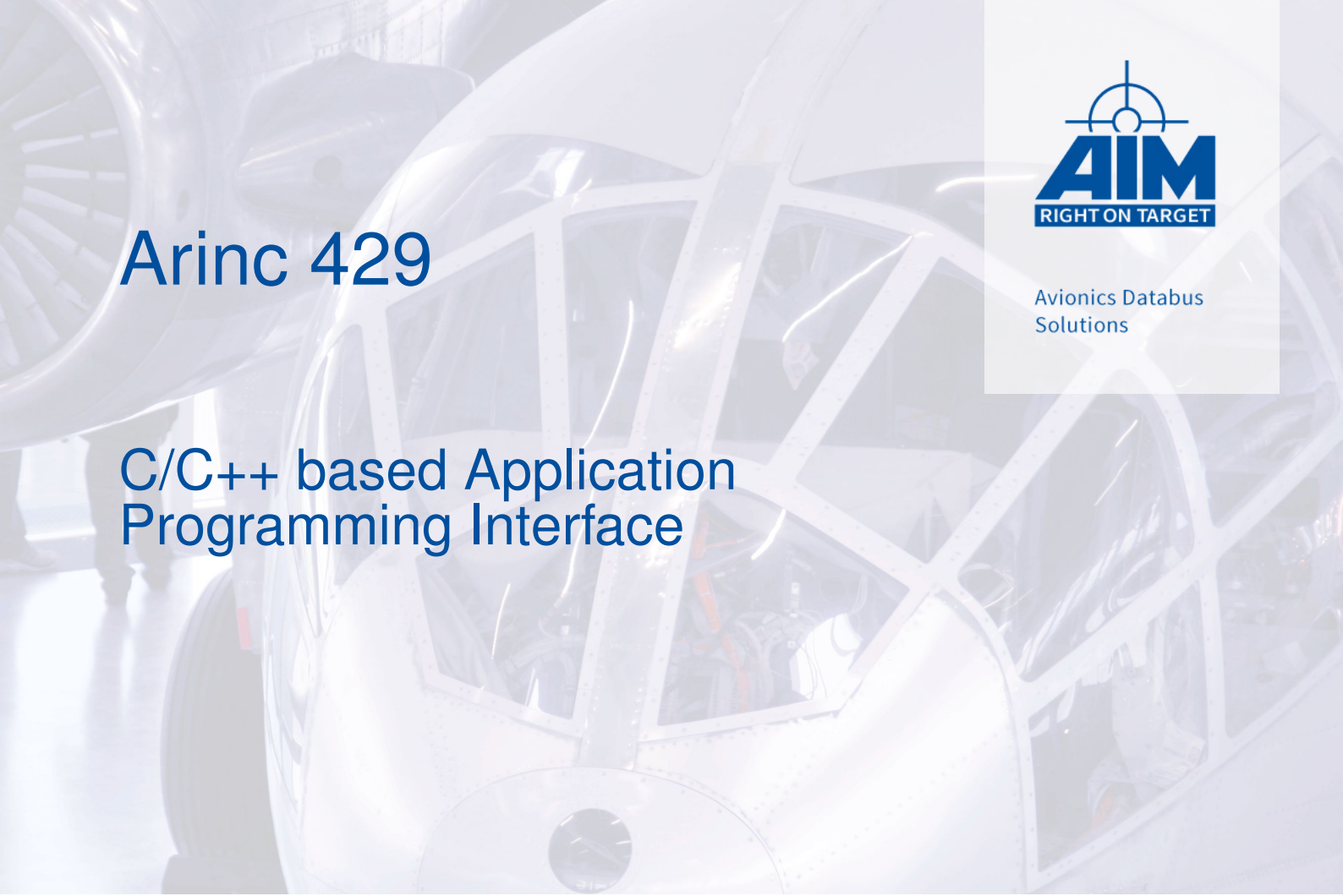


Arinc 429

C/C++ based Application Programming Interface

AIM
RIGHT ON TARGET

Avionics Databus Solutions

Programmer's Guide

Version 9.8.0
June, 2022

www.aim-online.com

Arinc 429

C/C++ based Application Programming Interface

Programmer's Guide

Version 9.8.0
June, 2022

AIM NO. 60-12900-37-9.8.0

AIM – Gesellschaft für angewandte Informatik und Mikroelektronik mbH

AIM GmbH
Sasbacher Str. 2
D-79111 Freiburg / Germany
Phone +49 (0)761 4 52 29-0
Fax +49 (0)761 4 52 29-33
sales@aim-online.com

AIM GmbH – Munich Sales Office
Terofalstr. 23a
D-80689 München / Germany
Phone +49 (0)89 70 92 92-92
Fax +49 (0)89 70 92 92-94
salesgermany@aim-online.com

AIM UK Office
Cressex Enterprise Centre, Lincoln Rd.
High Wycombe, Bucks. HP12 3RB / UK
Phone +44 (0)1494-446844
Fax +44 (0)1494-449324
salesuk@aim-online.com

AIM USA LLC
Seven Neshaminy Interplex
Suite 211 Trevose, PA 19053
Phone 267-982-2600
Fax 215-645-1580
salesusa@aim-online.com

TABLE OF CONTENTS

1 Arinc 429 Specification Overview

The ARINC429 Specification defines the standard requirements for the transfer of digital data between avionics systems on commercial aircraft. ARINC429 is also known as the Mark 33 Digital Information Transfer System (DITS) Specification. Signal levels, timing and protocol characteristics are defined for ease of design implementation and data communications on the Mark 33 DITS bus.

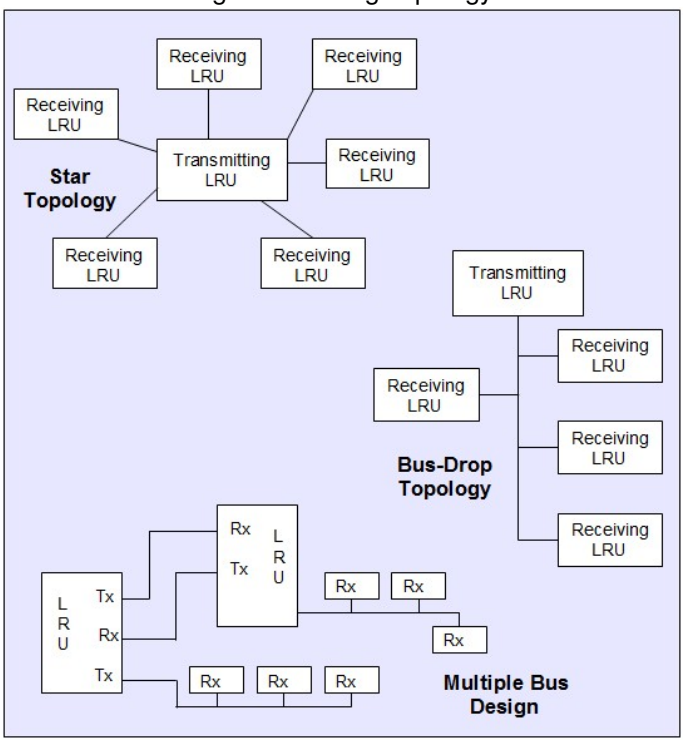
ARINC429 is a privately copywritten specification developed to provide interchangeability and interoperability of line replaceable units (LRUs) in commercial aircraft. Manufacturers of avionics equipment are under no requirement to comply to the ARINC429 Specification, but designing avionics systems to meet the design guidelines provides cross-manufacturer interoperability between functional units.

Detailed information regarding the ARINC429 Specification can be obtained by visiting the ARINC website at www.arinc.com.

1.1 Arinc429 Wiring Topology

ARINC429 is implemented as a simplex, broadcast bus. Each bus consists of a single transmitter – or *source* – connected to from 1-20 receivers – or *sinks* – on one twisted wire pair. Data can be transmitted in one direction only – simplex communication – with bi-directional transmission requiring two channels or buses. The devices, line replaceable units or LRUs, are most commonly configured in a star or bus-drop topology as shown in Figure 1 Wiring Topology. Each LRU may contain multiple transmitters and receivers communicating on different buses. This simple architecture, almost point-to-point wiring, provides a highly reliable transfer of data.

Figure 1: Wiring Topology



1.2 Arinc429 Transmission Characteristics

ARINC429 specifies two speeds for data transmission. Low speed operation is stated at 12.5 kHz, with an actual allowable range of 12 to 14.5 kHz. High speed operation is 100 kHz +/- 1% allowed. These two data rates can not be used on the same transmission bus.

Data is transmitted in a bipolar, Return-to-Zero format. This is a tri-state modulation consisting of HIGH, NULL and LOW states.

Transmission voltages are measured across the output terminals of the source. Voltages presented across the receiver input will be dependent on line length, stub configuration and the number of receivers connected. The following voltage levels indicate the three allowable states:

- **TRANSMIT**

  HIGH +10.0 V +/- 1.0 V

  NULL 0 V +/- 0.5V

  LOW -10.0 V +/- 1.0 V

- **RECEIVE**

  HIGH +6.5 to 13 V

  NULL +2.5 to -2.5 V

  LOW -6.5 to -13 V

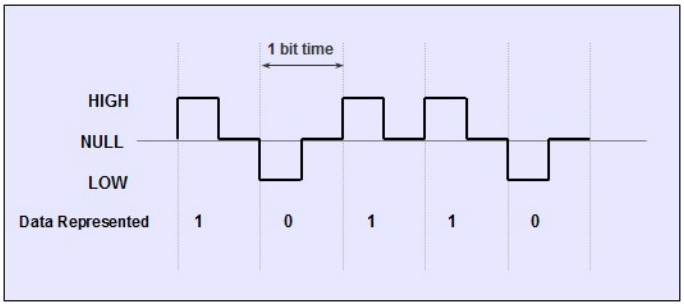
In bipolar, Return-to-Zero (RZ) format, a HIGH (or 1) is achieved with the transmission signal going from NULL to +10 V for the first half of the bit cycle, then returning to zero or NULL.

A LOW (or 0) is produced by the signal dropping from NULL to –10 V for the first half bit cycle, then returning to zero.

With a Return-to-Zero modulation format, each bit cycle time ends with the signal level at 0 Volts, eliminating the need for an external clock, creating a self-clocking signal.

An example of the bipolar, tri-state RZ signal is shown in Figure 2 Bipolar return to zero.

Figure 2: Bipolar return to zero

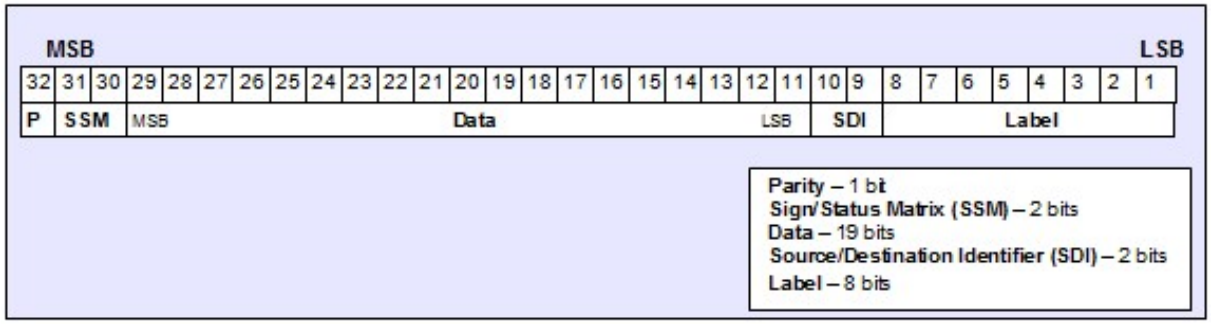


1.3 Arinc429 Protocol and Word Formats

A transmitter may 'talk only' to a number of receivers on the bus, up to 20 on one wire pair, with each receiver continually monitoring for its applicable data, but does not acknowledge receipt of the data. A transmitter may require acknowledgement from a receiver when large amounts of data have been transferred. This handshaking is performed using a particular word style, as opposed to a hard wired handshake. When this two way communication format is required, two twisted pairs constituting two channels are necessary to carry information back and forth, one for each direction.

Transmission from the source LRU is comprised of 32 bit words containing a 24 bit data portion containing the actual information, and an 8 bit label describing the data itself as shown in Figure 3 Arinc 429 Data Word Format.

Figure 3: Arinc 429 Data Word Format



The transmitter is always transmitting, either data words or the NULL state. Most ARINC messages contain only one data word consisting of either Binary (BNR), Binary Coded Decimal (BCD) or alphanumeric data encoded using ISO Alphabet No. 5. File data transfers that send more than one word are also allowed.

Sequential words are separated by at least 4 bit times of null or zero voltage. By utilizing this null gap between words, a separate clock signal is unnecessary.

The only two fields definitively required are the Label and the Parity bit, leaving up to 23 bits available for higher resolution data representation. Many non-standard word formats have been adopted by various manufacturers of avionics equipment. Even with the variations included, all ARINC data is transmitted in 32 bit words. Any unused bits are padded with zeros.

Parity

ARINC429 defines the Most Significant Bit (MSB) of the data word as the Parity bit. ARINC uses odd parity as an error check to ensure accurate data reception. The number of Logic 1s transmitted in each word is an odd number, with bit 32 being set or cleared to obtain the odd count. ARINC429 specifies no means of error correction, only error detection.

Label

Bits 1-8 contain the ARINC Label known as the Information Identifier. The Label is expressed as a 3 digit octal number with receivers programmed to accept up to 255 Labels. The Label's Most Significant Bit resides in the ARINC word's Least Significant Bit location.

The Label is used to identify the word's data type (Binary (BNR), Binary Coded Decimal (BCD), Discrete, etc) and can contain instructions or data reporting information. Labels may be further refined by utilizing the first 3 bits of the data field, Bits 11-13, as an Equipment Identifier to identify the bus transmission source. Equipment IDs are expressed in hexadecimal values.

LRUs have no address assigned through ARINC429, but rather have Equipment ID numbers which allow grouping equipment into systems, which facilitates system management and file transfers.

For example, BNR Label 102 is Selected Altitude. This data can be received from the Flight Management Computer (Equipment ID 002Hex), the DFS System (Equipment ID 020Hex) or the FCC Controller (Equipment ID 0A1Hex).

The Label is always sent first in an ARINC transmission and is a required field, as is the Parity bit. Labels are transmitted MSB first, followed by the rest of the ARINC word, transmitted LSB first.

Source/Destination Identifier (SDI)

The Source/Destination Identifier (SDI) utilizes bits 9-10 and is optional under the ARINC429 Specification. The SDI can be used to identify which source is transmitting the data or by multiple receivers to identify which receiver the data is meant for.

For higher resolution data, bits 9-10 may be used instead of using them as an SDI field. When used as an Identifier, the SDI is interpreted as an extension to the word Label.

Sign/Status Matrix (SSM)

Depending on the Label, which indicates the type of data is being transmitted, the SSM field can provide different information. In BCD data words the bits 30 and 31 are assigned to the SSM, in BNR data words the bits 29, 30 and 31. Each Label has its own unique implementation of the SSM Sign function.

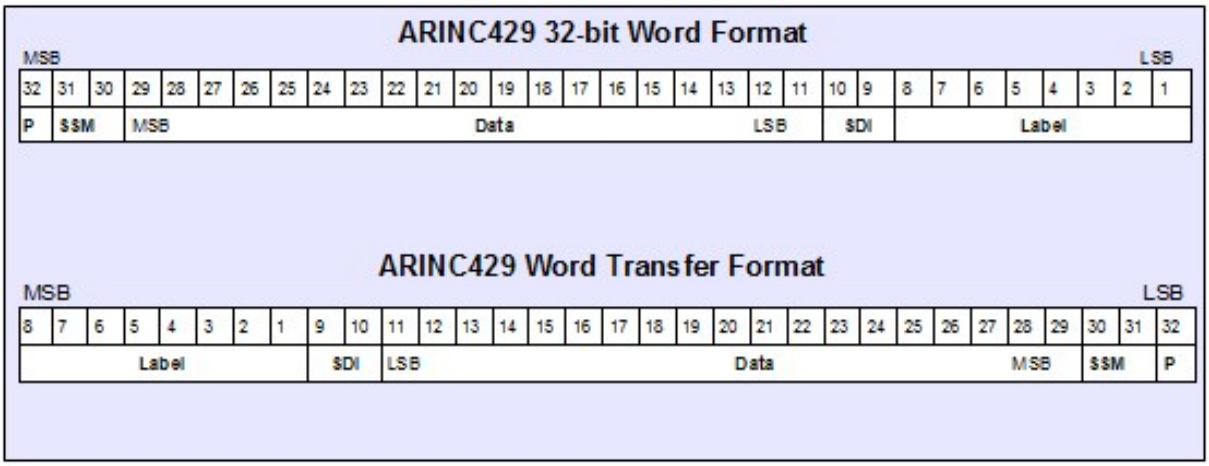
Data

ARINC429 defines bits 11-29 as those containing the word's Data information. Formatting of the Data bits, as well as the entire ARINC429 word, is very flexible.

When transmitting words on the ARINC bus, the Label is transmitted first, MSB first, followed by the rest of the bit field, LSB first as shown in Figure 4 Arinc 429 Data Word Transfer Format.

The Label is always transmitted first, in reverse order to rest of the ARINC word – a compensation for compatibility with legacy systems. The receiving LRU is responsible for data translation and regrouping

Figure 4: Arinc 429 Data Word Transfer Format



of bits into proper order.

The Data types available in ARINC429 are:

- Binary (BNR) – Transmitted in fractional two's complement notation
- Binary Coded Decimal (BCD) – Numerical subset of ISO Alphabet No. 5
- Discrete Data – Combination of BNR, BCD or individual bit representation
- Maintenance Data and Acknowledgement – Requires two-way communication
- Williamsburg/Buckhorn Protocol – A bit-oriented protocol for file transfers

2 General Programming

An AIM Arinc 429 board has some features that affect the whole board. For example the IRIG-B time encoder/decoder, the trigger lines or the discrete channels. But depending on its assembly the board has also up to 32 channels that may either be configured as arinc 429 transmitter or receiver, but not both. Some functions to show more information about a specific board and its channels are listed in Section 2.3 "Getting Information"

2.1 Initialization and Board Setup

The API S/W Library is capable of controlling AIM boards which are located on your computer or remotely on a network server that has the AIM Network Server (ANS) software installed. Initialization and shutdown commands required for your application depend on this configuration. This section will discuss both configurations and the function calls required to support each configuration.

Note:
The ANET429 device works like an ANS Server with a single board inserted. Please refer to the ANET429 User's Manual for details

The basic Library Administrative and System functions supporting initialization and shutdown include the following functions which should be issued in the order shown:

- **Api429LibInit** - first initialize the API S/W Library interface and find out how many AIM boards/-modules are on the local PC
- **Api429LibServerConnect** - returns the count of Arinc 429 boards (boardCount) on the remote computer and status for success/failure of the execution of the function.

  Note: do not call *Api429LibInit* after *Api429LibServerConnect*
- **board mapping functions (only VxWorks)** - the commands *AiPciScan*, *AiVmeExamineSlot* and/or *AiVme429MapModule* are used to map a board to the PCI or VME bus. The output of the command *AiVme429MapModule* is the board ID assigned to this board and shall be used as input parameter of *Api429BoardOpen*. If interrupts are used, the function *AiVmeInitGenericInterrupt* also has to be called at this point.
- **Api429BoardOpen** - the application interface to the target module must then be opened to establish connectivity between the application interface and the AIM board.

  Api429BoardOpen returns a handle which will be used to reference this AIM board for the rest of the program. If the AIM boards are located on a remote computer, the parameter ac_SrvName has to be set to the ServerName or Internet Protocol (IP) address. If addressing boards on the local machine, the parameter shall be set to "local".

  In VxWorks the input parameter board_id of *Api429BoardOpen* shall be set to the value returned by function *AiVme429MapModule*.

Upon successful execution of these functions, the AIM boards are ready to be commanded with all other API S/W Library function calls.

```
wBoardCountLocal = Api429LibInit(); /* Local board access */

if( useServer )
{
    /* Remote board access */
    retVal = Api429LibServerConnect(strServerName, &wBoardCountRemote);
}

retVal = Api429BoardOpen(API429_BOARD_1, strServerName, &boardHandle);

/* at this point one or more channels of the first board can be
 * initialized to either transmit or receive */
```

2.2 Board cleanup and program termination

- **Api429ChannelClear** - The Api429ChannelClear command cleans up the settings made for each channel.
- **Api429BoardReset** - The Api429BoardReset command resets the board to its initial state.
- **Api429BoardClose** - The Api429BoardClose function terminates the communication to the AIM board(s). Subsequent calls of driver-related functions after issuing an Api429BoardClose function call will result in an error code and should be avoided. To re-establish connection to the AIM board, the Api429BoardOpen function must be called.
- **Api429LibServerDisconnect** - If a connection to a remote PC has been established, the command Api429LibServerDisconnect function should be issued to disconnect the network connection to the remote PC.
- **Api429LibExit** - The Api429LibExit command terminates the communication to all local boards.

```
/* Clear the used channel.
 * It will be stopped and all used resources are freed
 */
retVal = Api429ChannelClear(boardHandle, channelId);

retVal = Api429BoardClose(boardHandle);

if( useServer )
    Api429LibServerDisconnect(strServerName);

Api429LibExit();
```

2.3 Getting Information about the AIM Board and the Configuration

Once you have initialized and opened the connection to the AIM board as described in the previous section, you can obtain the status of the configuration of the board and the software versions contained on your AIM board. The functions that provide this status are as follows:

- **Api429BoardNameGet** - Shows the name of the board.
- **Api429VersionGetAll** - Reads the version of all onboard components
- **Api429BoardInfoGet** - Retrieves basic information about the board
- **Api429ChannelInfoGet** - Returns basic information about the specified ARINC 429 channel
- **Api429BoardServerInfoGet** - Shows information abut a given ANS server

2.4 Board Time

The Arinc 429 boards internally track the time to be able to add time tags for example to the data in the receiver monitor buffer. By default the board uses its onboard time generator, but an external IRIG-B signal may also be used as time source.

- **Api429BoardTimeGet** - The Api429BoardTimeGet command shows the current time.
- **Api429BoardTimeSet** - This command sets the onboard time generator
- **Api429BoardTimeSourceGet** - This commands shows the the time source. It may be from the board internal time generator or from an external IRIG-B time source.
- **Api429BoardTimeSourceSet** - This commands sets the time source.

2.5 Event Handling

The board can be setup to raise interrupts on certain occassions. Based on these the device driver will throw events. A callback function can be registered to be able to react to these events. On VxWorks systems AiVmeInitGenericInterrupt has also to be called to provide the driver library with additional information required to handle the interrupt.

It is also possible to register a function for PnP events, like plugging in or removing an USB device.

Please note that the overall event latency and the jitter thereof are strongly dependent on the environment (like the host PC). If the event latency is important, a long time analysis of the complete system is strongly recommended. Please note that events via the USB bus and via ANS (network) have a significantly high event latency.

The following functions may be used:

- **Api429ChannelCallbackRegister** - registers a function to handle the events of a channel.

- **Api429ChannelCallbackUnregister** - unregisters an event handling function.
- **Api429LibPnpCallbackSet** - registers a callback function for PnP. To remove a registered callback function, call this function with parameter null.

```
/* user defined callback function that may used as input to
 * Api429ChannelCallbackRegister */
static void AI_CALL_CONV userCallbackFunction(AiUInt8 boardHandle,
                                              AiUInt8 channelId,
                                              enum api429_event_type type,
                                              struct api429_intr_loglist_entry* x_Info)
{
    /* the code to react on an event may be placed here.
     * parameter type should be used to identify the source of the event */
}

uw_RetVal = Api429ChannelCallbackRegister( boardHandle, channelId, 0,
                                                        userCallbackFunction );
```

The following list shows some channel based events and the function by which they can be enabled.

- **API429_EVENT_TX_HALT**, always enabled (provoked with Api429TxRepetitionCountSet)
- **API429_EVENT_TX_SKIP**, enabled with Api429TxXferCreate
- **API429_EVENT_TX_LABEL**, enabled with Api429TxXferCreate
- **API429_EVENT_TX_INDEX**, enabled with Api429TxXferCreate
- **API429_EVENT_RX_ANY_LABEL**, enabled with Api429RxLabelConfigure
- **API429_EVENT_RX_INDEX**, enabled with Api429RxLabelConfigure
- **API429_EVENT_RX_ERROR**, enabled with Api429RxLabelConfigure
- **API429_EVENT_FUNC_BLOCK**, enabled with Api429RmCreate
- **API429_EVENT_RM_TRIGGER**, enabled with Api429RmCreate
- **API429_EVENT_RM_BUFFER_FULL**, enabled with Api429RmCreate
- **API429_EVENT_RM_BUFFER_HALF_FULL**, enabled with Api429RmCreate
- **API429_EVENT_REPLAY_HALF_BUFFER**, enabled with Api429ReplayInit
- **API429_EVENT_REPLAY_STOP**, enabled with Api429ReplayInit
- **API429_EVENT_TX_FIFO**, always enabled (provoked with Api429TxFifoInterruptCreate)

3 Transmitter Channel

There are several methods to send data on a transmitter channel to the ARINC 429 bus. Each one designed for a different scenario. After a reset all channels are disabled and may be set to a transmission mode with the command *Api429TxInit*. Don't forget to start the channel with Api429ChannelStart after having set it up. The following command can be used for all transmitter channels

- **Api429TxInit** - initializes the channel in a given transmission mode.
- **Api429TxAmplitudeSet** - sets the amplitude of a given channel. (Not available on all boards)
- **Api429TxStatusGet** - gets some status information, like how many transfer were sent
- **Api429ChannelSpeedSet** - allows to change from high speed to low speed and back
- **Api429ChannelInfoGet** - gets some overall information about this channel
- **Api429ChannelStart** - starts the channel. Without it the channel stays inactive and will do nothing
- **Api429ChannelHalt** - renders the channel inactive

3.1 TX FIFO

This is the most simple method to send data. Initialize the channel in FIFO mode by using parameter API429_TX_MODE_FIFO for *Api429TxInit*, start it and write data to the ARINC 429 bus with little overhead in a fire and forget manner.

- **Api429TxFifoDataWordsWrite** - write instructions to the FIFO, so that some Arinc 429 data words will be sent to the bus.
- **Api429TxFifoWrite** - write some instructions into the FIFO that will be handled by the board. These entries may be created by functions like *Api429TxFifoDataWordCreate*, *Api429TxFifoDelayCreate* or *Api429TxFifoInterruptCreate*
- **Api429TxFifoStatusGet** - provides information about the current status of the FIFO.
- **Api429TxFifoSetup** - allows to change basic parameters of the FIFO, like the size.

Listing 1: Simple Tx Fifo

```
retVal = Api429TxInit(boardHandle, channelId, API429_TX_MODE_FIFO, 0);

retVal = Api429ChannelStart(boardHandle, channelId);

labels_to_send[0] = 0xF001;
labels_to_send[1] = 0xF002;
labels_to_send[2] = 0xF003;
labels_to_send[3] = 0xF004;
retVal = Api429TxFifoDataWordsWrite(boardHandle, channelId, 4, labels_to_send,
                                    AiTrue, &ulEntriesWritten);
```

Listing 2: More complex Tx Fifo

```
retVal = Api429TxInit(boardHandle, channelId, API429_TX_MODE_FIFO, 0);

retVal = Api429ChannelStart(boardHandle, channelId);

/* send four data words and after that trigger an interrupt event */
retVal = Api429TxFifoDataWordCreate(&fifo_input[0], 0xF001, 4, 0);
retVal = Api429TxFifoDataWordCreate(&fifo_input[1], 0xF002, 4, 0);
retVal = Api429TxFifoDataWordCreate(&fifo_input[2], 0xF003, 4, 0);
retVal = Api429TxFifoDataWordCreate(&fifo_input[3], 0xF004, 4, 0);
retVal = Api429TxFifoInterruptCreate(&fifo_input[4], 0);

retVal = Api429TxFifoWrite(boardHandle, channelId, 5, fifo_input, AiTrue,
                                                    &ulEntriesWritten);
```

3.2 Rate Oriented

This method is designed that individual transfers are setup once and repeatedly be sent by the board. There are also limited methods available to send some data in between the transfers with a repetition rate. For this mode the channel shall be initialized with parameter API429_TX_MODE_RATE_CONTROLLED for *Api429TxInit*. The following commands shall be used to set up the channel:

- **Api429TxXferCreate** - creates a transfer
- **Api429TxXferBufferWrite** - writes data to the transmit buffer of a transfer. Can be updated while the channel is active
- **Api429TxXferRateAdd** - sets the repetition rate of the transfer
- **Api429TxXferRateShow** - shows the real repetition rate of a transfer (may differ from what has been set, due to rounding)
- **Api429TxXferRateRemove** - clears the repetition rate, so that the transfer will not be sent
- **Api429TxPrepareFraming** - calculate the framing from all the repetition rates set. Is also automatically done within *Api429ChannelStart*
- **Api429TxStartOnTrigger** - instead of starting the channel immediately, a transmitter channel can also be started when an external trigger was received.
- **Api429TxAcycFrameCreate** - creates a frame that is to be sent on demand within an existing cyclic frame
- **Api429TxAcycFrameSend** - sends a previously created acyclic frame

```
retVal = Api429TxInit( boardHandle, channelId, API429_TX_MODE_RATE_CONTROLLED, 0 );
```

```
memset(&x_Xfer, 0, sizeof(x_Xfer));
x_Xfer.xfer_id     = FRM_1_ID;    /* any unused Id is possible */
x_Xfer.xfer_type   = API429_TX_LAB_XFER;
x_Xfer.buf_size    = 1;
x_Xfer.xfer_gap    = 4;
retVal = Api429TxXferCreate(boardHandle, channelId, &x_Xfer, &x_XferInfo);

aul_XferLWord[ 0 ] = 0x0000F000 + FRM_1_ID;
retVal = Api429TxXferBufferWrite(boardHandle, channelId, FRM_1_ID, 1,
                                 x_Xfer.buf_size, aul_XferLWord);

retVal = Api429TxXferRateAdd(boardHandle, channelId, FRM_1_ID, FRM_1_RATE);

retVal = Api429ChannelStart(boardHandle, channelId);
```

```
/* we assume that the framing was created and is already started
 * and we want to send an acyclic frame in between */

memset( &x_Xfer, 0, sizeof(x_Xfer));
x_Xfer.xfer_id     = 5;  /* any other unused ID is also possible */
x_Xfer.xfer_type   = API429_TX_LAB_XFER;
x_Xfer.buf_size    = 1;
x_Xfer.xfer_gap    = 4;
retVal = Api429TxXferCreate(boardHandle, channelId, &x_Xfer, &x_XferInfo);

aul_XferLWord[0] = 0x0000F001;
retVal = Api429TxXferBufferWrite(boardHandle, channelId, x_Xfer.xfer_id, 1,
x_Xfer.buf_size, aul_XferLWord);

aul_Xfers[ 0 ] = x_Xfer.xfer_id;
retVal = Api429TxAcycFrameCreate  ( boardHandle, channelId, 1, &aul_Xfers[0],
&AcycFrameID /* receive the frame id here */);

/* now the acyclic frame can be sent on demand */
retVal = Api429TxAcycFrameSend (boardHandle, channelId, AcycFrameID);
```

3.3 Framing

This method is like the rate oriented mode, but allows more control on the frames being set up. In the rate oriented mode a major frame and several minor frames are automatically set up to achieve the wanted repetition rates. By doing so you don't have direct control on the frames. With the framing mode (with parameter API429_TX_MODE_FRAMING in *Api429TxInit*) you can create a framing that more

suits your needs. Again some transfers may be sent in between with acyclic transmissions like discribed for the rate oriented mode. In this case the following commands shall be used:

- **Api429TxXferCreate** - creates a transfer
- **Api429TxXferBufferWrite** - writes data to the transmit buffer. Can be updated while the channel is active
- **Api429TxFrameTimeSet** - sets the time difference of the start of two consecutive minor frames
- **Api429TxMinorFrameCreate** - creates a minor frame, which may contain one or more transfers. You may include the same transfer several times if required.
- **Api429TxMajorFrameCreate** - creates the major frame, which may contain one or more minor frames. You may include the same minor frame several times if required.
- **Api429TxRepetitionCountSet** - In this mode you can tell the board to send the major frame only for a certain amount of times.
- **Api429TxStartOnTrigger** - instead of starting the channel immediately, a transmitter channel can also be started when an external trigger was received.

```
retVal = Api429TxInit(boardHandle, channelId, API429_TX_MODE_FRAMING, 0);

retVal = Api429TxFrameTimeSet(boardHandle, channelId, 10); /*sets frame time to 10ms*/

memset( &x_Xfer, 0, sizeof(x_Xfer));
x_Xfer.xfer_id    = 1;   /* any other unused ID is also possible */
x_Xfer.xfer_type  = API429_TX_LAB_XFER;
x_Xfer.buf_size   = 4;  /* often a buffer size 1 is also a good choice */
x_Xfer.xfer_gap   = 4;
retVal = Api429TxXferCreate(boardHandle, channelId, &x_Xfer, &x_XferInfo);

for( ul_Index=0; ul_Index<x_Xfer.buf_size; ul_Index++ )
    aul_XferLWord[ ul_Index ] = 0x0000F000 | (ul_Index+1);
retVal = Api429TxXferBufferWrite(boardHandle, channelId, x_Xfer.xfer_id, 1,
                                 x_Xfer.buf_size, aul_XferLWord);

aul_Xfers[ 0 ] = 1;

memset(&x_MFrame, 0, sizeof(x_MFrame));
x_MFrame.ul_FrmId    = 0;
x_MFrame.ul_XferCnt = 1;
x_MFrame.pul_Xfers   = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);

aul_Frames[0] = 0;  /* list of minor frames within the major frame */
retVal = Api429TxMajorFrameCreate(boardHandle, channelId, 1, aul_Frames);

retVal = Api429ChannelStart(boardHandle, channelId);
```

Example for complex framing

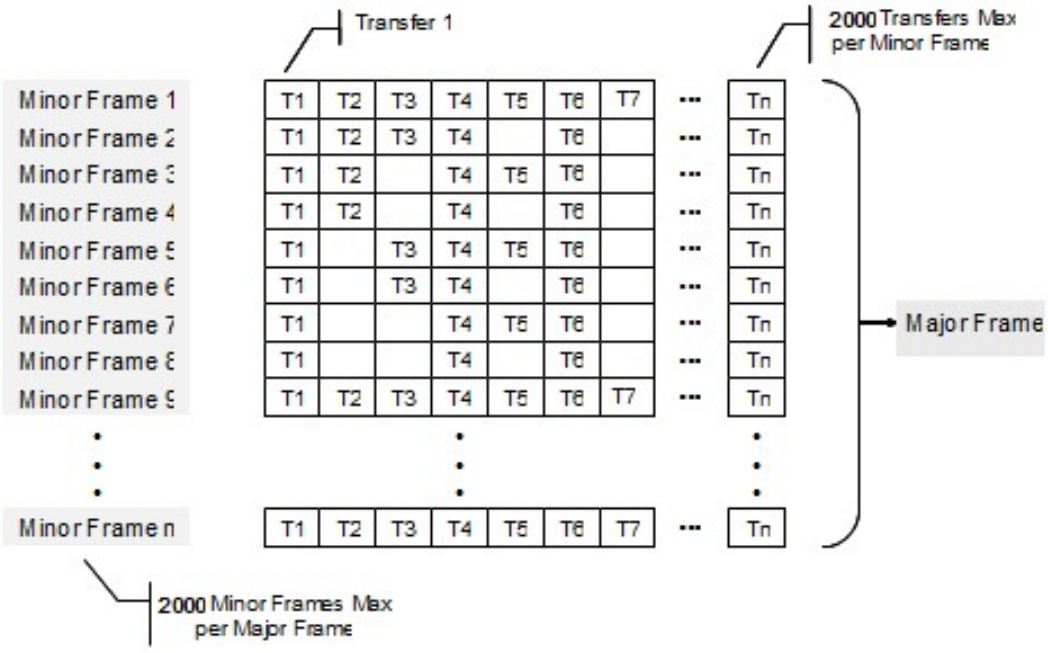
If a framing is more complex it may be hard to find out which minor frames are to be set up. In this case it can be helpful to draw the requirements into a figure.

Let us assume the following requirement:

- Transfer T1 has to be sent every 10ms
- Transfer T2 has to be sent 4 times with a time delta of 10ms, but have a pause of 40ms after that
- Transfer T3 has to be sent twice with a time delta of 10ms and have a pause of 20ms after that
- Transfer T4 has to be sent every 10ms
- Transfer T5 has to be sent every 20ms
- Transfer T6 has to be sent every 10ms
- Transfer T7 has to be sent every 80ms

This requirement allows us to divide it into chunks of 10ms each. Writing this down leads to the Figure 5 Complex Framing Sample

Figure 5: Complex Framing Sample



To finally set up this framing requires us to set the minor frame time to 10ms and to set up eight minor frames.

```
retVal = Api429TxInit(boardHandle, channelId, API429_TX_MODE_FRAMING, 0);
```

```
/* set up transfers here */

retVal = Api429TxFrameTimeSet(boardHandle, channelId, 10);

aul_Xfers[ 0 ] = T1;
aul_Xfers[ 1 ] = T2;
aul_Xfers[ 2 ] = T3;
aul_Xfers[ 3 ] = T4;
aul_Xfers[ 4 ] = T5;
aul_Xfers[ 5 ] = T6;
aul_Xfers[ 6 ] = T7;
x_MFrame.ul_FrmId   = 1;
x_MFrame.ul_XferCnt = 7;
x_MFrame.pul_Xfers  = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);

aul_Xfers[ 0 ] = T1;
aul_Xfers[ 1 ] = T2;
aul_Xfers[ 2 ] = T3;
aul_Xfers[ 3 ] = T4;
aul_Xfers[ 4 ] = T6;
x_MFrame.ul_FrmId   = 2;
x_MFrame.ul_XferCnt = 5;
x_MFrame.pul_Xfers  = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);

aul_Xfers[ 0 ] = T1;
aul_Xfers[ 1 ] = T2;
aul_Xfers[ 2 ] = T4;
aul_Xfers[ 3 ] = T5;
aul_Xfers[ 4 ] = T6;
x_MFrame.ul_FrmId   = 4;
x_MFrame.ul_XferCnt = 5;
x_MFrame.pul_Xfers  = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);

aul_Xfers[ 0 ] = T1;
aul_Xfers[ 1 ] = T2;
aul_Xfers[ 2 ] = T4;
aul_Xfers[ 3 ] = T6;
x_MFrame.ul_FrmId   = 4;
x_MFrame.ul_XferCnt = 4;
```

```
x_MFrame.pul_Xfers   = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);
```

```
aul_Xfers[ 0 ] = T1;
aul_Xfers[ 1 ] = T3;
aul_Xfers[ 2 ] = T4;
aul_Xfers[ 3 ] = T5;
aul_Xfers[ 4 ] = T6;
x_MFrame.ul_FrmId    = 5;
x_MFrame.ul_XferCnt = 5;
x_MFrame.pul_Xfers   = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);
```

```
aul_Xfers[ 0 ] = T1;
aul_Xfers[ 1 ] = T3;
aul_Xfers[ 2 ] = T4;
aul_Xfers[ 3 ] = T6;
x_MFrame.ul_FrmId    = 6;
x_MFrame.ul_XferCnt = 4;
x_MFrame.pul_Xfers   = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);
```

```
aul_Xfers[ 0 ] = T1;
aul_Xfers[ 1 ] = T4;
aul_Xfers[ 2 ] = T5;
aul_Xfers[ 3 ] = T6;
x_MFrame.ul_FrmId    = 7;
x_MFrame.ul_XferCnt = 4;
x_MFrame.pul_Xfers   = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);
```

```
aul_Xfers[ 0 ] = T1;
aul_Xfers[ 1 ] = T4;
aul_Xfers[ 2 ] = T6;
x_MFrame.ul_FrmId    = 8;
x_MFrame.ul_XferCnt = 3;
x_MFrame.pul_Xfers   = &aul_Xfers[0]; /* list of transfers within this minor frame */
retVal = Api429TxMinorFrameCreate(boardHandle, channelId, &x_MFrame, &x_MFrameInfo);
```

```
aul_Frames[0] = 1;  /* list of minor frames within the major frame */
aul_Frames[1] = 2;
aul_Frames[2] = 3;
aul_Frames[3] = 4;
aul_Frames[4] = 5;
aul_Frames[5] = 6;
aul_Frames[6] = 7;
aul_Frames[7] = 8;
retVal = Api429TxMajorFrameCreate(boardHandle, channelId, 1, aul_Frames);

retVal = Api429ChannelStart(boardHandle, channelId);
```

3.4 Framing with Dyntag

This mode is nearly identical to the framing mode. The only difference is that the use of function *Api429TxXferDyntagAssign* is allowed in this mode and the buffer size of transfers is limited to 1.

This mode is enabled by calling *Api429TxInit* with mode API429_TX_MODE_FRAMING_DYNTAG.

3.5 Replay

The physical replay mode allows to retransmit data that was previously recorded. It has to written in the format as recorded during the Monitor Operation to the replay buffer and can be updated in a half buffer mechanism.

In general, the order in which you will need to setup your Replay Configuration using the Replay functions is as follows:

1. **Replay Initialization** – For a transmitter channel, *Api429ReplayInit* has to be called to set up the replay buffer and for some additional options
2. **Getting Status Information** - Get the Replay Buffer start address and size of half the Replay buffer using *Api429ReplayStatusGet*
3. **Initial Data Write** - Use *Api429ReplayDataWrite* to fill half of the Replay buffer with replay data using the Replay Buffer start address and Replay Buffer half size (max). To fill the next half of the Replay buffer before starting the transmitter, repeat the above to get the start address of the next half of the Replay buffer and issue *Api429CmdWriteRepData* again.
4. **Channel Start** - Start transmitting the Replay data using *Api429ChannelStart*
5. **Update Data** - If your application requires the Replay buffer to be refilled, then be sure to program a Half Buffer interrupt with function *Api429ReplayInit*. Once half of the replay buffer (128k entries) has been transmitted, your interrupt handler can then refill the appropriate half of the Replay buffer

with new replay data using *Api429ReplayDataWrite*. To get the location and size of the half buffer that should be refilled you can use the Replay Next Buffer Pointer provided in Log List Entry Word LLB when you receive the interrupt. If you are polling to determine when to refill you can use *Api429ReplayStatusGet* (ul_StartAddr and ul_Size in structure api429_replay_status) to obtain the location and size of the half buffer that should be refilled.

For cyclic transmission of the Replay buffer (setup in *Api429ReplayInit*), transmission of the Replay buffer will continue until an *Api429ChannelHalt* function call is issued.

For single transmission of the Replay buffer (setup in *Api429ReplayInit*), once the end of the Replay data is reached, transmission is stopped and a stop interrupt will be issued (if enabled in *Api429ReplayInit*).

```
retVal = Api429TxInit(boardHandle, channelId, API429_TX_MODE_PHYS_REPLAY, 0);

retVal = Api429ReplayInit(boardHandle,
            channelId,
            0,              // uc_ClrEntryBit,
            0,              // uc_NoRepCnt,
            1,              // uc_CycOpr,       --> cyclically replay
            1,              // uc_RepErrors,    --> also replay errors
            0x01,           // uc_RepIntMode,   --> raise interrupts on half buffers
            0,              // uc_AbsLongTTag,
            0,              // uw_DayOfYear,
            0,              // ul_Min,
            0,              // ul_MSec,
            ul_FileSize);


// Status Read to get Size and load address
retVal = Api429ReplayStatusGet(boardHandle, channelId, &x_RepStatus);
// write first Half Buffer
retVal = Api429ReplayDataWrite(boardHandle, channelId, &x_RepStatus, p_DataBuffer,
                                                                    &ul_BytesWritten);

read_new_data_into_p_DataBuffer(p_DataBuffer);


// Status Read to the next load address
retVal = Api429ReplayStatusGet(boardHandle, channelId, &x_RepStatus);
// write second Half Buffer
retVal = Api429ReplayDataWrite(boardHandle, channelId, &x_RepStatus, p_DataBuffer,
                                                                    &ul_BytesWritten);

/* Start the replay */
Api429ChannelStart(boardHandle, channelId);


/* Read initial value for ul_OldRpiCnt */
```

```
retVal = Api429ReplayStatusGet(boardHandle, channelId, &x_RepStatus);

while ((API429_BUSY == x_RepStatus.uc_Status))
{
    if( ul_RpiCnt_was_incremented(x_RepStatus)  )
    {
        read_new_data_into_p_DataBuffer(p_DataBuffer);

        /* write next half buffer */
        retVal = Api429ReplayDataWrite(boardHandle, channelId, &x_RepStatus,
                                       p_DataBuffer, &ul_BytesWritten);
    }

    AiOsSleep(100);

    retVal = Api429ReplayStatusGet(boardHandle, channelId, &x_RepStatus);
} // End While
```

4 Receiver Channel

To receive data, the channel has to be configured to be a receiver channel first. This is done with the *Api429RxInit* command. You can configure the channel to write the received data to either a seperate buffer for each label or into a shared monitor buffer. After that you have to activate the channel with *Api429ChannelStart*.

- **Api429RxInit** - initializes the channel so that is able to receive data. Any previous configuration for this channel is cleared.
- **Api429RxStatusGet** - gets some general status information about a receiver channel, like the number of data received, or if the channel is started or not.
- **Api429ChannelSpeedSet** - allows to change from high speed to low speed and back
- **Api429ChannelInfoGet** - gets some overall information about this channel
- **Api429ChannelStart** - starts the channel. Without it the channel stays inactive and will do nothing
- **Api429ChannelHalt** - renders the channel inactive

4.1 Selective Receiving

For each label an individual receiver buffer can be created. Once created the data received for this label can be read from this buffer. It is also possible to trigger an event on data reception.

- **Api429RxLabelConfigure** - creates an individual label receive buffer. Receiver Interrupts may be enabled/disabled here.
- **Api429RxLabelBufferWrite** - writes data to the label receive buffer. May be used to initialize or clear the buffer.
- **Api429RxLabelBufferRead** - reads data from the label receive buffer
- **Api429RxLabelStatusGet** - provides some status information for this label
- **Api429RxLabelBufferOffsetGet** - shows the offset to the start of the global memory. This may be used to read the buffer contents directly (for example with Api429BoardMemBlockRead)

```
retVal = Api429RxInit(boardHandle, channelId, AiFalse, AiFalse);

/* loop over all labels 0..255 --> 256 possible lables */
for(ul_LabIndex = 0; ul_LabIndex < 256; ul_LabIndex++)
{
    memset(&label_setup, 0, sizeof(label_setup));
    label_setup.label   = ul_LabIndex;
    label_setup.con     = API429_ENA;
    label_setup.bufSize = 1;
```

```
    retVal = Api429RxLabelConfigure(boardHandle, channelId, &label_setup,
                                                    &uc_Status, &ul_FreeMem);
}

retVal = Api429ChannelStart(boardHandle, channelId);

...

/* let's have a look at all received data */
retVal = Api429RxStatusGet(boardHandle, channelId, &b_RxStatus, &l_MsgCnt, &l_ErrCnt);

/* let's see, what happened on label 5 */
retVal = Api429RxLabelStatusGet(boardHandle, channelId, 5, 0, 0,
                                            pw_LabIx, px_LabCnt, px_LabErr );

/* and now let's read the current buffer contents */
retVal = Api429RxLabelBufferRead(boardHandle, channelId, 5, 0/*no SDI*/, 1,
                                                    &x_Ctl, ax_LData);
```

4.2 Receiver Monitor

In addition to the individual receive label buffers, you can also enable receiver monitoring. All received data will be written into a shared buffer, along with a time tag and some additional information. This allows to read all data of a channel with a single command. Usefull functions for the receiver monitor are:

- **Api429RmCreate** - enables receiver monitoring
- **Api429RmLabelConfigure** - allows to enable/disable monitoring for each label, allowing a basic filter.
- **Api429RmTriggerConfigSet** - allows to start monitoring on a specific event
- **Api429RmDataRead** - reads data out of the receiver monitor buffer. This function should be repeatedly called.
- **Api429RmFuncBlockConfigure** - a function block has several parameters that define the conditions that have to be met for it to become active. Other parameters describe the actions that are executed once the conditions are met. Actions include raising an event, generating an output strobe and filtering the RM buffer.

```
retVal = Api429RxInit(boardHandle, channelId, AiFalse, AiFalse);

setup.mode = API429_RM_MODE_LOC;
setup.size_in_entries = API429_RM_MON_DEFAULT_SIZE;
setup.tat_count = API429_RM_CONTINUOUS_CAPTURE;
```

```
setup.interrupt_mode = API429_RM_IR_DIS;
retVal = Api429RmCreate(boardHandle, channelId, &setup);

/* enable all labels...
 * loop over all labels 0..255 = 256 possible labels
 * we assume sdi sorting is disabled */
for (ul_LabIndex = 0; ul_LabIndex <= 255; ul_LabIndex++)
{
    label_config.label_id = ul_LabIndex;
    label_config.sdi = 0;  /* note: sdi sorting is not enabled here */
    label_config.enable = AiTrue;
    retVal = Api429RmLabelConfigure(boardHandle, channelId, &label_config);
}

retVal = Api429ChannelStart(boardHandle, channelId);

...
/* let's have a look at all received data */
retVal = Api429RxStatusGet(boardHandle, channelId, &b_RxStatus, &l_MsgCnt, &l_ErrCnt);

/* and read some data. This is the oldest data after channel start
 * if 32 entries were read (uw_Count == 32), there is probably newer
 * data still available. */
retVal = Api429RmDataRead(boardHandle, channelId, 32, &uw_Count, ax_SData);

for (i = 0; i < uw_Count; i++) {
    printf("  %08X  %02d:%02d:%02d.%06d  %d  %d\r\n",
      ax_SData[i].ldata,
      ax_SData[i].brw.b.hours, ax_SData[i].tm_tag.b.minutes,
      ax_SData[i].tm_tag.b.seconds, ax_SData[i].tm_tag.b.microseconds,
      ax_SData[i].brw.b.channel + 1,
      ax_SData[i].brw.b.e_type);
}
```

5 Automatic data processing

5.1 Loop/Pollution

It is possible to link a receiver channel to a transmitter channel, so that all incomming data is forwarded. If required the forwarded data can be modified ("polluted"). Api429CmdTxInit has to be called with API429_TX_MODE_LOOP. All other set up has to be done at the receiver channel.

Note:
boards with 32 channels are internally handled as two boards, whith 16 channels each.

The incoming data is forwarded on low level. It may be modified (polluted) on a small scale, if wanted. The following commands should be used in this case:

- **Api429RxDataLoopAssign** - After calling this function, all data received for enabled labels will be forwarded to the specified transmitter channel and sent on this channel at once.
- **Api429RxPollutionConfigure** - Used to define data modifications in a receiver/transmitter data loop

```
retVal = Api429RxInit(boardHandle, rxChannelId, AiFalse, AiFalse);

retVal = Api429TxInit(boardHandle, txChannelId, API429_TX_MODE_LOOP, AiFalse);

retVal = Api429RxDataLoopAssign(boardHandle, rxChannelId, txChannelId);

/* set up a pollution block that adds 0x100 to each arinc 429 data word */
memset(&pollBlk, 0, sizeof(pollBlk));
pollBlk.pb_id = 1;
pollBlk.and_mask = 0xFFFFFFFF; /* don't modify the data with AND */
pollBlk.addsub_val = 0x100; /* add 0x100 to the looped data */

for(labelId=0; labelId<256;labelId++)
  retVal = Api429RxPollutionConfigure(boardHandle, rxChannelId, labelId,
                                      0/*sdi*/, AiTrue, &pollBlk);

retVal = Api429ChannelStart(boardHandle, rxChannelId);

retVal = Api429ChannelStart(boardHandle, txChannelId);
```

5.2 Frame Response

It is also possible to react on a normal receiver channel to a specific data word, where based on the contents of the received data word a previously prepared acyclic frames will automatically be sent.

- **Api429RxFrameResponseAssign** - Attach an automatic response on a transmitter channel to a receive label
- **Api429RxFrameResponseRelease** - Used to clear the automatic response

```
retVal = Api429RxInit(boardHandle, rxChannelId, AiFalse, AiFalse);

retVal = Api429TxInit(boardHandle, txChannelId, API429_TX_MODE_RATE_CONTROLLED,
                                                                    AiFalse);



memset( &x_Xfer, 0, sizeof(x_Xfer));
x_Xfer.xfer_id    = 5;  /* any other unused ID is also possible */
x_Xfer.xfer_type  = API429_TX_LAB_XFER;
x_Xfer.buf_size   = 1;
x_Xfer.xfer_gap   = 4;
retVal = Api429TxXferCreate(boardHandle, txChannelId, &x_Xfer, &x_XferInfo);

aul_Xfers[ 0 ] = x_Xfer.xfer_id;
retVal = Api429TxAcycFrameCreate  ( boardHandle, txChannelId, 1, &aul_Xfers[0],
                                    &AcycFrameID /* receive the frame id here */);



/* send an acyclic transfer, if the lowest "data byte" is 0x01 */
memset(&p_Setup, 0, sizeof(p_Setup));
p_Setup.tx_channel = txChannelId;
p_Setup.tx_acyc_frame_id = AcycFrameID;
p_Setup.compare_mask = 0x100;
p_Setup.compare_value = 0xFF00;

for(labelId=0; labelId<256;labelId++)
    retVal = Api429RxFrameResponseAssign(boardHandle, rxChannelId, labelId, 0,
                                                                    &p_Setup );

retVal = Api429ChannelStart(boardHandle, rxChannelId);

retVal = Api429ChannelStart(boardHandle, txChannelId);
```

6 Troubleshooting

This section is designed to help in case problems appear. Some of these points may seem obvious, but some can easily be overlooked and might be helpfull.

6.1 Checking Return Values

For every function call the return value should be checked. Most functions will return zero in case of success. In case of a different return value it should be used as input parameter for function *Api429LibErrorDescGet* to provide a human readable error description.

6.2 Checking the Channel

In case a channel is not doing what is expected, check the channel status with the command *Api429ChannelInfoGet*. It will show the channel speed and whether the channel is active or not.

6.3 Checking the Transmitter

If you don't receive the correct data, you should check the transmit counters. Is the content of output parameter pl_GlbCnt of function *Api429TxStatusGet* sensible or not?

6.4 Checking the Receiver

If no data is received in a label buffer, it is helpfull to check if the data was received at all. Is the content of the output parameters px_MsgCnt and px_ErrCnt sensible? Does it match what was sent by the transmitter channel (using *Api429TxStatusGet*)

6.5 Checking the Cabling

Are TRUE and CMPL connected between the transmitter and the receiver? If only one connection is drawn it may lead to undefined behaviour.

6.6 Monitoring the Bus Traffic

Either enable the monitoring of the receiver channel or attach a seperate board to the ARINC 429 bus. This may be a different board, for example with the PBA.pro running on it. Understanding the traffic on the ARINC 429 bus can help to understand the nature of the problem.

6.7 Contacting Support

Collect data about the board. What is the correct name of the board, which are the versions on board? The output of *Api429BoardInfoGet* and *Api429VersionGetAll* should always be provided in support requests to speed up the problem determination.

Support either be addressed with the technical support form on www.aim-online.com or be sent via email to support@aim-online.com

7 Frequently Asked Questions

7.1 How to find a Transfer ID, Minor Frame ID or Function Block ID

These Ids can freely be choosen within the valid range, shown in the description of the function. Any of them will do, as long as it was not used before.

If I choose an ID that is already in use for an item, its contents will be overwritten. This means that all references to that item will use the last configuration.

7.2 How to clean up

For many items there is a deletion function (like *Api429TxXferDelete*). This will reset this item to its initial state. A channel can be reverted with the command *Api429ChannelClear* and the whole board with the command *Api429BoardReset*.

Resetting the board with *Api429BoardReset* implicitely resets all channels. Clearing a channel implicitely resets all of its items (like minor frames, transfers and so on).

7.3 Why is only a part of my transfer buffer sent?

When setting up a transfer with *Api429TxXferCreate* you can refer to it - for example within a minor frame using parameter *px_MFrame->pul_Xfers* of command *Api429TxMinorFrameCreate*. Each time the transfer is handled by the framing one ARINC429 data word is being sent on the bus.

When creating a transfer with *Api429TxXferCreate* and parameter *buf_size* greater than one, a ring buffer is set up. Each time that transfer is handled by the framing, one entry (== ARINC429 data word) of this ring buffer is handled by the framing.

If you want to send the complete buffer contents within one minor frame, you have to refer the transfer repeatedly within a minor frame - once for each entry of the ring buffer.