   RT_FLOAT S_W_Model_1;
   RT_FLOAT S_W_Model_2;
   RT_FLOAT S_W_Model_3;
};

struct _Subsys_1_in {
   RT_FLOAT S_W_Model_1;
   RT_FLOAT S_W_Model_2;
   RT_FLOAT S_W_Model_3;
};

/**** System Ext I/O and Subsystem I/O type definitions and     ****
 **** Pointers to SubSystem Outputs ReadOnly/Work areas.       ****/
struct _Sys_ExtOut sys_extout;
struct _Sys_ExtIn sys_extin;
struct _Subsys_1_in subsys_1_in;

static RT_FLOAT                       ExtIn       [NUMIN+1];
static RT_FLOAT                       ExtOut      [NUMOUT+1];

/* Model variable definitions. */
VAR_INTEGER loop_cnt;
VAR_FLOAT result;
/* Model variable declarations. */
extern VAR_INTEGER loop_cnt;
extern VAR_FLOAT result;

/******** Tasks declarations ********/

/******* Subsystem 1  *******/
extern void subsys_1(  struct _Subsys_1_in *U);

/******** Tasks code ********/

/******* Subsystem 1  *******/

void subsys_1(   struct _Subsys_1_in *U
)
{
      static RT_INTEGER iinfo[4];

      /***** Local Block Outputs. *****/

      RT_INTEGER S_W_Model_23_1;
      RT_FLOAT S_W_Model_1_1;
      RT_FLOAT S_W_Model_13_1;
      RT_FLOAT S_W_Model_4_1;
      RT_INTEGER S_W_Model_3_1;
      RT_INTEGER S_W_Model_97_1;
      RT_FLOAT S_W_Model_15_1;
```

```
RT_INTEGER S_W_Model_20_1;
RT_INTEGER S_W_Model_16_1;

/******* Initialization. *******/

if (SUBSYS_PREINIT[1]) {
   iinfo[0] = 0;
   iinfo[1] = 1;
   iinfo[2] = 1;
   iinfo[3] = 1;
   SUBSYS_PREINIT[1] = FALSE;
   return;
}

/***** Output Update. *****/
/* ---------------------------  IfThenElse */
/* {S_W Model..2} */
if( U->S_W_Model_2 != 0.0 ) {

   /* ---------------------------  ElementDivision */
   /* {S_W Model..1} */
   S_W_Model_1_1 = U->S_W_Model_1/U->S_W_Model_2;
   /* ---------------------------  Write to Variable */
   /* {S_W Model..14} */
   result = S_W_Model_1_1;

}
else {

  /* ---------------------------  Write to Variable */
  /* {S_W Model..12} */
  result = U->S_W_Model_2;

}
/* ---------------------------  Algebraic Expression */
/* {S_W Model..23} */
S_W_Model_23_1 = 1;

/* ---------------------------  Write to Variable */
/* {S_W Model..99} */
loop_cnt = S_W_Model_23_1;

/* ---------------------------  While */
/* {S_W Model..5} */
while (TRUE) {

        /* ---------------------------  Read from Variable */
        /* {S_W Model..13} */
        S_W_Model_13_1 = result;
        /* ---------------------------  Summer */
        /* {S_W Model..4} */
        S_W_Model_4_1 = S_W_Model_13_1 - U->S_W_Model_3;
```

```
/* ----------------------------  Write to Variable */
/* {S_W Model..21} */
result = S_W_Model_4_1;
/* ----------------------------  Read from Variable */
/* {S_W Model..3} */
S_W_Model_3_1 = loop_cnt;
/* ----------------------------  Algebraic Expression */
/* {S_W Model..97} */
S_W_Model_97_1 = 5;
/* ----------------------------  Relational Operator -- LT-EQ-GT */
test = S_W_Model_3_1 > S_W_Model_97_1;
/* ----------------------------  Break */
/* {S_W Model..98} */
if( test ) {
   break;
}
/* ----------------------------  Algebraic Expression */
/* {S_W Model..20} */
S_W_Model_20_1 = 1;
/* ----------------------------  Summer */
/* {S_W Model..16} */
S_W_Model_16_1 = S_W_Model_3_1 + S_W_Model_20_1;
/* ----------------------------  Write to Variable */
/* {S_W Model..10} */
loop_cnt = S_W_Model_16_1;
}

if(iinfo[1]) {
   SUBSYS_INIT[1] = FALSE;
   iinfo[1] = 0;
}
return;
EXEC_ERROR: ERROR_FLAG[1] = iinfo[0];
   iinfo[0]=0;
}
```

A AutoCode Options

This appendix describes options that can be used when invoking AutoCode from within Xmath or from the OS command line. This appendix also describes how to use an `autostart.opt` file. This appendix supplements Chapter 2, *Using AutoCode*.

A.1 Options When Invoking AutoCode

As described in Chapter 2, AutoCode can be invoked from the Catalog Browser, the Xmath Commands window, or the operating system command line. Table A-1 lists the various AutoCode command-line options. The code generator is invoked by using the `autocode` command (Xmath) or the `autostar` command (command line).

TABLE A-1 Options When Invoking AutoCode

Option from Xmath	Option from O/S	Description
`allgscope`	`-allgscope`	Force all Output Scopes to be Global and all procedure Input Scopes to be Local.
`arraymin`	`-Oarray` *n*	Minimum size of vectorized arrays. (default: 2)
`backpmap`	`-bmap` *map*	A string specifying the map associating background procedures with processors. The syntax parallels that specified for the `subsysmap` option, except procedure numbers are used rather than task numbers.
`config`		Replaced by `options`.

TABLE A-1 Options When Invoking AutoCode (Continued)

Option from Xmath	Option from O/S	Description
`csi`	`-csi` *n*	`csi` specifies the continuous task sample interval (see Section 5.4.2 on page 5-4 for details.). **NOTE:** Use the `csi` option so generated code will match sim results for continuous systems.
`docit`	`-doc`	Enables the DocumentIt tokens as described in the *Template Programming Language User's Guide*.
`doublebuf`	`-doublebuf`	Force double-buffering for single-rate systems.
`epinfo`	`-epi`	Boolean (default=0). Generate extended procedure information data structures. Procedures generated with this option should not automatically be mixed with those generated without it.
`epsilon`	`-eps`	Float (default is machine epsilon). Set the value of `epsilon` used in the generated code. This option is used to initialize model outputs. `epsilon` is set to a very small value so that inital outputs will be near zero; this prevents division by zero problems. The AutoCode token `epsilon_r` is used to access the `epsilon` value. If `epsilon` is left blank, a default value is used.
`errcheck`	`-e`	Boolean (default=0). Enables error checking in the generated code. Default is 0, error checking disabled.
`file`	`-o` *file*	The default name is taken from the name of the model file; the default extension is `.c` or `.a`, depending on the language chosen.

`fmarker`	`-f`	Boolean (default=0). When true (1), this option's value forces single-precision floating-point markers for encoded numbers (C language option only).
		Example:
		no `-f` option, generated code looks like
		`y=2.3 * u;`
		with `-f` option, generated code looks like
		`y=2.3f * u.`

TABLE A-1 Options When Invoking AutoCode (Continued)

Option from Xmath	Option from O/S	Description
`foverflow`	`-ovfp` *n*	Integer (default=2). Indicate state of overflow protection for integer and fixed-point calculations. 0 = overflow protection disabled 1 = overflow protection forced 2 = overflow protection selected by block's option
`glbvarblkopt`	`-Ogvarblk`	Optimize read-from global varblocks.
See [a] on page A-8.	`-h`	Obtains a help display.
`ialg`	`-i` *n*	`ialg` specifies the selected integrator as one of the following: 1. First order Runge-Kutta -- Euler 2. Second order Runge-Kutta -- Modified Euler 3. Fourth order Runge-Kutta -- Simpson's Second Rule) 4. Kutta-Merson 5. User Integrator See Section 5.4.2 on page 5-4 for details.
`indent`	`-indent` *n*	This integer value specifies the amount of indentation in output between levels. Default is 3.
`initmerge`	`-Oinitmerge`	Merge block INIT sections into one INIT section, if possible.
`interpmap`	`-imap` *map*	A string specifying the map associating interrupt procedures with processors. The syntax parallels that of the backpmap option.
`ipath`	`-I` *pathname*	Adds a *pathname* to the list of directories in which to search for template `@include` files. Can be used multiple times on the command line (limit 10).
`krstyle`	`-kr`	Generate old-style (Kernighan and Ritchie) C function prototypes.
`language`	`-l` *lang*	The language for generating the code: C or Ada. The following are accepted: `c`, `C`; `a`, `ada`, `Ada`, `ADA`.
`linesz`	`-linesz` *n*	This integer value specifies the maximum number of output characters per line. The integer value must be ≥ 78. The default is 80.

A

TABLE A-1 Options When Invoking AutoCode (Continued)

Option from Xmath	Option from O/S	Description
`loopmin`	`-Oloop` *n*	Loop threshold for vectorized code (default: 2).
`locvarblkopt`	`-Olvarblk`	Optimize read-from local varblocks.
`mapfile`	`-pfile` *file*	A string defining the map file associating subsystems and background, startup, and interrupt procedures with processors. The subsysmap, backpmap, startpmap, and interpmap options override the specifications in this file, and if none of these options is supplied and the file doesn't exist, it's created using a default map. A single-line comment is indicated using `//` characters.
`minsf`	`-minsf` *n*	Specifies minimum AutoCode scheduler frequency. The real-time scheduler frequency is set to the larger value of the frequency determined by the block diagram application and the value specified by `-minsf`. Normally, the default value of 0.0 should be used, which allows the application to set its own scheduler frequency. Deviation from this default should be approached with caution, as a consistent scheduler frequency should normally be based on a least common multiple of the inverse of the application timing requirements (i.e., frequencies).
`namelen`	`-nl` *n*	This integer value adjusts the maximum variable length in the generated code. The integer value must be ≥ 20 (the default is 48).
`nodiscon`	`-odiscnout`	Optimize away disconnected outputs.
`noerr`	`-noerr`	Do not generate error detection code after a procedure call is made.
`nogscope`	`-nogscope`	Force all Output Scopes to be Local.
`noinfo`	`-noinfo`	If possible, eliminate a procedure's INFO structure.
`noicmap`	`-noicmap`	Sets the constant variable XREMAP to False, which prevents initial values of states from being set.
`nomap`	`-nomap`	Boolean (default=0). Turn off the structure map indicating subsystem and system external inputs and outputs by setting the `nobusmap_b` token to True.
`norestart`	`-Onorestart`	Optimize out the restart capability.
`nosmooth`	`-nosmooth`	Turn off floating-point constant number smoothing.

TABLE A-1 Options When Invoking AutoCode (Continued)

Option from Xmath	Option from O/S	Description
`nouy`	`-nouy`	Pass procedure input and outputs as actual arguments to the function.
`numproc`	`-np` *n*	Integer (default=1). The number of processors to generate code for.
`options`	`-opt` *file*	Specifies the name of the options file. Options are entered in the file using the same syntax as if they were specified in the command line. The exception is that map specifications are not enclosed between quotes. Options c an be on one line, separate lines or a combination. Command-line options override all of the options in the `-opt` file. The `rtf` file name cannot be specified in the `-opt` file. Single line comments are done by using `//` characters. See Section A.2 on page A-9 for more information about the options file.
`parname`	`-p`	Boolean (default=0). When true (1), specifies use of parameter names specified by scripts in language blocks instead of RP and IP arrays.
`priomap`	`-prio`	A string defining the priority of the subsystems. It has a form similar to that of the skewmap option (see above), except <skew value> is replaced by an integer priority. Provided that AutoCode can assign each subsystem a unique priority while obeying the priomap, range and list operators in the priomap are permitted for both subsystems and priorities.
`procs_only`	`-procs`	Sets the template parameter `procs_only_b` as `true`, and default template only generates Procedure SuperBlocks and generates UCBs and subsystem wrappers for each of these procedures.
	`-prompt`	Prompt for command-line options.
`propconst`	`-Opc`	Propagate constants across blocks.
`reuse`	`-Oreuse` *n*	Strategy for reusing subsystem local outputs. 0 : Do not reuse (default) 1 : Reuse by matching named outputs 2 : Reuse whenever possible.
`roundfloat`	`-round`	Force an implicit float-to-integer conversion to be rounded rather than truncated.

TABLE A-1 Options When Invoking AutoCode (Continued)

Option from Xmath	Option from O/S	Description
`rtf`	See note [b] on page A-8.	The name of the generated real-time file (`.rtf`). Default: `modelname.rtf`
`rtos`	`-rtos`	Boolean (default=0). Read the default Real-Time Operating System configuration file, `ac_rtos.cfg`, to obtain RTOS parameter information (or create `ac_rtos.cfg` if it doesn't exist). This option is incompatible with the rtosfile, subsysmap, startpmap, backpmap, interpmap, and priomap options. More information is given in Chapter 2 of the *AutoCode Reference*.
`rtosfile`	`-rtosf` *`file`*	A string specifying the name of the file from which to read the Real-Time Operating System configuration information. This option is incompatible with the RTOS, subsysmap, startpmap, backpmap, interpmap, and priomap options.
`scheduler`	`-sched` *`n`*	Choose a scheduler type: 0 : one-stage output posting (default) 1 : pre&post output posting
`sd`	`-sd` *`n`*	Integer. Specifies the number of significant digits for encoded numbers. Default: long constants are emitted to full machine width.
`skewmap`	`-skew` *`n`*	A string defining the skew of each subsystem. It has the form: <skewmap> :== <subsystem #> <skew value> { <skewmap> ... } <subsystem #>:== An integer naming the subsystem <skew value> :== A float defining the skew Subsystem numbers and skew values must be separated by a single space in the string. Optionally, a range or list of subsystem numbers can be used instead of <subsystem #> above.
`smcallout`	`-smco`	Boolean (default=0). Generate call-outs for access to all elements in shared memory. Turn on the shared memory function call out.
`startpmap`	`-smap` *`map`*	A string specifying the map associating startup procedures with processors. The syntax parallels that of the `backpmap` option.

TABLE A-1 Options When Invoking AutoCode (Continued)

Option from Xmath	Option from O/S	Description
`subsysmap`	`-pmap` *`map`*	A string specifying the map associating subsystems with processors. The map contained in the string has the form: <map> :== <prsr #> <list> { <map> ... } <prsr #> :== a processor number (starting at 1) <list> :== <task #>, { <task #> ... } <task #> :== a task number (starting at 0) Processor numbers and lists must be separated by a single space, while elements of the list must be separated by a single comma.
`tpldac`	`-d` *`file`*	A direct access template file (see Chapter 4). Default: `$CASE/ACC/templates/c_sim.dac` for C `$CASE/ACA/templates/ada_rt.dac` for Ada
`tplsrc`	`-t` *`file`*	This is a template source file (see Chapter 4). Default: `$CASE/ACC/templates/c_sim.tpl` for C `$CASE/ACA/templates/ada_rt.tpl` for Ada
`typecheck`	See note [c]	Boolean (default=1). Enables data type checking. See the SystemBuild online help for simulation for details about data typing. For AutoCode the default is 1; type checking is enabled. If `typecheck` is set to false, all the variables in the model are hardcoded with float data type.
`ucbparams`	`-ucbparams`	Use RP/IP temporaries as actuals to UCB call instead of %var variables.
`vars`	See note [d] on page A-8.	Boolean (default=1). Same functionality as the `sim` keyword by that name (see the `sim` chapter of the *SystemBuild User's Guide*). The default is 1, meaning that %vars are included in the model. Note that if the code is generated from a model, the model is processed using the `vars` keyword. However, if the AutoCode function is invoked from the operating system command line using an existing `.rtf`, the `vars` keyword is ignored. To turn off `vars`, specify `!vars`.

TABLE A-1 Options When Invoking AutoCode (Continued)

Option from Xmath	Option from O/S	Description
`vbcallout`	`-vbco`	Boolean (default=0). Generate call-outs around the critical section of variable block accesses in the generated code.
`vectormode`	`-Ov` *n*	Vectorization option: 0 : Scalar code generation (default) 1 : Vectorize based on labeling 2 : Maximal vectorization

a. There is no keyword, but has the command **`help autocode`** for Netscape help.

b. The name of the `rtf` file must always be specified when invoking from the command line.

c. The `typecheck` feature applies only to the creation of the `rtf` file, thus there is no equivalent option from the command line.

d. This Xmath option is used for creation of the `rtf`. When invoking from the O/S, the `rtf` must already exist; therefore, there is no O/S option equivalent.

A.2 Using the `autostar.opt` File

If you invoke AutoCode with the same options consistently, you can put these options into an options file, saving a lot of error-prone, repetitive typing each time you invoke AutoCode. AutoCode reads the options file at startup, and performs the options as though you had entered them on the command line. Although you can use an options file whether you invoke AutoCode from the Xmath Commands window or the operating system command line, the only options that you can specify in the options file are operating system command-line options.

The default options file is `autostar.opt`. If you have an `autostar.opt` file in the current working directory from which you invoke AutoCode, the options in that file will be executed when you invoke AutoCode. If you specify an option on the command line that is also in the options file, the command line option overrides the same option in the options file (see Example A-1 and the paragraphs following).

For different applications, you might need to invoke AutoCode differently. For this reason, you can have multiple options files. To invoke AutoCode with an options file other than `autostar.opt`, specify the name of the options file when you invoke AutoCode (see Example A-2 and the paragraphs following).

Options are entered into the options file using the same syntax as if they were specified in the command line. The exception is that map specifications are not enclosed between quotes. Options can be on one line, separate lines or a combination. The `rtf` file name cannot be specified in the options file. Single line comments are done by using `//` characters.

Example A-1 shows an options file.

EXAMPLE A-1: Example `autostar.opt` Options File

```
// Sample options file //
-l c
-t c386_c860_mb2.tpl
-o myoutput
```

To use this file, invoke AutoCode as follows:

```
autostar model.rtf
```

This invokes AutoCode with the `autostar.opt` options file. Output is in file `myoutput`.

Consider the following command:

```
autostar -o myoutput3 model.rtf
```

The options file is again used, but the output file option (`-o`) is specified on the command line, so it overrides the corresponding command in the options file. The output documentation will be in file `myoutput3`, not in file `myoutput` as specified in the options file.

Example A-2 shows an options file called `myopt.opt`.

EXAMPLE A-2: Example Options File Called `myopt.opt`

```
// Sample options file //
-l c
-t c386_c860_mb2.tpl
-o myoutput2
```

To use this file, invoke AutoCode as follows:

```
autostar -opt myopt.opt model.rtf
```

This invokes AutoCode with the `myopt.opt` options file.

So, if you have both of the above option files (shown in Examples A-1 and A-2) in your directory, invoking autostar without the `-opt` option puts the generated code into file `myoutput` (the `autostar.opt` options file is used). Invoking autostar with the `-opt myopt.opt` option as shown above puts the generated code into file `myoutput2`.

A.3 Mapping Options

This section describes the options for controlling subsystem execution.

A.3.1 Setting Subsystem Priorities

If you do not specify any priorities, AutoCode allocates priorities to the subsystems in the following manner. For C it allocates priority 0 to the scheduler, and 1, 2, ..., total number of subsystems to subsystem 1, subsystem 2, ..., respectively. For Ada by default, it allocates the number equal to the total number of subsystems to the scheduler, and grows downward for subsystem 1, subsystem 2, ... , respectively.

'For C or Ada, if you specify only the scheduler priority using the `-prio` option, the subsystems will be given the priority 1 plus the scheduler priority. Or, if at least one more subsystem is specified, depending on the sequence of priorities you establish in specifying subsystems, the priority order will be ascending or descending.

Depending on the operating system you are using, you might need to change the default mapping. In some operating systems, the smaller the number, the higher the priority, but in others, the opposite sequence might prevail. You can change the priorities by using the option `-prio`. Syntax:

`-prio subsystem#  priority`

When specified in the command line, the syntax is

`-prio "subsystem# priority"`

Example:

`-prio 0 30 1..3 29 3..n 28.`

In command line the above example would be:

`-prio "0 30 1..3 29 4..n 28"`

This example sets the priority 30 to scheduler subsystem 0 and 29 to subsystems 1,2 and 3 and priority 28 to subsystem 4 onwards. n is always 65535. Be sure to place the scheduler priority at the beginning of the `-prio` option. The subsystem# and priorities must be given in pairs and delimited by a space.

The priority of a subsystem, if not specified, depends on the previous and successor subsystems. The system will not allocate a priority greater than the successor subsystem. AutoCode will not allocate a negative subsystem ID. Every subsystem priority must be less than the scheduler subsystem priority, whether the sequence of priority numbers is ascending or descending. '..' is used to specify the range and ',' is for listing. In `-pfile` the option can be:

-prio 0 30

1,3 29

4..7 28

As no subsystem can have more than one priority, the priorities cannot have the range or list operators '..', ','.

Example:

-prio 0 30

1 28..26

2 25,24 is INVALID, but

Example:

-prio 0 30

1..n 9..2 is VALID and subsystems 1 to n will be assigned priorities between 9 and 2.

A.3.2 Setting Subsystem Skews

The `-skew` option allows skews specified in SystemBuild to be changed. One use of this option is to reset the skews which have been applied to SuperBlocks in order to split a large subsystem into two or more parts. A reason for splitting up such a subsystem is that the parts can be run in parallel on multiple processors. Resetting the skew ensures that they all start at the same time.

The syntax of the skew option is similar to that of `-prio` option.

-skew subsystem# skew

Example:

-skew 1 .1 2 .002 and -skew "1 .1 2 .002" for command line.

Only the subsystems can use the range and list operators '..', ','.

Example:

```
-skew 1     .1
      2..5  .002
      6,8   .003
```

A.3.3 Setting Processor Subsystem Map

When generating code for multiple processors, a subsystem-to-processor map must be specified. If no mapping is specified, AutoCode will map the subsystems to different processors using the following rule:

```
Processor No. = ((subsys_id-1)% max. no. of processors) + 1
```

For a system with six subsystems and 2 processors, this would assign subsystems 1, 3, and 5 to processor 1 and subsystems 2, 4, and 6 to processor 2. Scheduler 0 is always assigned to processor 1. When the default mapping rule is used, the mapping is saved in file `autocode.pmp` in the working directory. You can edit this file and specify this file name for future invocations of AutoCode using the `-pfile` option.

If the `-pfile` option is specified but the file does not exist, AutoCode will create the file and save the default mapping to it rather than to `autocode.pmp`. Note that `-pmap` specifications are not saved to the file specified by the `-pfile` option.

To change the default processor mapping, use the `-pmap` option. The syntax is similar to that of `-prio` and `-skew` options.

```
-pmap processor#  subsystem#
```

Example:

```
-pmap  1  0 2 2,3,4 3 5,6

-pmap "1  0 2 2,3,4 3 5,6" (Command Line)
```

The above example allocates subsystem 0 to processor 1 and 2, 3, and 4 to 2 and 5, 6 to 3.

```
-pmap  1  0,1

       2  2..6
```

The subsystems can have the range and list operators '..', ','.

A.3.4 Processor Map Specification on Command Line

This section describes the command-line processor mapping for subsystem tasks, background, startup and interrupt procedure SuperBlocks. The mapping is specified by using the `-pmap`, `-bmap`, `-smap`, and `-imap` options.

The following is the format of these options:

```
<option> "<map>"

where: <option>:== [ -pmap, -bmap, -smap, -imap ]
       <map>   :== <prsr #> <list> { <map> ... }
       <prsr #>:== <processor number starting at 1>
       <list>  :== <task/procedure numbers>

example:
  autostar -l c -pmap "1 0,1,3 2 2,4" -np 2 test.rtf
```

You can use the `-pfile` option to generate a default set of mappings. The example maps subsystems 0 (the scheduler), 1, and 3 to processor 1 and subsystems 2 and 4 to processor 2.

Index

A

B

C

D

E

F

G

N

O

P

R

S

T

U

V

X

Symbols

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at ni.com/support. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting ni.com/support. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit ni.com/training for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.