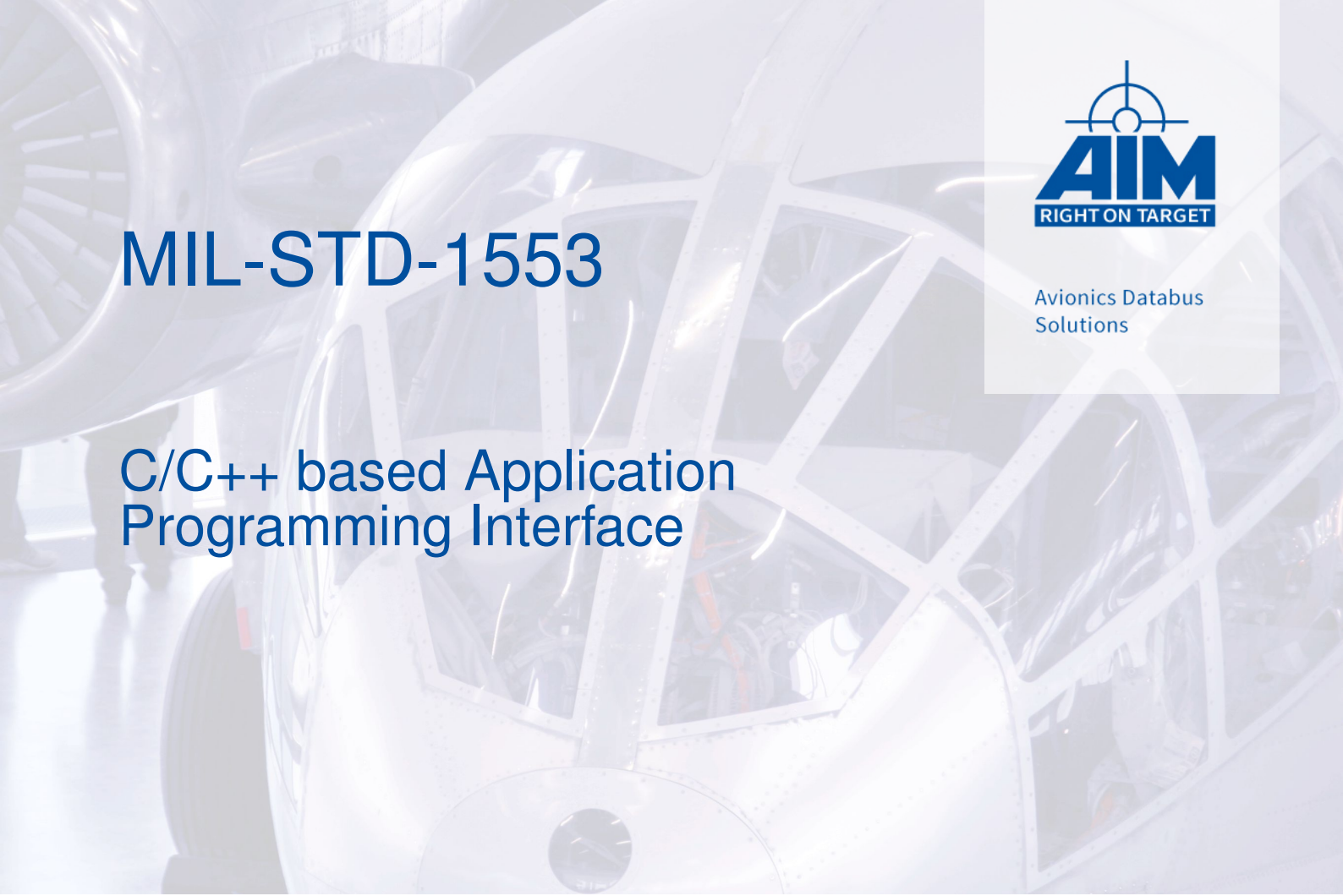


MIL-STD-1553

C/C++ based Application Programming Interface

AIM
RIGHT ON TARGET

Avionics Databus Solutions

Programmer's Guide

Version 24.22.0
December, 2024

www.aim-online.com

MIL-STD-1553

C/C++ based Application Programming Interface

Programmer's Guide

Version 24.22.0
December, 2024

AIM NO. 60-11900-37-24.22.0

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
sales@aim-online.us

TABLE OF CONTENTS

1 Introduction

1.1 General

Welcome to the Programmer's Guide for MIL-STD-1553 for AIM devices. This programmer's guide, in conjunction with the associated Reference Manual, is intended to provide the software (s/w) programmer with the information needed to develop a host computer application interface to AIM devices. The Programmer's Guide provides the application developer with high-level s/w development information including high level system design information, user application system design concepts, function call guidelines and sample programs for the AIM bus interface card. The Software Library Reference Manual provides the user with detailed programming information including s/w library function call and header file details and specific troubleshooting information. The BSP Getting Started manual contains information about the board support package (BSP) contents.

Note:
Please note that not all functionality explained in this document is available for all AIM modules. Please see the Appendix B of the Reference Manual for details on which limitations apply to different board variants

1.2 How this Programmer's Guide is organized

The first sections describe the information needed to start with programming the AIM device and contain background information:

1. **Introduction** - Provides an introduction to the contents of the programmer's guide documentation conventions and applicable documents.

2. **MIL-STD-1553 Overview** - Provides a high level overview of the MIL STD 1553 communication protocol including Command, Data and Status word formats and message sequencing.

3. **AIM1553 Module Overview** - Provides a high level overview of the hardware and software architecture. Included in the software section is a description of an application program structure overview and basics about 1553 message transfers and minor/major framedefinition.

The middle sections describe how the AIM library is interfaced in order to program an application:

4 **Library Administration And System Programming** - Provides the programming guidelines for the administration and system functions.

5 **Simulation Buffer Programming** - Provides details on how data buffers can be assigned to BC and RT functions.

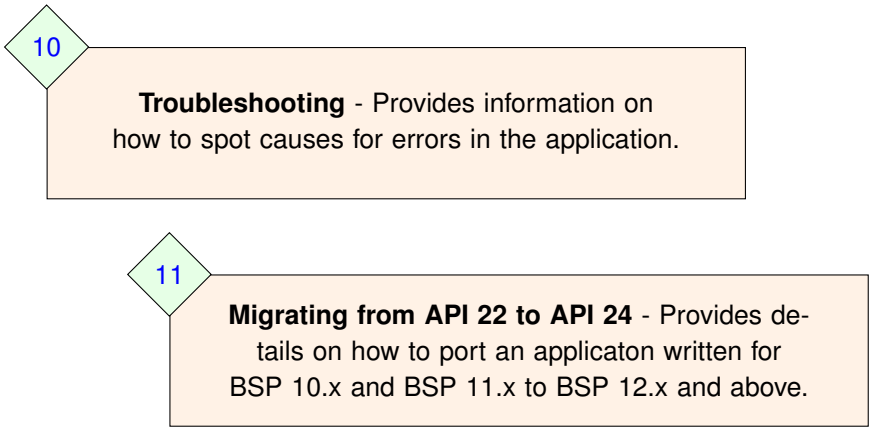
6 **Bus Controller Programming** - Provides details on how to send transfers on the bus.

7 **Remote Terminal Programming** - Provides details on how to implement different types of Remote Terminals.

8 **Bus Monitor Programming** - Provides details on how to implement a Bus Monitor application.

9 **Replay Programming** - Provides details on how to implement a physical bus replay.

The last sections wrap up the document by providing troubleshooting information and a migration guide



1.3 AIM Document Family

AIM has developed several documents that may be used to aid the developer with other aspects involving the use of the 1553 bus interface card. These documents and a summary of their contents are listed below:

- **MIL-STD-1553 Reference Manual** - provides the 1553 application developer with detailed programming information including library function call and specific troubleshooting information. This guide is to be used in conjunction with this Programmer's Guide.
- **MIL-STD-1553 Tutorial** - provides a general overview of MIL-STD-1553 including MIL-STD-1553 history and a complete annotated version of the MIL-STD-1553B specification and an interpretation of the specification contents.
- **Getting Started Manual** - assists the first time users of the AIM 1553 boards with software installation, hardware setup and starting a sample project. The different BSPs for the supported Operating Systems contain dedicated Getting Started Manuals for each OS.
- **Hardware Manual** - assists users with installation and initial setup of the device in the system. Each hardware has its dedicated Hardware Manual.
- **Users Manual** - the AIM ANET device family has User Manuals which is a combination of the Hardware Manual and Getting Started Manual.

2 MIL-STD-1553 Specification Overview

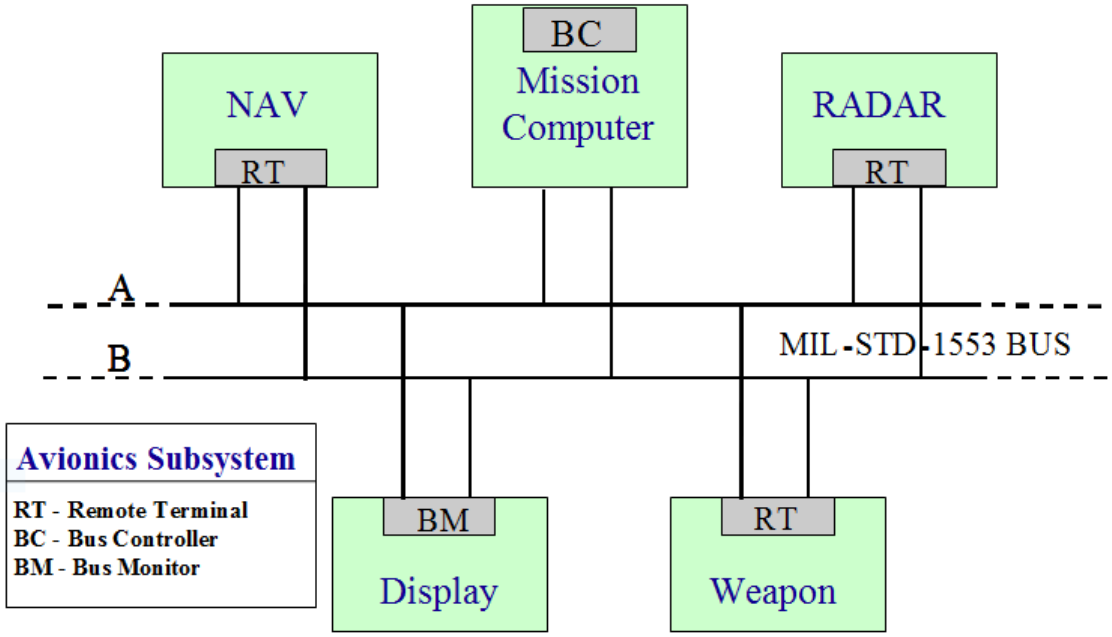
For any 1553 application, the user will define the requirements for the terminal under design.

Terminal device types include:

- **Bus Controller**
  - Manages all message traffic on the bus in a Command-Response fashion
  - Only one bus controller is allowed, provisions for a backup BC to take control if needed, but only one at a time
  - Executes scheduled messages on the bus in a pre-determined priority scheme, utilizing frame and sub-frame time scheduling
- **Remote Terminal**
  - Up to 32 RT devices, each with a unique individual address
  - Respond only when requested by the BC
  - Responds back to every BC command
- **Bus Monitor**
  - No address, no transmission on the bus

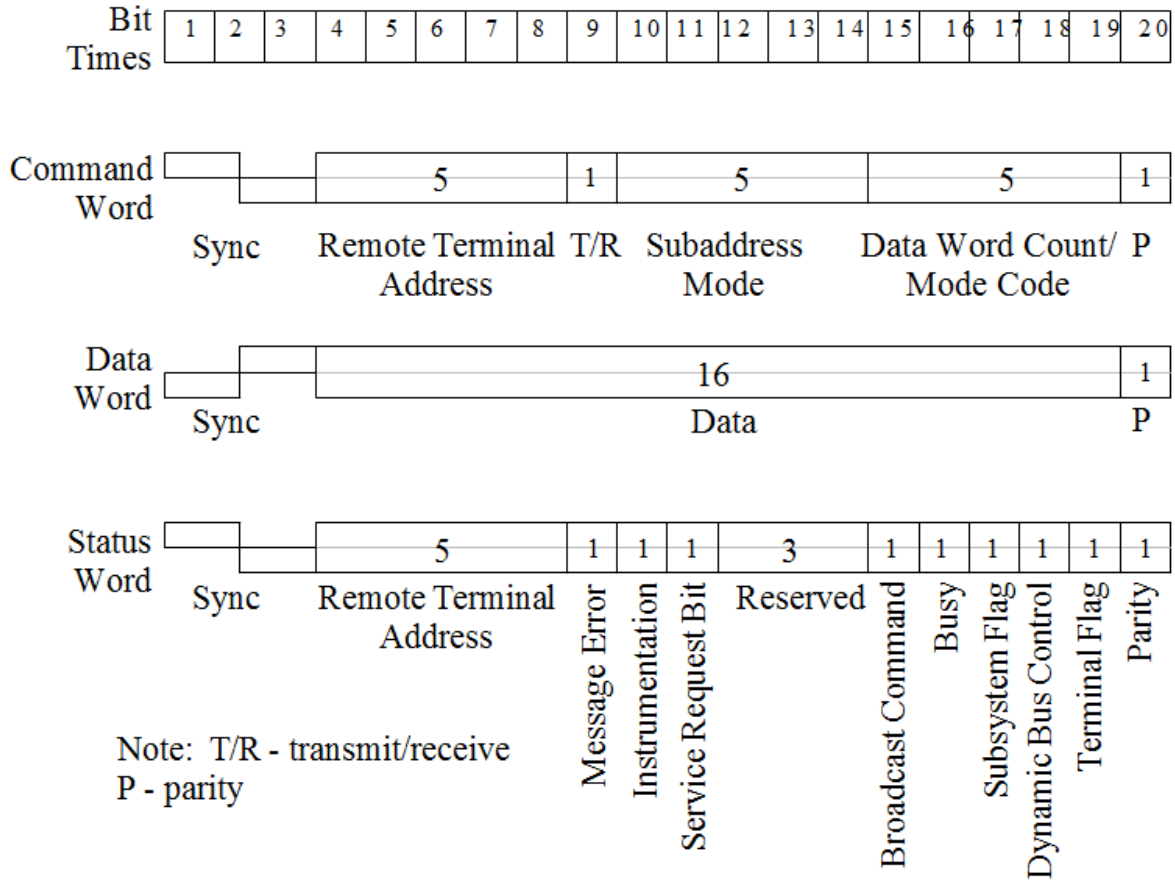
Figure 2.0.0-I shows an example of a bus configuration.

Figure 2.0.0-I: Example Bus Configuration



The user's application will likely be required to adhere to an interface control document which will define the data interfaces between the RT and BC. To insure common terminology and understanding of the API programming design information contained in the following sections, we will quickly review the basic concepts of the MIL-STD-1553 message content and transfer protocol. Message traffic on the 1553 bus consists of command, data, and status words with the format shown in Figure 2.0.0-II.

Figure 2.0.0-II: Word Formats



MIL-STD-1553 defines 10 types of transfers as shown in Figure 2.0.0-III and 2.0.0-IV. Each transfer is composed of control words (command and status) and data words and is always initiated by the BC. Normal communications start with a command from the BC, transmitted to a selected RT address. The RT receives or transmits data - depending on the BC command - and transmits a status word response. The BC can also initiate an information transfer between two RTs by issuing one a transmit command and the other a receive command.

Mode commands are used in terminal control and systems checks. Mode commands may or may not involve the transfer of a data word.

Broadcast commands are commands sent to multiple RTs at once. Although restricted under Notice 1, Notice 2 allows broadcast command transmission but only a limited set. The RT is responsible for distinguishing between broadcast and non-broadcast command messages.

Data transmission on the bus between terminal devices is performed in a deterministic manner with no more or less than a 4-12 µsec response intermessage gap allowed.

Figure 2.0.0-III: Information Transfer Formats

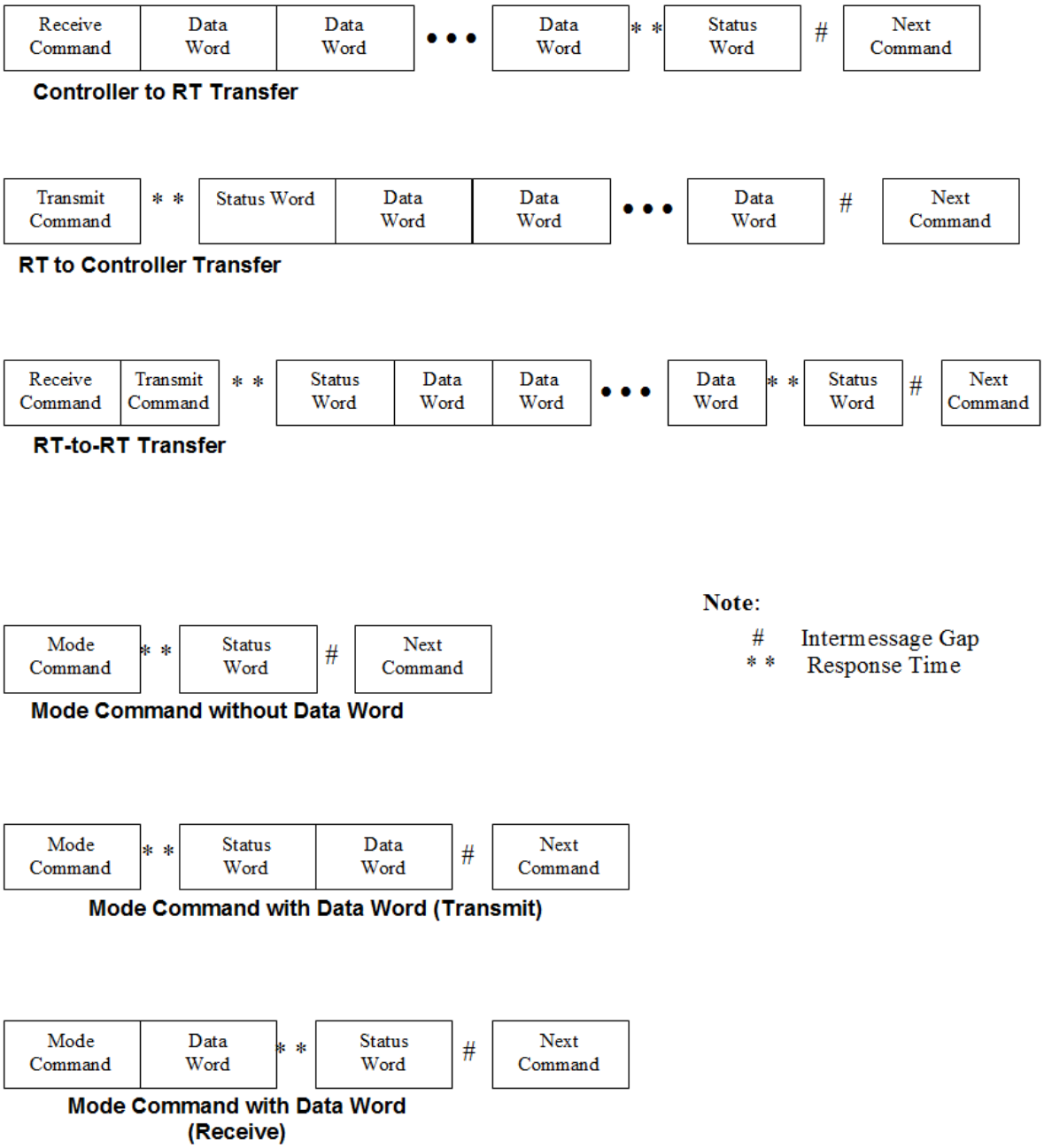
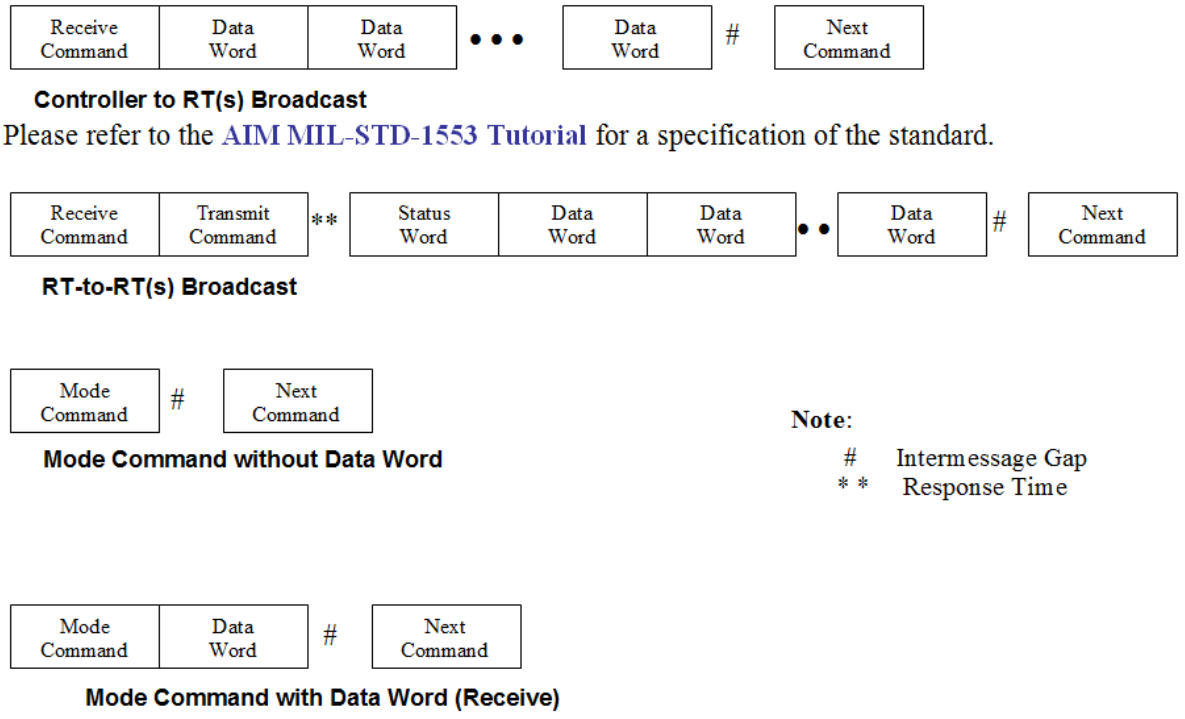


Figure 2.0.0-IV: Broadcast Information Transfer Formats



3 AIM 1553 Bus Interface Module Overview

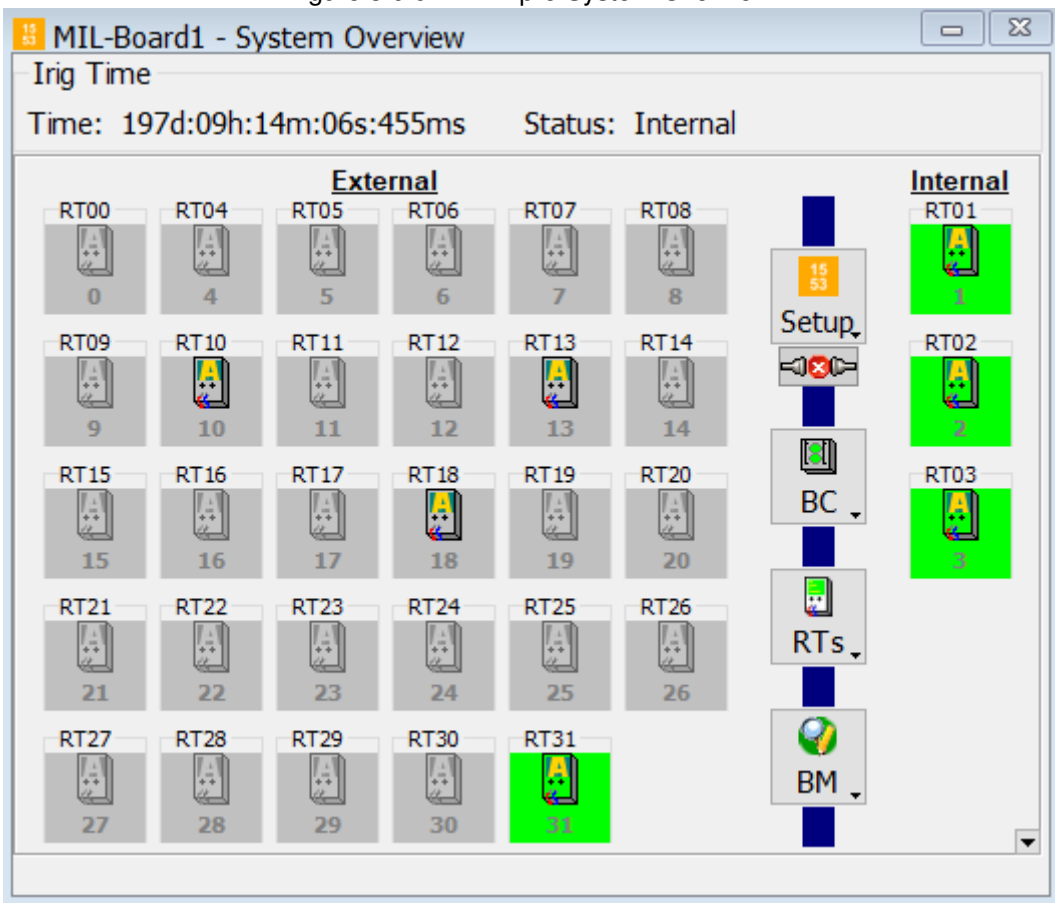
The standard AIM 1553 modules are part of a family of bus interface modules providing full function test, simulation, monitoring and databus analyzer functions for MIL-STD-1553A/B applications. The embedded AIM 1553 modules have a reduced function set and are optimized for embedded use cases. Additionally AIM offers standard variants with reduced function sets. Each module might provide the following functions on one or more dual redundant MIL-STD-1553 A/B buses.

- **Bus Controller** - provides real-time Bus Controller functions concurrently with Multiple RT and Bus Monitor operation. Key features include:
  - Autonomous operation including sequencing of minor/major frames
  - Support for acyclic message insertion/deletion
  - Programmable BC Retry without host interaction
  - Full error injection down to word and bit level (AS4112 Compliant) *
  - Multi-buffering for Data Consistency and Message Multiplexing
  - Synchronization of BC operation to external trigger inputs
  - 4 µs intermessage gaps *
- **Multiple Remote Terminal** - Simulates up to 32 Remote Terminals including all subaddresses. Key features include:
  - Programmable RT response time down to 4µs for each simulated RT *
  - Programmable and intelligent response to mode codes
  - Full error injection down to word and bit level (AS4112 Compliant) *
  - Multi-buffering with real-time data buffer updates
  - Programmable response time out
- **Chronological Bus Monitor** - Provides accurate time tagging of all bus traffic to 1 µs resolution including response time and gap time measurements down to 0.25 µs resolution. Key features include:
  - 100% data capture on two streams at full bus rates
  - Autonomous message synchronization and full error detection
  - Two dynamic complex trigger sequences *
  - Message filter and selective capture
  - Bus activity recording independent from trigger and capture mode
- **Bus Replay** * - Key features include:
  - Reconstruction of previously recorded MIL-STD-1553A/B databus traffic
  - Recorded data selectively replayed with any or all RT responses enabled
- **IRIG-B** time code decoder - Key features include:
  - Allows synchronization of MIL-STD-1553A/B bus traffic using one common IRIG-B time source or the on-board Time Code Generator

Note:
* Check Appendix B of the Reference Manual for a detailed overview of functions supported for different devices.

Figure 3.0.0-I shows a sample configuration of an AIM 1553 device with 32 RTs, a BC, BM and Replay functionality. This visual graphic was generated by the PBA.pro software package which runs on a host and controls an AIM 1553 bus interface. See the PBA.pro Bus Analyzer Getting Started Manual for further information about the AIM analyzer software.

Figure 3.0.0-I: PBA.pro System Overview



Note:
The PBA.pro software packages also utilize the API S/W Library to setup and control the AIM bus interface modules.

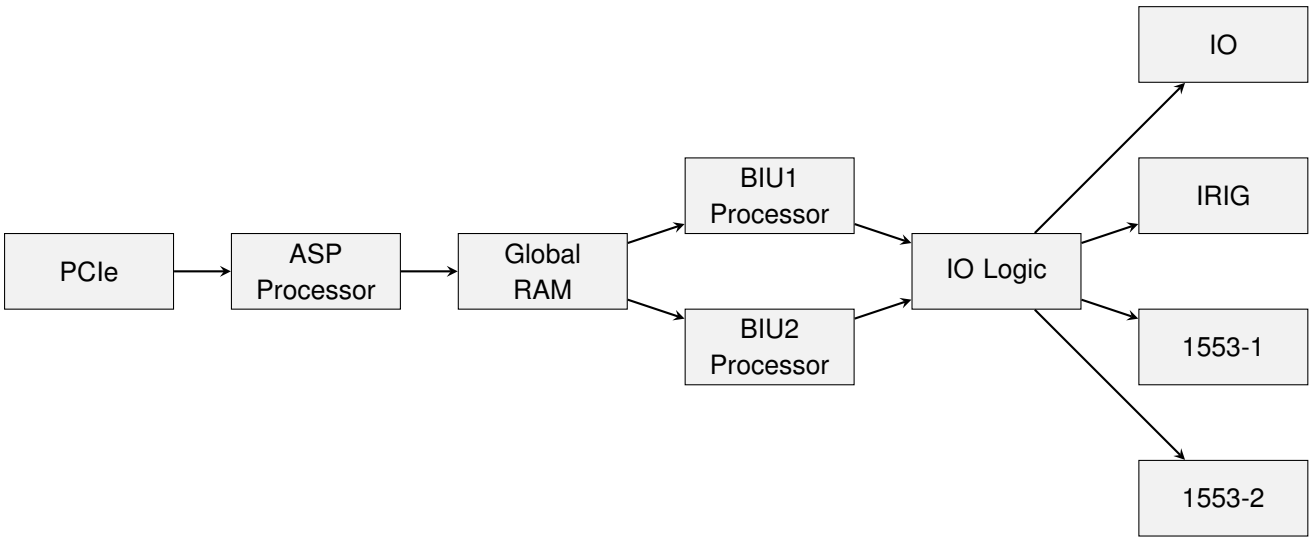
If you have just purchased your module, the associated Getting Started Manual provides first time users with instructions on software installation, hardware setup and starting a sample project.

The following sections provide an overview of the hardware and software architecture, and a description of the API S/W Library.

3.1 Hardware Architecture

The AIM 1553 bus analyzers are based upon AIM's "Common Core" hardware architecture with different bus interfaces. Among others the following busses and form factors are supported: PCIe, PCIe Mini Card, XMC, PXIe, PCI, cPCI, PCIX, PXI, PMC, VME, USB and Ethernet.

The following graph shows a schematic block diagram of a dual channel 1553 card. Detailed hardware design information can be found in the associated hardware manuals.



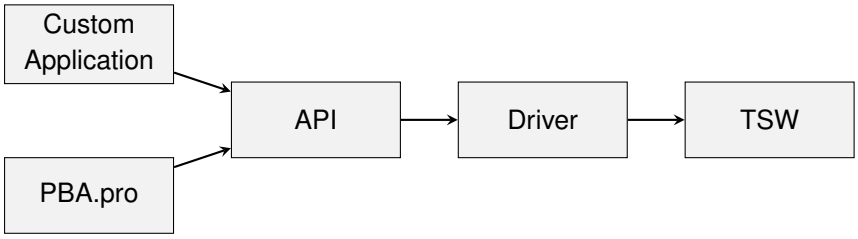
The main hardware components in AIM's "Common Core" design illustrated above are are as follows:

- **PCIe** - represents the host interface. This can be any of the supported buses PCIe, PCI, USB, VME among others
- **ASP Processor** - translates the API calls from the host into Firmware commands and processes Firmware data for the Host. Some modules do not have a dedicated ASP processor and include the TSW within the device driver.
  - Run the on-board driver software (TSW or Target Software)
  - Setup the Global RAM for BIU Processor operation
  - Interface to the host
- **Global RAM** - Shared communication memory between the BIUs, the ASP and the Host Interface
- **BIU Processor** – each runs a real time Firmware that handles one or two dual-redundant MIL-STD-1553 stream and controls the receive and transmit data flow.
- **IO Logic** – Interface between the BIU Firmware and the HW
  - Receives the incoming serial data stream, detects the synchronization (sync) pattern, converts 16-bit Manchester encoded serial data to parallel and receives the parity bit
  - Generated command and data words on the bus using a Manchester Encoder with full error injection capability
- **IO** – HW IO Signals GPIOs and Trigger lines

- **IRIG** – HW Irig signal lines
- **1553-1** – First HW 1553 stream including primary and secondary lanes
- **1553-2** – Second HW 1553 stream including primary and secondary lanes

3.2 Software Architecture

The AIM hardware design, as shown in the previous section, provides for the utilization of a common application s/w library of function calls to support host application interfaces to the AIM 1553 modules. The following graph shows the high-level software architecture of the AIM 1553 bus interface module and it's interface to a host computer application:



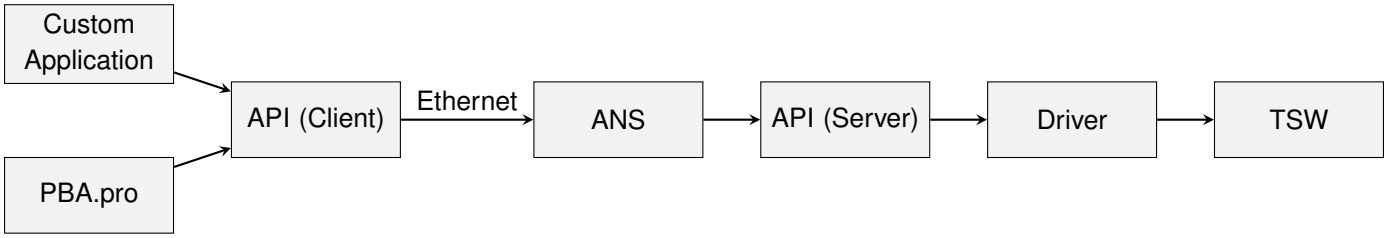
The main software components illustrated above are are as follows:

- **Custom Application** - This can be any custom application written in any of the supported languages which currently are C, C++, LabView VI or Python.
- **PBA.pro** - This is the AIM analyzer software which uses the same API as any other customer written software.
- **API** - stands for Application Programming Interface which is the interface that needs to be programmed in order to access any AIM module. The API is documented in the Reference Manual included and is further explained in the programming sections of this manual. Typical file names for the API are *api_mil.24.dll* for Windows or *libaim_mil.so* for Linux.
- **Driver** – The driver provides access to the memory of the device and handles the physical interrupts. In cases where the device does not contain an ASP Processor the driver does also execute the on-board driver software code. Typical names for the driver are *aim_mil.sys* for Windows or *aim_mil.ko* for Linux.
- **TSW** – The Target Software which is the interface between the API and the on-board Firmware.

As shown in the graph above, the API S/W Library is utilized by the User's Application program (Custom Applicaton) to control the target module. (As an option, the application developer can utilize the AIM PBA.pro Bus Analyzer Software Bus Monitor function to monitor bus traffic setup by the User's Application.) Both the PBA.pro and the User's Application program utilize the same API S/W Library.

The API S/W Library encapsulates operating system specific handling of Host-to-Target communication in order to support multiple platforms with one set of library functions.

The following graph shows the high-level software architecture in case of an AIM network device (e.g. ANET1553) or if the AIM Network Server (ANS) is used:



The software components in case of an ANET1553 access as illustrated above are are as follows:

- **Custom Application** - Custom application as described above
- **PBA.pro** - The PBA.pro analyzer as described above
- **API (Client)** - The API as described above. In case of a network device the API is communicating with the AIM Network Server over Ethernet instead of the device driver.
- **ANS** - The AIM Network Server is listening on socket 1553 for connections from the Ethernet client API. The server forwards each received API function call to the API available on the ANET1553 side or server side.
- **API (Server)** - The API as described above. The same API is used by the ANS to forward the function calls of the client.
- **Driver** – The device driver as described above.
- **TSW** – The Target Software as described above.

3.3 API S/W Library Structure/Content

The API S/W Library function calls are divided into the following subgroups:

- **Library Administration/Initialization Functions** - provide general library initialization, and shutdown, interrupt handler setup, and error message handling setup. These functions will be discussed in Section 4.
- **System Functions** - provide general device control, response timeout setup, IRIG setup, board configuration status and control of the generation of dynamic data words/datasets. These functions will be discussed in Section 4.
- **Calibration Functions** - provide configuration of the physical bus including coupling mode, transmitter amplitude output and data rate (default 1 Mbps). These functions will be discussed in Section 4.
- **Buffer Functions** - provide setup and status of the RT/BC global RAM message buffer memory area and ASP shared RAM dataset buffer area used for message transfers. These functions will be discussed within the context of defining and executing BC and RT transfers in Sections 5, 6 and 7.

- **First-in-first-out (FIFO) Functions** - provide setup and status for FIFO buffers used for 1553 message transfers. These functions will be discussed within the context of defining and executing BC and RT transfers in Sections 5, 6 and 7.

- **Bus Controller Functions** - provide definition of 1553 transfers within the minor frame(s) and setup of the minor frame(s) within the major frame(s) including definition of minor frame timing. The BC functions also provide definition BC transfer properties and real-time BC transfer control including insertion of acyclic messages. These functions will be discussed in Section 6.

- **Remote Terminal Functions** - provide configuration, status and error insertion for RT transfers. These functions will be discussed in Section 7.

- **Bus Monitor Functions** - provide configuration of the Bus Monitor for Chronological recording of all or filtered data streams. These functions will be discussed in Section 8.

- **Replay Functions** - provide configuration of the Replay process to replay pre-recorded Bus Monitor data entries in entirety or filtered by specified RTs. These functions will be discussed in Section 9.

3.4 Application Program Structure Overview

The API function calls can be used to simulate a BC and/or RT on the bus as well as monitor bus activity and replay recorded data. The graph below shows the structure of a basic application program and the API function categories associated with each major part of the program.

Decide if you need to simulate a BC, and/or one or more RT(s). Determine if you want to program the BM to capture/replay data. For BC simulations, know the BC-to-RT, RT-to-BC, and RT-to-RT transfers that must be initiated. Determine how the data words will be generated and whether errors are to be injected into the transfer. Know how all simulated transfers are to be included in the minor/major frame structure.

For RT simulations, know which RT's and RT SA's need to be simulated, the response time and the status to be returned, or whether the RT is to be configured for mailbox monitoring (no response on the bus). For RT-to-RT and RT-to-BC transfers, determine how the data words will be generated and whether errors are to be injected into the transfer. Know whether the RT is required to respond to Mode Codes and which mode codes are required.

If you want the databus to be monitored, know how you want the monitor to be started (triggered) i.e trigger on error, external trigger, first word received... Determine if you want all or specific RT SAs/Modecodes to be monitored.

Determine if you need an interrupt handler to handle BC or RT interrupt signals. BC and RT function calls can be setup to produce interrupts based on certain conditions.

Follow the basic structure below utilizing the function call groups indicated to create your application program.

1

Initialization - Library Administration Functions

- Initialize API Library
- Open Stream
- Optional interrupt handler setup

2

Board Setup - System Functions

- Initialize device
- Define MILBus protocol(A or B) and Response timeout
- Define MILBus coupling mode
- IRIG time setup

3

BC Simulation Setup - BC Functions, Buffer Functions, FIFO Functions

- Initialize the BC retry mechanism, Service Request Control, define whether all transfers use the same/different bus
- Define the properties of each BC to RT, RT to RT, RT to BC transfer to be simulated including data word count, reponse control, retry control, status word handling, error/gap injection, Status word handling and interrupt generation.
- Assign Header ID and and Message Buffers IDs to each transfer.
- Insert initial data words into the transfer message buffers.
- Determine framing mode (Standard or BC Instruction Table mode) and configure minor/major frames or BC instruction tables accordingly.

4

RT Simulation Setup - RT Functions, Buffer Functions, FIFO Functions

- Initialize the RT defining the RT's address, whether RT is simulated or configured for Mailbox monitoring, response time, and the Status word the RT will respond with.
- Assign Header ID and Message Buffers IDs to each sub address or mode code. Enable interrupt generation if required.
- Enable the RT's SA for transmit, receive or modecode transfers, and specific Status word response value.
- For RT to RT or RT to BC transfers, insert initial data words into the message buffers.

5

Bus Monitor Setup - BM Functions

- Initialize monitor
- Configure capture mode for Recording mode
- If required, define the trigger(s) for start of capture/arm the trigger and whether the trigger creates an interrupt.
- If required, define filter captured data based on transfer ID.

6

Start Bus Monitor - BM Functions

- Start receiving and/or monitoring the MILBus data

7

Start RTs/BC - RT Functions, BC Functions

- Start all RTs
- Start the BC

8

Retrieve Status/Data - System Functions, BC Functions, RT Functions, BM Functions, Replay Functions

- BC Status
- RT Status
- Retrieve Captured data

9

Stop RT/BC/BM Shutdown - Library Administration Functions, BC Functions, RT Functions, BM Functions

- Stop BC/RT/BM
- Uninstall any interrupt handlers
- Close each resource
- Exit API library

3.5 MIL-STD-1553 Transfers and Frame Structure Overview

Programming the AIM 1553 bus interface module to simulate a BC and/or RT revolves around the concept of defining transfers. Additionally, for the BC the transfers must then be scheduled into minor and major frames. The following two sections will provide an overview of these two concepts.

3.5.1 About MIL-STD-1553 Transfers

In order to simulate the BC, the user must define BC transfers. A BC transfer is assigned a Transfer ID and Header ID (usually the same number) for the BC side of the BC-to-RT, RT-to-BC or RT-to-RT transfers to simulate on the 1553 bus. The BC Header ID points to memory locations containing Status and Event information for a specific BC transfer and the Message buffer(s) which contain the BC transmit/receive Data words within the 1553 transfer. In order to simulate the RT SA using 1553 protocol standards the user must assign a Header ID which is associated with all the RT SA properties, status and transmit/receive message buffers for the RT side of the BC-to-RT, RT-to-BC or RT-to-RT transfers. For BC and RT simulations, the maximum number of Transfers/Header IDs that can be defined will vary based on the amount of Global RAM available on your board and the current memory layout.

To obtain the maximum number of available transfer ids use the function *ApiCmdSysGetMemPartition*. The output structure *TY_API_GET_MEM_INFO* contains all the information about the memory layout. The following table contains the parameter names for the struct members of *TY_API_GET_MEM_INFO* for Transfer Ids, Header Ids and Buffer Ids.

Struct member	ID type
x_BiuCnt[n].ul_BcXferDesc*	BC Transfer Id count
x_BiuCnt[n].ul_BcBhArea*	BC Buffer Header Id count
x_BiuCnt[n].ul_RtBhArea*	RT Buffer Header Id count
x_Sim[n].ul_BufSize*	RT/BC Buffer Id count

Note:
* n is the BIU that should be checked

Note:
Our examples will utilize the maximum value of 511 Transfer/Header IDs which are available on all our boards.

A BC transfer defines the characteristics/properties (including error injection) for the BC side of any one of the MIL-STD-1553 Command/Data/Status.

All BC and RT message transfers utilize a common message buffer pool located in Global RAM. All BC and RT message transfers require the user to assign at least one Global RAM message buffer. For instance, for an application with a simulated BC and simulated RT, for one BC-to-RT transfer, at least one message buffer will be assigned to the BC transmit message and at least one message buffer assigned for the RT receive message. If there are no MIL-STD-1553 Data words within the 1553 transfer, a message buffer will still need to be assigned. See table above on how to obtain the maximum available buffer ids.

See Section 5 for more details on Buffer Header Ids and Buffer Ids.

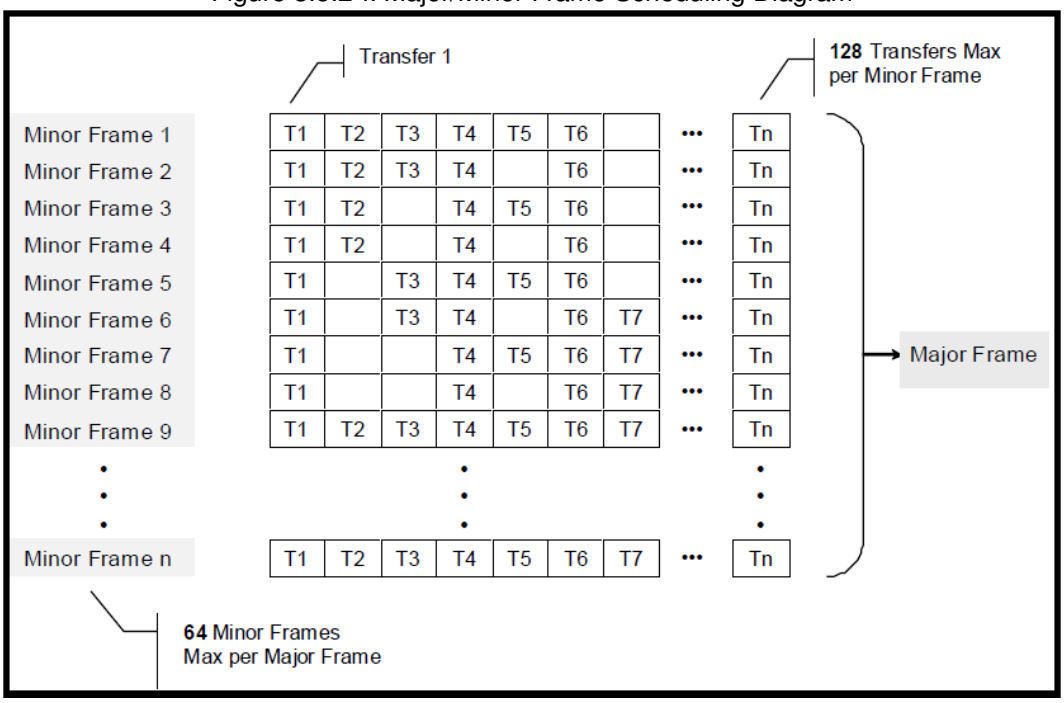
3.5.2 Major/Minor Frame Scheduling Design Concepts

Once the developer has determined the configuration and RT/BC interface requirements of the 1553 application, the scheduling of message transfers on the bus is required. The API S/W supports 128 transfers per minor frame and 64 minor frames per major frame as shown in Figure 3.5.2-I. In this diagram, a transfer consists any of the 1553 transfers on the 1553 bus. Each transfer depicted in this diagram is defined by the API S/W Library as one of the following types (all initiated by the BC):

1. BC-to-RT
2. RT-to-BC
3. RT-to-RT.

Note:
If the function *ApiCmdBCMFrameDefEx* is used the application can set up 512 minor frames per major frame.

Figure 3.5.2-I: Major/Minor Frame Scheduling Diagram



4 Library Administration And System Programming

An AIM 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 8 streams that may either be configured as RT, BC, BM or a combination. Some functions to show more information about a specific board and its channels are listed in Section 4.3

This section will discuss some of the typical scenarios a programmer would encounter that would require the use of the library administration, system, and calibration calls. These scenarios include:

- Initializing your application and application interface to the AIM board
  - for AIM boards located on your computer
  - for AIM boards located on a remote server
- Getting AIM Board Status and Configuration Information
- Defining MILbus protocol
- Defining External Connectivity
- Configuring response timeout
- Utilizing IRIG-B
- Interrupt handling
- Discrete I/O programming
- Debugging

4.1 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 ANET1553 device works like an ANS Server with a single board inserted. Please refer to the 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:

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

  Note: do not call *ApiInit* after *ApiConnectToServer*

- **board mapping functions (only VxWorks)** - the commands *AiPciScan*, *AiVmeExamineSlot* and/or *AiVmeMapModule* are used to map a board to the PCI or VME bus. The output of the command *AiVmeMapModule* is the board ID assigned to this board and shall be used as input parameter of *ApiOpen*. If interrupts are used, the function *AiVmeInitGenericInterrupt* also has to be called at this point.
- ***ApiOpenEx*** returns a handle which will be used to reference a stream on an 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 device_id of *ApiOpenEx* shall be set to the value returned by function *AiVmeMapModule*.
- ***ApiCmdReset*** resets a stream to initial state. After power up the device is already in a reset state and this function call is not necessary for proper board operation. However, for cases where a previous application has been running and the state of the board has been altered it is common practice to call the *ApiCmdReset* function during the start of a new application. Please note that even though the function resets a dedicated stream it does also reset the shared simulation buffer area. This does affect data transmitted and received by all streams on the board. Use the Reset Control mode API_RESET_WITHOUT_SIMBUF to avoid this.

  ApiCmdReset will return the Bus Monitor and Simulator buffer memory size/address information at the address specified in the third parameter.

  Parameters (of many) initialized by *ApiCmdReset* include but is not limited to:

  – MILBus protocol which is set to a default value of Type B (1553B).
  – Response Timeout is set to a default value of 14µs.

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

```
TY_API_OPEN xApiOpen;

wBoardCountLocal = ApiInit(); /* Local board access */

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

xApiOpen.ul_Module = 0;
xApiOpen.ul_Stream = API_STREAM_1;
sprintf( xApiOpen.ac_SrvName, xConfig.acServer );

retVal = ApiOpenEx(&xApiOpen, &ulModHandle);

/* At this point the opened stream can be used */
```

4.2 Board cleanup and program termination

- ***ApiClose*** - The *ApiClose* function terminates the communication to the AIM board(s) after all streams have been closed. Subsequent calls of driver-related functions after issuing an *ApiClose* function call might result in an error code and should be avoided. To re-establish connection to the AIM board, the *ApiOpenEx* function must be called.
- ***ApiDisconnectFromServer*** - If a connection to a remote PC or ANET device has been established, the command *ApiDisconnectFromServer* function should be issued to disconnect the network connection.
- ***ApiExit*** - The *ApiExit* stops the PNP thread and deletes allocated data for a safe unload of the library.

```
retVal = ApiClose(ulModHandle);

if( useServer )
    ApiDisconnectFromServer(strServerName);

ApiExit();
```

4.3 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:

- ***ApiCmdBite*** - Performs a self-test of the AIM board.
- ***ApiGetBoardName*** - Shows the name of the board.
- ***ApiReadAllVersions*** - Reads the versions of all components.
- ***ApiCmdSysGetBoardInfo*** - Retrieves basic information about the board.

4.4 Symmetric Multiprocessing

This section describes how tasks can be executed in parallel and what limitations the API has in terms of parallel execution. We use the term Symmetric Multiprocessing in this document because all the tasks and threads use a shared I/O path in the system driver to access the board.

For this document it is assumed that the concurrent tasks or threads access unique parts of the module. E.g. two threads could be used to access stream 1 and 2 each or BC and RT. It is not recommended to implement access to the same component on a stream. E.g. two threads control RT receive and RT transmit.

Different operating systems have different naming schemes on addressing task with shared memory vs. tasks with independent memory. In this document we use the term task to address a program that does not have a shared memory. This is the case for all our operating systems when one starts the same executable twice. In this case, where the memory is not shared, each task has its own library instance. We use the term thread for cases where the memory is shared. In this case all threads use the same library instance and therefor access the same global variables in the library.

To explain the technical differences of parallel code execution we use the terms multi-threading and multi-tasking.

4.4.1 Multi-threading

We use multi-threading in this document to explain cases where a single task spawns multiple threads with a common memory segment. In this case the threads share the library instance and its global variables in memory. Threads on Windows are typically created with the function CreateThread. Threads on Linux are typically created with pthread_create.

The AIM library is thread safe with a few exceptions that are listed below.

The recommended API function call order is as follows:

1. Call ApiInit
2. Call ApiOpenEx for all streams that need to be accessed
3. Call ApiCmdReset for all streams that need to be accessed
4. Spawn the threads and pass the module handle assigned to the thread.
5. Each thread is now calling the API functions in parallel
6. Join the threads in the main application
7. Call ApiClose for each open handle
8. Call ApiExit

The API contains some functions that are not thread save. This functions can only be called in the main application before and after the thread execution.

ApiInit This function initializes the list of boards and is called only once at the beginning of each application.

ApiExit This function removes the list of boards and cleans up the memory. It is called only once at the end of each application.

ApiConnectToServer This function initializes the list of boards on a remote PC or ANET device and is called only once at the beginning of applications that require remote access.

ApiDisconnectFromServer This function removes the list of boards on a remote PC or ANET device and cleans up the memory. It is called only once at the end of applications that require remote access.

ApiSetDllDbgLevel This function is not thread safe and only needs to be called if necessary.

ApiFlushIntHandler This function deletes all pending interrupt events of the device.

4.4.2 Multi-tasking

We use the term multi-tasking in this document to explain cases where multiple task or programs are started which do not share a common memory segment by default. In this case the tasks do not share a common library instance and do not share the libraries global variables. Tasks on Windows and Linux are typically created by executing separate applications or by executing the same application several times.

The recommended API function call order is as follows:

1. Call ApiInit
2. Call ApiOpenEx for all streams that need to be accessed
3. Call ApiCmdReset rc=API_RESET_WITHOUT_SIMBUF for all streams
4. Each task is now calling the API functions in parallel
5. Call ApiClose for each open handle
6. Call ApiExit

4.4.3 Shared resources handling

We recommend to handle the shared resources of the board by a single thread or task. Accessing shared resources on the board can have an impact on all streams and all threads.

ApiCmdReset This function can be called for each stream individually. However, by default this function call resets all buffer data to zero. If parallel access to the other streams is expected use rc=API_RESET_WITHOUT_SIMBUF to prevent the function from initializing the shared simulation buffer.

ApiCmdBufDef Reading and writing shared buffer data from different threads is possible but not recommended. We recommend to partition the available buffers into segments and use each segment in a dedicated thread. In case buffer sharing is required it is possible to read and write buffers from different threads.

ApiCmdSetIrigTime The board has only one Irig time. Setting the time has an effect on all streams and should be used with care.

ApiCmdSetIrigStatus The board has only one Irig circuit. Setting the time to internal or external has an effect on all streams and should be used with care.

ApiCmdInitDiscretes The board has only one set of discrete registers. Configuring or setting a discrete should be done with care.

4.5 Defining MIL-STD-1553 Protocol

If your application requires MIL-STD-1553 Type A for any or all RTs, you should include the System function, *ApiCmdDefMilbusProtocol*, in your program. Default after *ApiCmdReset* is Type B.

Note:
This function is not available for all modules. Please see the reference manual for a detailed table of supported functions for your module.

4.6 Defining External Connectivity

Initialization and configuration of the MILBus for your application may require the execution of the following System and Calibration functions:

- ***ApiCmdCalCplCon*** - defines the coupling mode for the BIU:

  The MILbus coupling network of the Programmable-PBI consists of sophisticated relay circuitry which offers various coupling capabilities. The coupling modes which can be programmed by your software application for each MILbus (primary and secondary) include:

  - Isolated (Internal termination) (*API_CAL_CPL_ISOLATED*)
  - Transformer coupled (*API_CAL_CPL_TRANSFORM*)
  - Direct coupled. (*API_CAL_CPL_DIRECT*)
  - Transformer coupled with resistive network emulation. (on-board MILbus network simulation) (*API_CAL_CPL_EXTERNAL*)
  - Onboard, digital wrap-around loop between MILbus Encoder and Decoder (*API_CAL_CPL_WRAP_AROUND_LOOP*)

  Note:
  Have a look to the 1553 Reference Manual for some coupling restrictions on various board types.

  In the network emulation mode, the MILbus emulation circuitry emulates a transformer-coupled network without the use of MILbus couplers (using a resistor network). Thus, an external dual-redundant MIL-STD-1553 Terminal can be directly connected to the module.

  The following functions are examples of the *ApiCmdCalCplCon* function. The coupling parameter you use will be dependent upon your MILbus configuration.

```
// Setup for Isolated Bus Coupling
ApiCmdCalCplCon( ulModHandle, 0, API_CAL_BUS_PRIMARY,   API_CAL_CPL_ISOLATED );
ApiCmdCalCplCon( ulModHandle, 0, API_CAL_BUS_SECONDARY, API_CAL_CPL_ISOLATED );
```

- ***ApiCmdCalXmtCon*** - defines the amplitude of the MILbus transceiver: Some boards contain MIL-STD-1553 transceivers that offer the capability to control the output amplitude on the MILbus via the voltage control pins. If necessary for your application, the output amplitude of the MILbus transceiver can be adjusted using the *ApiCmdCalXmtCon* function. Note that the relationship between the input value and the transceiver might not be exactly linear. If exact output voltage

values are required it is recommended to determine the input values for each module type by trying different values. See the Hardware Manual of the module for details.

A digital eight bit value (0..255) has to be provided as input to the *ApiCmdCalXmtCon* function. The 100% value depends on the transceiver type, the coupling mode and the bus termination. A typical value is 22 Volts peak-to-peak for a transformer coupled stub terminated with 70 Ohm.

Note:
The output amplitude value is dependent on the coupling mode setting.

The *ApiCmdCalXmtCon* function below adjusts the output amplitude:

```
// Modify Output Amplitude (check with Scope) to 25%
ApiCmdCalXmtCon(ulModHandle,0,API_CAL_BUS_PRIMARY,0x40);
// Modify Output Amplitude (check with Scope) to 63%
ApiCmdCalXmtCon(ulModHandle,0, API_CAL_BUS_PRIMARY,0xA0);
// Set maximum
ApiCmdCalXmtCon(ulModHandle,0, API_CAL_BUS_PRIMARY,0xFF);
```

Note:
This function is not available for all modules. Please see the reference manual for a detailed table of supported functions for your module.

- ***ApiCmdCalSigCon*** - enables/disables a 1 Mhz square wave calibration test signal: The AIM board is capable of generating a 1MHz test signal applied at the Encoder outputs of the primary or secondary MILbus. This test signal can be enabled/disabled using the *ApiCmdCalSigCon* Calibration function. The amplitude of this signal can be controlled using the *ApiCmdCalXmtCon* as described above. Following is sample code to enable the 1 Mhz test signal, modify the amplitude output, reset the amplitude output to default, then disable the 1 Mhz test signal.

```
// Enable Test Signal
ApiCmdCalSigCon(ulModHandle,0, API_CAL_BUS_PRIMARY,API_ENA);
// Modify Output Amplitude (check with Scope) to 63%
ApiCmdCalXmtCon(ulModHandle,0, API_CAL_BUS_PRIMARY,0xA0);
// Set maximum
ApiCmdCalXmtCon(ulModHandle,0, API_CAL_BUS_PRIMARY,0xFF);
// Disable Test Signal
ApiCmdCalSigCon(ulModHandle,0, API_CAL_BUS_PRIMARY,API_DIS);
```

4.7 Configuring Response Timeout

Configuration of timeout values for your application may require the execution of the following functions:

- **ApiCmdDefRespTime** - defines the timeout value (default is 14 µs) The Response Timeout value is used in all modes of operation to define the minimum time the Bus Controller waits for a Status Word response. The response timeout value is used by the Bus Controller and Bus Monitor as "No Response" timeout and from the RT's to calculate the RT-to-RT Status Word timeout. The default

timeout value, as initially configured using the *ApiCmdReset* function is set to 14µs for compliance with the specification MIL-STD-1553B. The Response Timeout value, however, can be modified by the *ApiCmdDefRespTout* function within a range of 0 to 63.75 in .25 µs steps.

Attention: Due to the Response-/Gap time measurement specification of MIL-STD-1553, the Response time is measured from the mid-bit zero crossing of the last bit of the previous word to the mid- zero crossing of the Command Sync of the current word. Thus, timeout value of (e.g.) 14µs allows a Bus dead time of 12µs at 1 Mbit Transmission Mode.

Note:
For broadcast transfers, the response timeout is used to check that no status word has responded. Therefore, if the response timeout is greater than the Intermessage Gap Time in Standard Gap Mode, the Intermessage Gap will be extended.

4.8 Utilizing IRIG-B

The API S/W Library provides four System function calls for IRIG-B processing, these include:

- ***ApiCmdSetIrigTime*** - Sets the IRIG-B time on the on-board IRIG timecode encoder
- ***ApiCmdSetIrigStatus*** - Sets the IRIG-B internal/external status on the on-board IRIG timecode encoder
- ***ApiCmdGetIrigTime*** - Reads the IRIG-B time on the on-board IRIG timecode decoder
- ***ApiCmdGetIrigStatus*** - Reads the IRIG-B internal/external status on the on-board IRIG timecode decoder

The following is an example of the *ApiCmdSetIrigTime*.

```
TY_API_IRIG_TIME api_irig_time;
// Set onboard IRIG time to Day 288 11:32:25 (Hours:Minutes:Seconds),
api_irig_time.day  = 288;
api_irig_time.hour = 11
api_irig_time.min  = 32
api_irig_time.sec  = 25;
ApiCmdSetIrigTime( ulModHandle, &api_irig_time);
// ... wait 3 seconds until time is locked ..
```

Note:
To obtain an accurate time stamp value you should delay the immediate reading of the IRIG time by 3 seconds.

Note:
IRIG time starts with "DAY one" (First of January) not with "DAY zero".

4.9 Interrupt Handling

If setup by the user, interrupts can be generated by the BC, RT, BM and/or Replay functions for different events. Interrupt Log list Event Entries (see *ApiInstIntHandler* in API reference manual for Log list Event Entry format) are updated by the BIU when an interrupt occurs.

The user-developed interrupt handler should include code to check the Interrupt Log list Event Entry for expected interrupts and process as required based on the user's application requirements.

Note:
The interrupts are asserted at the end of a transfer. Thus, more than one interrupt event for a BC/RT transfer can become valid. In this case, only one interrupt entry with multiple interrupt reason bits set is written to the interrupt log-list and one physical interrupt on the ASP will be asserted.

The functions available to setup interrupts and interrupt handler execution include the following Library Administration functions:

- **AiVmeInitGenericInterrupt** - (only for VME environments) Allows to provide environment specific interrupt setup and to set interrupt level and vector of the VME bus interrupt.
- **ApiInstIntHandler** - Provides a pointer to the interrupt handler function. The following code installs an Interrupt Handler function named userInterruptFunction to handle interrupts generated by the BC (*API_INT_BC*) and the RT (*API_INT_RT*). On 1553 modules the second argument is always *API_INT_LS*.

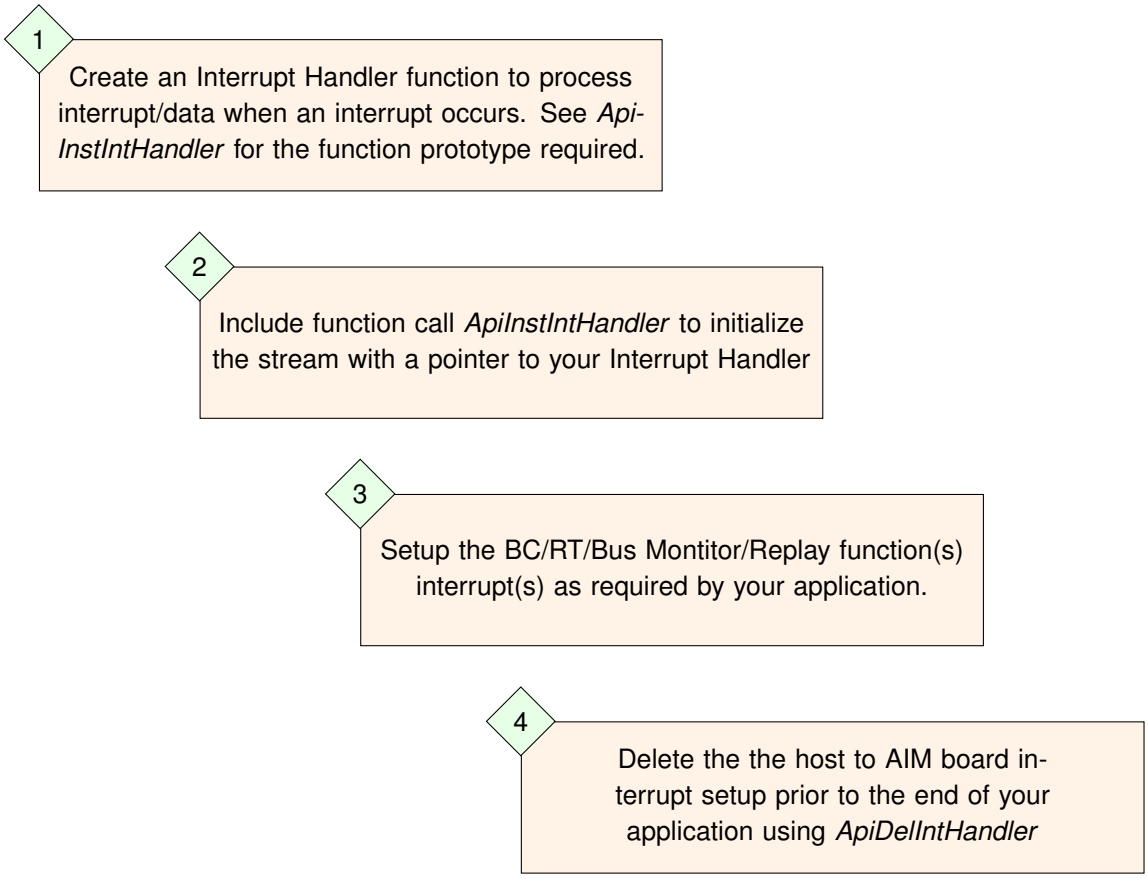
```
//Install Interrupt Handler function to handle BC and RT interrupts
ApiInstIntHandler( ulModHandle,
                   API_INT_LS,
                   API_INT_BC,
                   userInterruptFunction );
ApiInstIntHandler( ulModHandle,
                   API_INT_LS,
                   API_INT_RT,
                   userInterruptFunction );
```

  - The Interrupt Handler function is a function that you create to perform application specific processing based on the type of interrupt to be monitored.
  - Only one interrupt handler is required, however, you can also create one interrupt handler for each type of interrupt.
  - On Windows interrupt callbacks must be defined as __cdecl.
  - Interrupt Handler function input parameters must follow a pre-defined format as defined in the *ApiInstIntHandler* function call in the Reference Manual.

- ***ApiDelIntHandler*** - Removes the pointer interface to the interrupt handler function. This function should be called prior to the module close (*ApiClose*). The following code uninstalls an Interrupt Handler function used to handle interrupts generated by the BC (*API_INT_BC*) and the RT (*API_INT_RT*).

```
//Uninstall the BC and RT interrupt handler function(s)
ApiDelIntHandler( ulModHandle, API_INT_LS, API_INT_BC );
ApiDelIntHandler( ulModHandle, API_INT_LS, API_INT_RT );
```

Interrupt Setup Process:



Further definition and examples of these interrupt scenarios can be found in the BC, RT, BM or Replay applicable sections to follow. In addition, the sample program, *ls_interrupt_sample.c* is included and provides an excellent example of Interrupt Handling programming.

4.10 Debugging

The API S/W Library provides the developer with the capability to control the output of error messages which are detected by the API S/W Library functions using the *ApiSetDllDbgLevel* function. One, multiple, or all types of error messages can be enabled/disabled by using this function.

The default setting provides for the output of *DBG_ERROR* outputs to warn the application user of "out of range" parameters within the API function parameters used in the software program and if any errors occur within the on-board software. See Section 10 for additional troubleshooting techniques.

The types of error message outputs are shown in the following table:

Constant	Description
DBG_DISABLED	Force no debug output
DBG_INT	Force interrupt related debug output
DBG_INIT	Force initialization related debug output
DBG_OPEN	Force module open related debug output
DBG_CLOSE	Force module close related debug output
DBG_IO	Force module I/O related debug output
DBG_READREC	Force recording related debug output
DBG_WRITEREP	Force replay related debug output
DBG_MUTEX	Force mutex related debug output
DBG_TRACE	Log function calls in aim_mil.log
DBG_INFO	Force informational debug output
DBG_ERROR	Force general error related debug output (e.g. range check errors)
DBG_ERRORMSG	Force error message box, if I/O to the board fails with error or range check fails
DBG_ALL	Force all available debug output

4.11 GPIO Programming

Some AIM modules provide up to eight GPIO Discrete I/O signals. See the Hardware Manual of your device for details and pinout. Depending on the hardware the GPIO can be used as input, as output or both.

The following function calls are required to generate outputs or receive inputs from these GPIO pins:

- **ApiCmdInitDiscretes** – used to define whether the GPIO signal is to be as an input or output signal. This function call must be called before either of the following two function calls.
- **ApiCmdWriteDiscretes** – used to generate a discrete output to one or more of the eight discrete GPIO pins which have previously been configured as output with *ApiCmdInitDiscretes*.
- **ApiCmdReadDiscretes** – used to read whether the GPIO discrete register has been set. The GPIO register bit set must have previously been configured as input with *ApiCmdInitDiscretes*.
- **ApiCmdReadDiscretesConfig** – used to read the current GPIOs configuration.
- **ApiCmdReadDiscretesInfo** - can be used to obtain information about the number of available GPIOs and if these can be used as input, output or both.
- **ApiCmdSysGetBoardInfo** - can be used to obtain the number of available GPIOs (*TY_BOARD_INFO_DISCRETE_CNT*) as well as their configuration (*TY_BOARD_INFO_DISCRETE_CONFIG*)

5 Simulation Buffer Programming

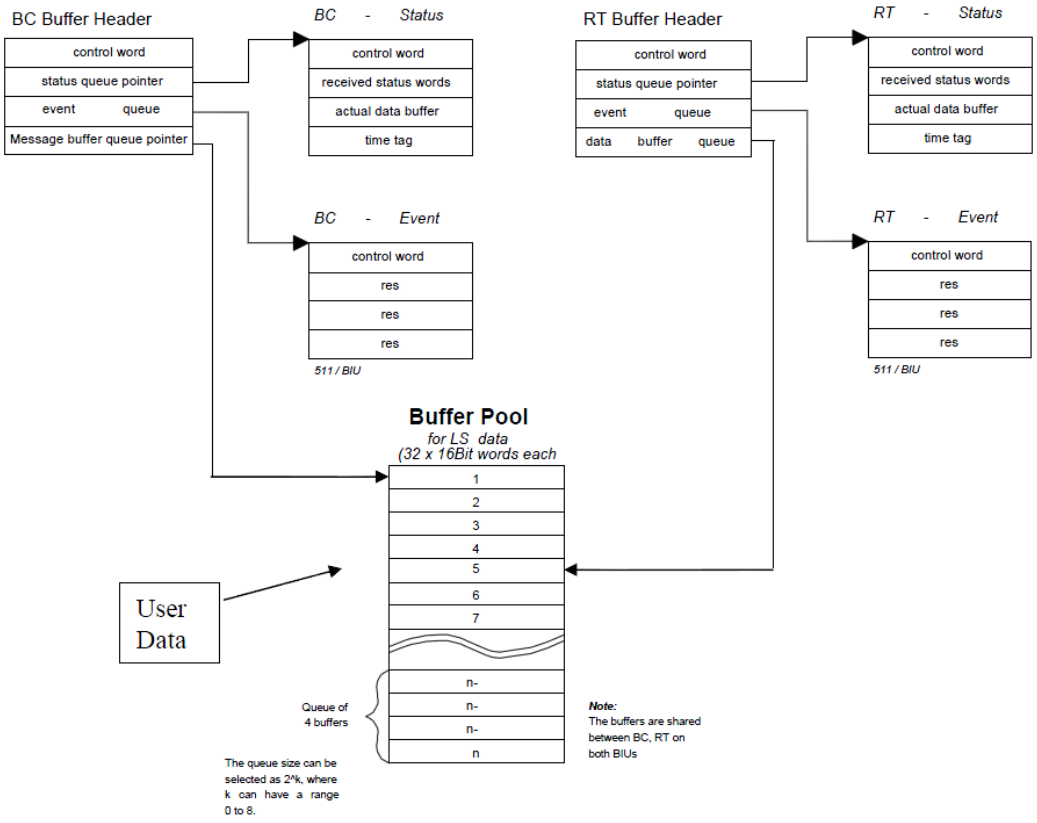
Data transferred and received on the bus by either the BC, the RT or both is stored in simulation buffers. The following sections are here to provide a deeper understanding of how these buffers are handled by the firmware and how data consistency in receive and transmit buffers can be ensured.

5.1 1553 Buffer Header Ids and Buffer Ids

Basically Buffer Header Ids and Buffer Ids are indexes within chunks of memory on the device. Since the firmware requires one buffer for internal use the available ids start with 1. With a call to *ApiCmdBCBHDef* or *ApiCmdRTBHDef* the address of the data buffer is calculated and stored in the Buffer Header descriptor. In case of a buffer queue size greater than *API_QUEUE_SIZE_1* the size is stored in the Buffer Header descriptor too.

The Figure 5.1.0-I shows the connection between BC/RT buffer headers and the Buffer ID in the Buffer Pool. It is important to understand that on most devices there is only one Buffer Pool for all streams as well as BC and RT. See Section 5.1.2 for additional information.

Figure 5.1.0-I: Buffer and Header ID connection



5.1.1 1553 Buffer Header IDs

Each stream has a separate chunk of memory for BC and for RT operation. This means that Buffer Header IDs are not shared by BC, RT operation and are not shared between different streams.

The available BC Buffer Header IDs count can be obtained with *ApiCmdSysGetMemPartition* and is stored in *ax_BiuCnt[stream].ul_BcBhArea*. A BC Buffer Header ID is configured with *ApiCmdBCBHDef*.

The available RT Buffer Header IDs count can be obtained with *ApiCmdSysGetMemPartition* and is stored in *ax_BiuCnt[stream].ul_RtBhArea*. A RT Buffer Header ID is configured with *ApiCmdRTBHDef*.

5.1.2 1553 Buffer IDs

On most cards there is only one chunk of memory in which all Buffer Data is stored. A Buffer Id is an index which is multiplied by 64 (32*16bit words) to get the start address of the requested buffer within this chunk of memory. The first 32 byte are reserved for the firmware which is the reason why Buffer Ids start with index one. The idea behind this concept is that Buffer Ids can be shared. The same chunk of on board memory, identified by its id, can be assigned to a receive and an transmit buffer. This enabled the board to forward received data on a different Stream, BC and/or SA.

The available Buffer IDs count can be obtained with *ApiCmdSysGetMemPartition* and is stored in *x_Sim[0].ul_BufCount*. In case of an ACX-6U card which has a second memory all streams of the second PBI use the ids from *x_Sim[1].ul_BufCount*.

Be aware that multiple consecutive ids are occupied if a buffer queue is configured with *ApiCmdBCBHDef* or *ApiCmdRTBHDef* qsize greater than *API_QUEUE_SIZE_1*.

The Figure 5.1.2-I contains unique single buffers, unique buffer queues and one shared single buffer. The Buffer ID 10 is assigned as single buffer to BC Header ID 6 as well as RT Header ID 1. Thus any data received into this buffer will be transmitted with the next transmit operation. Regardless of BC or RT receive/transmit operation.

Buffer Ids 2 to 5 belong to a buffer queue assigned to BC Header 2. This means that Buffer IDs 3,4 and 5 even though never referenced directly in the application are used by the Firmware and should not be assigned to a different Header Id. The exception to this rule would be a shared buffer as explained with Buffer Id 10 above.

Be extra careful when choosing a Buffer Id. It is common practice to split the available ids into blocks of ids used for one stream and BC or RT.

Figure 5.1.2-I: Example Buffer and Header ID usage

BC Header Ids		
ID	Buffer ID	Size
1	1	1
2	2	4
3	6	2
4	8	1
5	9	1
6	10	1
...	...	...

RT Header Ids		
ID	Buffer ID	Size
1	10	1
2	11	1
3	12	4
4	16	1
5	17	1
...	...	...

Buffer Ids	
ID	DATA
1	32 Words
2	32 Words
3	32 Words
4	32 Words
5	32 Words
6	32 Words
7	32 Words
8	32 Words
9	32 Words
10	32 Words
11	32 Words
12	32 Words
13	32 Words
14	32 Words
15	32 Words
16	32 Words
17	32 Words
18	32 Words
19	32 Words
...	...

5.2 1553 Buffer Data Consistency

In order to guarantee real time behavior of the firmware there is no read and write lock implemented between the firmware and the host software. Because the firmware and the host software run asynchronously writing to a 1553 buffer might clash with reading the same buffer on a different processor. In order to guarantee buffer data consistency the host application needs to be synchronized with the firmware. There are two options commonly used. In the first option buffer read and write access by the host is done within an interrupt callback function. In the second option buffer queues are used to maintain a set of buffers that can be read while the firmware updated the other buffers in the queue. It is also common to have a combination of both methods. For buffer write access the most common way is using a double buffer mechanism with a host controlled buffer and interrupt synchronization. For data buffer read access it is more common to have a cyclic queue that can be polled. Both methods are implemented within our BSP sample program and described below.

Transmit double buffering can also be implemented in a host controlled way. In this case the firmware will not alter the buffer queue index itself but will use the same index over and over again. The host application can update the inactive buffer without provoking a collision. After the buffer update the buffer index used by the firmware can be updated by the host application. This does however only work if the buffer update rate is lower than the firmware transmission buffer rate.

5.2.1 sample/src/ls_interrupt_sample.c

This sample that can be found in the BSP sample folder shows how a consistent 1553 buffer can be updated by the host while it is transmitted on the bus. The sample uses a combination of a cyclic double buffer and synchronizes the buffer update with the firmware by using interrupt callbacks.

ApiCmdRTBHDef is used with qsize set to API_QUEUE_SIZE_2 to generate a cyclic double buffer. This gives the host application more time to update the buffer in case of high OS latency. In case of a OS with low latency a single buffer could be used as well as long as the buffer transmission rate on the 1553 bus is known to be lower than the OS interrupt latency and buffer write call time. Note that interrupt latency values and buffer update times are hardware architecture and OS specific and can not be provided by AIM.

ApiCmdBCXferDef tic parameter set to API_BC_TIC_INT_ON_XFER_END is used to enable the user interrupt generation on the dedicated transfer.

ApiInstIntHandler with parameters API_INT_BC and BcInterrupt is used to provide the library with the callback function BcInterrupt to be called every time the transfer transmitted.

As soon as the BC framing is started the callback function BcInterrupt is called every time the firmware finished transmitting the transfer. We know which buffer the firmware transmitted and we can update this inactive buffer while the firmware might already send the second active buffer.

The advantage of the double buffer is that we can work with higher interrupt latencies. The disadvantage is that we are always one buffer behind, meaning the data we write will be transmitted by the next but one transfer.

The BcInterrupt callback handles only a single transfer. If multiple transfers need to be handled in the same way the parameters provided can be used to decode the transfer id and other information. See

description of ApiInstIntHandler within the 1553 Reference Manual for details.

5.2.2 sample/src/ls_buffer_queue_sample.c

The 1553 buffer queue sample shows the approach of using a cyclic receive buffer queue to store the incoming data. The host application polls the receive queue and reads one or more inactive receive buffers from the queue while the firmware can safely write to the current index. Depending on the queue size this approach can buffer cases where the host application might not be in time to read the buffer. The bigger the queue size is the more delayed can the host application be before the buffer is read.

ApiCmdRTBHDef is used with qsize API_QUEUE_SIZE_16 to assign 16 consecutive buffers to the RT receive buffer header. The queue is set up in API_BQM_CYCLIC mode letting the firmware restart at the beginning when the end of the queue is reached.

ApiCmdRTSAMsgReadEx is used to get the current buffer queue index. If the current index is different from the last call the firmware received at least one buffer. In this case the sample application reads all the buffers from the last read position up to one before the current position. The firmware points to the buffer that will be updated next.

6 Bus Controller Programming

The main function of the BC is to provide data flow control for all transfers on the bus. In addition to initiating all data transfers, the BC must transmit, receive and coordinate the transfer of information on the data bus. All information is communicated in command/response mode. The BC sends a command to the RTs, which reply with a response. Information Transfer Formats are shown in Section 2.

Normal BC dataflow control includes transmitting commands to RTs at predetermined time intervals as defined in the minor/major frame definition. The commands may include data or request for data (including status) from RTs. Command word, Data word and Status word formats are shown in Section 2. The BC has control to modify the flow of bus data based on changes in the operating environment. These changes could be a result of an air-to-ground attack mode changing to air-to-air, or the failure mode of a hydraulic system. The BC is responsible for detecting these changes and initiating action to counter them. Error detection may require the BC to attempt communication to the RT on the redundant, backup bus.

If your application requires the simulation of a BC, this section will provide you with the understanding of the BC programming functions required for use within your application. Programming the BC may require the use of more than just the BC functions. Also needed may be the Buffer and FIFO functions. This section will discuss some of the typical scenarios a programmer would encounter that would require the use of the Buffer, FIFO and BC functions respectively. These scenarios include:

- Initializing the BC
- Defining 1553 transfers
  - BC transfers using single buffers
  - BC transfers using multiple buffers
  - BC transfers using FIFOs
  - BC transfers with dynamic data
- BC transfers with error injection
- Defining Minor/Major frames
  - Standard Framing Mode
  - BC Instruction Table Mode
- Starting the BC
- Acyclic 1553 transfers
- BC Interrupt programming
- Status word exception handling

6.1 Initializing the BC

Initialization of the BC must be the first BC setup function call issued. This function call, *ApiCmdBCIni*, will perform setup of the BC global transfer characteristics including:

- **Transfer Retry Protocol** - Definition of the number of retries the BC will perform if a transfer is erroneous, the retry mechanism and the bus switching protocol include selection of one of the following:
  - Retry disabled
  - One retry on the alternate bus
  - One retry on the same bus, one retry on the alternat bus
  - Two retries on the same bus, then one retry on the alternate bus
- **Service Request Control** - When enabled the BC will perform specific pre-defined actions when the Service Request bit in the status word received by the BC is set. The specific pre-defined action the BC will perform is defined for each transfer using the *ApiCmdBCXferDef* function. When this Service Request function is enabled (with the *ApiCmdBCIni* function), and the BC receives a status word with the Service Request Bit set, the BC will automatically generate and transmit to the RT a Transmit Vector Word mode code (Mode code 16). A Vector Word will then be transmitted by the RT containing information indicating the next action to be taken by the BC. The actions taken by the BC following the receipt of the vector word are also defined by the *ApiCmdBCXferDef* function.

  Note:
  If Status Word Exception handling is required for any RT, this function must enable Service Request capability for the BC.
- **BC Transfer Bus Mode/Global Start Bus** - Provides one of the following options:
  - Allows the user to select the bus (primary or secondary) used for a transfer on a transfer-by-transfer basis (using *ApiCmdBCXferDef*)
  - Allows the user to specify a default bus (primary or secondary) for all MILBus transfers

  Note:
  Transfer Bus Mode Global (API_TBM_GLOBAL) is not available for all modules. Please see the reference manual for a detailed table of supported functions for your module.

The following code example uses API S/W Library constants to initialize the BC for Retry disabled, Service Request disabled, Transfer Bus Mode setup so that individual transfers can define the bus used for the transfer using *ApiCmdBCXferDef*. (The last parameter is ignored since the bus used for transfers can be defined individually.)

```
ApiCmdBCIni( ulModHandle,0,
             API_DIS,                  // retr - Retry Mode
             API_DIS,                  // svrq - Service Request / Vector Word Mode
             API_TBM_TRANSFER,         // tbm  - Transfer Bus Mode
             API_XFER_BUS_PRIMARY);    // gsb  - Global Start Bus
```

Note:
ApiCmdBCIni must be called first when programming the BC regardless of the framing mode (Standard or Instruction Table Mode).

6.2 Defining 1553 Transfers

Transfers controlled/generated by the BC will follow the formats/protocol as shown in the Information Transfer Formats in Section 2. The API S/W Library divides transfers into three basic types: BC-to-RT, RT-to-BC and RT-to-RT. Variations of these types of transfers include Mode commands and Broadcast commands.

The API S/W Library supports definition of a minimum value of 511 unique 1553 transfers. See Section 3.5.1 for details on how to obtain the maximum number of available ids. A transfer ID must be assigned by the user for each transfer. The method used by the AIM 1553 bus interface module to accomplish the transfer of the Command, Status and/or Data words within the transfers is to utilize a common buffer pool containing the message buffers located in Global RAM as shown in Figure 5.1.0-I. All BC message transfers require the user to assign at least one Global RAM message buffer. The message buffer will be used to transmit/receive the Data words within the message transfer. If there are no Data words within the transfer, a message buffer will still need to be assigned. (However, the API S/W Library does not prevent the user from using the same buffers in more than one transfer, therefore, the same message buffer can be assigned for transfers that do not require the transmission/reception of Data word(s)). Each message buffer has a unique ID which must be assigned by the user.

In addition to the assignment of the message buffer(s) for each transfer, a Buffer Header ID must be assigned for the BC to enable the processor to identify the location of the buffers used and status and event data generated for each transfer.

1553 transfers will require the use of the following BC function calls:

- **ApiCmdBCBHDef** - this BC function will define a BC Buffer Header structure. As shown in Figure 5.1.0-I, the BC Buffer Header structure enables the processor to identify the location of the message buffers used and status and event data generated for each 1553 transfer. The BC Buffer Header information to be setup by this function includes the following:
  - **BC Buffer Header ID** - the ID of the BC Buffer Header structure.
  - **Message Buffer ID** - the ID of the first buffer in the Global RAM message buffer pool to be used for the transfer of the Data words. See Section 3.5.1 for details on how to obtain the maximum number of available ids. The buffers are shared between BC and RT. A Message Buffer ID of 20 would indicate that the 20th buffer in the Global RAM Buffer Pool will be used. See Figure 5.1.0-I for a diagram of the structure of these message buffers.
  - **Buffer Queue Size** - the number of Global RAM message buffers to be used for the transfer of the Data words. One or more buffers can be used for the transfer. You will always need to assign at least one message buffer from the Global RAM Buffer Pool for your transfer. For example, assigning *API_QUEUE_SIZE_8* for a transfer indicates that 8 contiguous buffers. Using the example of Message Buffer ID of 20 above, and a queue size of 8, message buffers 20 - 27 would be used for the transfer.
  - **Buffer Queue Mode** - specifies the order in which multiple buffers as specified in the Buffer Queue Size, will be filled. In most cases, users will choose to store the Data words into the Message Buffers in a cyclic fashion (*API_BQM_CYCLIC*)
  - **Data Buffer Store Mode** - will allow the user to indicate the actions to be taken in case of error in the transmission or reception of Data words such as whether to keep transmitting the same message buffer at transfer error, or continue with the next buffer in the queue for the next transmission.

Note:
Buffer Queue Size, Buffer Queue Mode, Data Buffer Store Mode, and Buffer Size and the Current Buffer Index can be modified "on the fly" (i.e. after the BC has been started) using the Buffer function call *ApiBHModify*.

- **ApiCmdBCXferDef** - this BC function will utilize the *TY_API_BC_XFER* structure to define the properties of a 1553 transfer. Information contained in this structure will be used to create the Command word, Data word(s)/Mode Code and define the process for handling the Status word. It will also include error injection setup. This information will be sent to the BIU when this function is called.

The information contained in the structure *TY_API_BC_XFER* includes the information defined in the following table.

Struct Element	Definition	CW
xid	Transfer ID	
hid	Buffer Header ID - defines the ID of the Buffer Header that tracks the message buffer(s) used for this transfer.	
type	Transfer Type - BC-to-RT, RT-to-BC, RT-to-RT	X
chn	MILbus - Primary or secondary bus to be used for this transfer	
xmt_rt	Transmitting RT - RT address/number of the transmitting terminal (N/A for BC-to-RT transfer)	X
rcv_rt	Receiving RT - RT address/number of the receiving RT (N/A for RT-to-BC transfer)	X
xmt_sa	Transmitting RT subaddress - RT subaddress of the transmitting terminal or mode control indication (0 or 31)(N/A for BC-to-RT transfer)	X
rcv_sa	Receiving RT subaddress - RT subaddress of the receiving terminal or mode control indication (0 or 31) (N/A for RT-to-BC transfer)	X
wcnt	Word Count/Mode Code field - contains either the word count or the Mode code	X
tic	Interrupt control - setup for generation of a BC interrupt upon end of transfer, transfer error or status word exception	
hlt	Halt control - setup to halt the BC upon end of transfer, transfer error, status word exception or any interrupt	
sxh	Status word exception handling - defines the process to execute upon occurrence of a Status word Service Request	
swxm	Status Word Exception Mask - defines the bits in the status word received by the RT that will be checked by the BC	
rsp	Expected Response - defines the response the BC expects from the RT such as basic response as defined by ApiCmdDefMilbusProtocol, no status word 1 expected, or no status word 2 expected.	
rte	Retry enable flag - enables or disables retry of this transfer upon transfer error. Also allows the user to alternate tranmission of this transfer between the Primary and the Secondary bus.	
gap_mode	Gap Mode - defines the gap between this transfer and the next transfer	
gap	Gap Value - defines the transfer wait time for the gap mode specified above.	
err.type	Error Injection Type - defines 1 of the 14 possible error injection schemes	
err.sync	Error Sync Pattern - sync pattern to be used for command sync or data sync error injection scheme	

Continued on next page

Continued from previous page

Struct Element	Definition	CW
err.contig	Gap Error or Zero Crossing Error Value - the number of half bits to be used for the Gap Sync Error injection scheme or the Zero Crossing Error value to be used for the Zero Crossing Error injection scheme.	
err.err_spec	Word Position - the location of the 1553 transfer word for which the specified error will occur	
err.err_spec	Bit Position- the location of the bit in the 1553 transfer word (above) for which the specified error will occur	
err.err_spec	Number of bits - the number of bits at the location of the bit (above) in the 1553 transfer word (above) for which the specified error will occur	

The following excerpt of code is an example of setting up the BC for a BC-to-RT transfer to RT1, SA1 with a data word count of 4. This 1553 transfer is to be put on the secondary bus, has no associated interrupts, no requests for halt control, no service request handling, no error injection and specifies a gap of 0 µs will cause the BC to generate an intermessage gap of approximately 11 µs.

```
// BC-RT Transfer XF2: C01_R_01_04 (RT01,RCV,SA01,WC4)
memset( &api_bc_xfer, 0, sizeof(api_bc_xfer) );
xfer_id = 2;
api_bc_xfer.xid       = xfer_id;                    // Transfer ID
api_bc_xfer.hid       = bc_hid;                     // BID Buffer Header ID
api_bc_xfer.type      = API_BC_TYPE_BCRT;           // Transfer Type
api_bc_xfer.chn       = API_BC_XFER_BUS_SECONDARY;  // MILbus
api_bc_xfer.xmt_rt    = 0;                          // XMT-RT
api_bc_xfer.rcv_rt    = 1;                          // RCV-RT
api_bc_xfer.xmt_sa    = 0;                          // XMT-SA
api_bc_xfer.rcv_sa    = 1;                          // RCV-SA
api_bc_xfer.wcnt      = 4;                          // Word Count field
api_bc_xfer.tic       = API_BC_TIC_NO_INT;          // Interrupt control
api_bc_xfer.hlt       = API_BC_HLT_NO_HALT;         // Halt control
api_bc_xfer.rsp       = API_BC_RSP_AUTOMATIC;       // Response control
api_bc_xfer.sxh       = API_BC_SRVW_DIS;            // Service Req Handling
api_bc_xfer.rte       = API_DIS;                    // Retry disabled
api_bc_xfer.swxm      = 0;                          // Status Word Exception Mask
api_bc_xfer.gap_mode  = API_BC_GAP_MODE_DELAY;      // Gap Mode
api_bc_xfer.gap       = 0;                          // use default gap
ApiCmdBCXferDef(ulModHandle,0, &api_bc_xfer, &addr);
```

All structs should be initialized to zero with a memset call. All zero parameters can then be skipped during struct initialization.

The following table provides examples of how to change the values in the Transfer definition to setup different types of 1553 transfers.

1553 BC Transfer Definition Parameters				
Transfer structure	**Broadcast Xfer with word count of 2**	**RT-to-BC Xfer to RT02, SA03 with word count of 5**	**Mode code Xfer to RT05 SA0 where mode code = 17 (synchronize)**	**RT-to-RT Xfer from RT04 SA2 to RT03 SA1 with word count = 32**
api_bc_xfer.type	API_BC_TYPE_BCRT	API_BC_TYPE_RTBC	API_BC_TYPE_BCRT	API_BC_TYPE_RTRT
api_bc_xfer.xmt_rt	0	2	0	4
api_bc_xfer.rcv_rt	31	0	5	3
api_bc_xfer.xmt_sa	0	3	0	2
api_bc_xfer.rcv_sa	1	0	0	1
api_bc_xfer.wcnt	2	5	17	0

6.3 BC Transmit and Receive Message Data Word Generation/Processing

Now that you are familiar with the method used to define the characteristics of the 1553 transfers generated by a simulated BC, we can now discuss the following:

- How to setup and place data into the message buffers assigned for Data word transmissions by the BC for BC-to-RT and BC Broadcast type transfers.
- How to setup and obtain data from the message buffers used for Data word reception by the BC for RT-to-BC type transfers.

6.3.1 For BC-to-RT and BC Broadcast Type Transfers (BC Transmit)

The API S/W Library provides several methods to insert real-time/dynamic/fixed user data into the Global RAM 1553 Message Buffers used by the BC transmitting side of the 1553 transfer and specified in the *ApiCmdBCBHDef* function described in the previous section. The methods and functions used for each method are summarized in the following table.

Data Source	Functions Used
Fixed Data	1. *ApiCmdBCBHDef* 2. *ApiCmdBCXferDef* 3. *ApiCmdBufDef* to initialize buffer with fixed data words (or *ApiCmdBufWrite* to initialize a buffer with a single 16-bit word)
With Dynamic Data Words	1. *ApiCmdBCBHDef* 2. *ApiCmdBCXferDef* 3. *ApiCmdBufDef* to setup non-dynamic data 4. *ApiCmdBCDytagDef* to setup 1-4 dynamic data words

Continued on next page

Continued from previous page

Data Source	Functions Used
Using FIFOs	1. *ApiCmdBCBHDef* 2. *ApiCmdBCXferDef* 3. *ApiCmdFifoIni* to initialize the FIFO 4. *ApiCmdBCAssignFifo* to assign the FIFO to the transfer 5. *ApiCmdFifoReadStatus* to determine how much FIFO data to reload 6. *ApiCmdFifoWrite* to fill the FIFO with data
With Dynamic Dataset Buffers	1. *ApiCmdBCBHDef* 2. *ApiCmdBCXferDef* 3. *ApiCmdRamWriteDataset* to fill the dataset buffers with data to be used in the message buffers 4. *ApiCmdSystagDef* to assign the Dataset buffers to the transfer
Using an interrupt handler routine to interrupt on end-of-transfer	See Section 6.8

The Dynamic Data Word Generation method and the FIFO and Dynamic Dataset methods are described further in the following sections.

6.3.1.1 Dynamic Data Word Generation

Using the API function calls describe in the previous section, you can setup function generators to dynamically change data words within the transmit message buffer. The Function generators available are summarized in the following table.

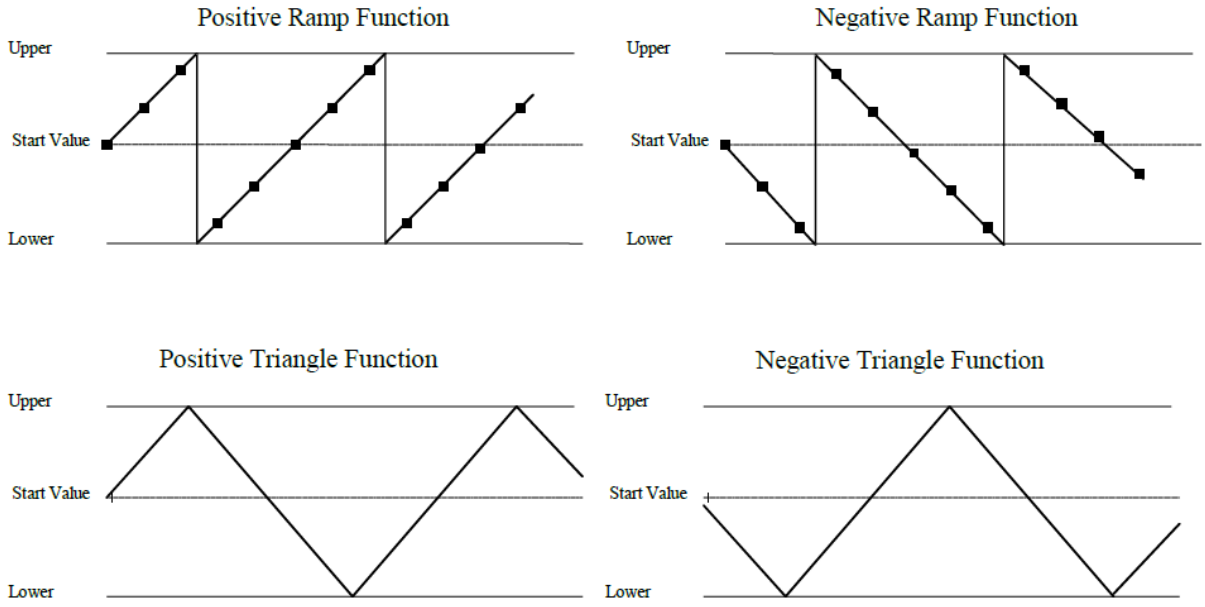
Mode	Description
Function Mode	Provides for up to two dynamic data words per transfer using any combination of the following functions as pictured in Figure 6.3.1-l 1. Positive Ramp 2. Negative Ramp 3. Positive Triangle 4. Negative Triangle 5. Transmit a Data word from a different Message Buffer

Continued on next page

Continued from previous page

Mode	Description
Tagging Mode	Provides for up to four dynamic data words per transfer using Sawtooth Tagging. The Sawtooth Tagging mode provides an incrementer by 1, which is performed on the user-specified location with each execution of the associated BC transfer. The location to be incremented can be setup with an initial value to be incremented each transfer, or the existing value can be incremented. The options are to increment any combination of the following byte or words: 1. 16-Bit Sawtooth 2. 8-Bit Sawtooth LSB (lower byte of word) 3. 8-Bit Sawtooth MSB (upper byte of word)

Figure 6.3.1-I: Data Generation Functions Diagram

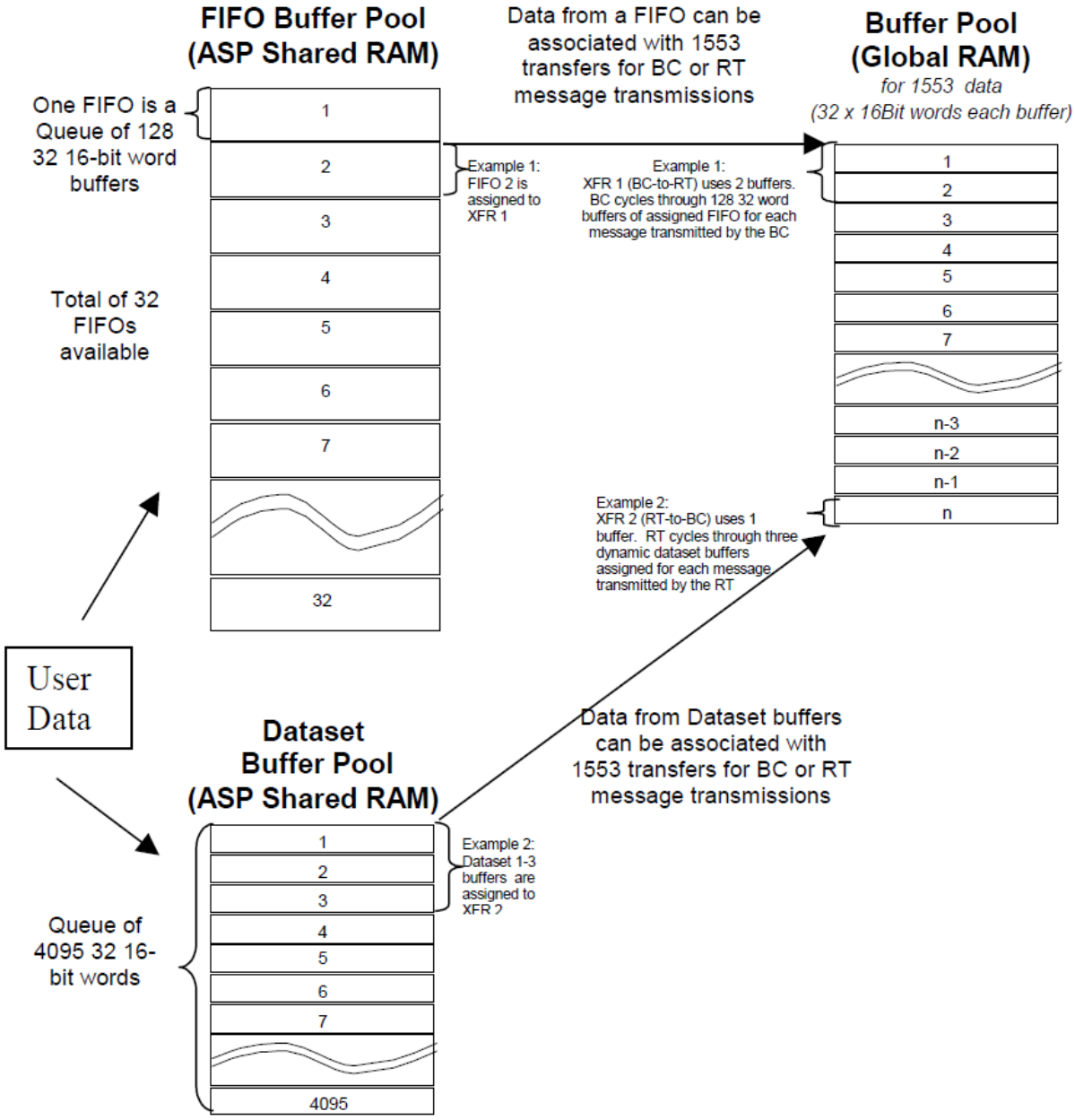


6.3.1.2 Using FIFOs and Dataset Buffers for Data Generation

BC transfer data can be defined using FIFOs or Datasets using the API function calls described in Section 6.3.1. The basic concept of each method is described below and shown in Figure 6.3.1-II:

- Assign a FIFO (ASP Shared RAM) to the transfer. (Each FIFO consists of 128 32-word buffers. There are 32 FIFOs.) Pre-fill the FIFO with application data, the data transmitted in the Global RAM message transmit buffer will be obtained from the FIFO. Re-fill the FIFO as needed.
- Assign Dataset buffers (ASP Shared RAM) to the transfer. (There are 4095 32-word buffers in the Dataset Buffer pool.) Pre-fill the Dataset buffer(s) with application data, the data transmitted in the Global RAM message transmit buffer will be obtained from the Dataset buffers. Refill the Dataset buffers as needed.

Figure 6.3.1-II: BC Transfer Data Generation via FIFO or Dataset Buffers



6.3.2 For RT-to-BC Type Transfers (BC Receive)

Before the BC receives the Data words from the RT, as part of the setup process, you may choose to clear your receive buffer to avoid any Data word transmission confusion. The receive buffers can be cleared before use by using the *ApiCmdBufDef* function.

After the BC has received Data words from an RT, software will need to be added to process the data. Processing data received by the BC can be accomplished in one of two ways: polling at pre-defined intervals to examine the BC data, or setting up the transfer to interrupt at end-of-transfer. To accomplish the interrupt on end-of-transfer method of processing the data, an interrupt handler routine would need to be developed to handle the interrupt which will occur after all Data words have been received. Upon end-of transfer interrupt, the interrupt handler would be called at which time the buffer could be read and processed as required by the application. Interrupt handling is discussed further in Section 6.8.

6.4 BC Transfers with Error Injection

BC transfers can be configured for error injection in any Command word or Data word transmitted by the BC. The BC is capable of injecting one of the following errors for a defined transfer:

- **Command Sync Error** - changes the transmitted Command word sync pattern to one specified by the user
- **Data Sync Error** - changes the transmitted Data word sync pattern to one specified by the user
- **Parity Error** - creates a parity error for the Command word or specified Data word
- **Manchester stuck at high error** - creates a Manchester stuck at high error for a specified Command word, or Data Word at a specified bit position
- **Manchester stuck at low error** - creates a Manchester stuck at low error for a specified Command word, or Data Word at a specified bit position
- **Gap error** - inserts specified Gap after defined Command or Data word
- **Word Count High** - transmits the number of Data words defined for the original transfer plus one
- **Word Count Low** - transmits the number of Data words defined for the original transfer minus one
- **Bit Count High** - transmits a specified number (1-3) additional bits for specified Command word or Data word.
- **Bit Count Low** - transmits a specified number (1-3) less bits for specified Command word or Data word.
- **Zero Crossing Low Deviation Error** - implements zero crossing low deviation at a specified Command word or Data word position, bit position with four predefined deviation values.
- **Zero Crossing High Deviation Error** - implements zero crossing high deviation at a specified Command word or Data word position, bit position with four predefined deviation values.

To setup for BC Command/Data word error injection, the transfer definition parameters pertaining to error injection should be set. The following error injection sample code will setup the transfer to inject a Data Sync Error on the third data word.

Set BC to RT RT01 RCV SA01 WC15 (Inject Data Sync Error in 3rd data word)

```
wpos     = 3; // Inject Error on the 3 data word
bpos     = 0; // not used
bc_bits  = 0; // not used
memset( &api_bc_xfer, 0, sizeof(api_bc_xfer) );
api_bc_xfer.xid           = 1;                          // Transfer ID
api_bc_xfer.hid           = 1;                          // BID Buffer Header ID
api_bc_xfer.type          = API_BC_TYPE_BCRT;           // Transfer Type
api_bc_xfer.chn           = API_BC_XFER_BUS_PRIMARY;    // Bus
api_bc_xfer.rcv_rt        = 1;                          // Receive RT
api_bc_xfer.rcv_sa        = 1;                          // Receive SA
api_bc_xfer.wcnt          = 15;                         // Word Count field
api_bc_xfer.gap_mode      = API_BC_GAP_MODE_DELAY;      // Gap Mode
api_bc_xfer.err.type      = API_ERR_TYPE_DATA_SYNC;     // error injection type
api_bc_xfer.err.sync      = 0x30;                       // invalid data sync 110000
api_bc_xfer.err.contig    = 0;                          // not used
api_bc_xfer.err.err_spec  = 0;
api_bc_xfer.err.err_spec |= (wpos << 16);               // word pos
api_bc_xfer.err.err_spec |= (bpos << 8);                // bit pos
api_bc_xfer.err.err_spec |= (bc_bits << 0);             // bit count
ApiCmdBCXferDef(ulModHandle,0, &api_bc_xfer, &addr);
```

6.5 Defining Minor/Major Frames Content & Timing

Once you have defined your 1553 transfers you will then need to program the device to insert the transfers into the minor/major frames as defined by your unique application requirements. In addition, you can program the BC with one of various options to time the transmission of the minor frames within the major frames. There are two methods provided in the API S/W Library to define the minor and major frames of the BC Bus traffic as follows:

- **Standard Framing Mode** - use this method if your application requires no more than 512 minor frames per major frame and no more than 128 transfers per minor frame. This method is more simplistic in that fewer functions are required.
- **BC Instruction Table Mode** - use this method if your application requires more than 512 minor frames per major frame, more than 128 transfers per minor frame, or the Standard Framing mode does not provide the BC instructions needed to support your framing requirements. In BC Instruction Table mode, major/minor frame size is only limited by allocation of the BC instruction list size in Global Memory. It is more complex, but it provides greater flexibility allowing creation of all available firmware instructions to be defined within the BC Instruction List and usage of several more minor frame transfer/instruction options.

Each mode utilizes the same API S/W Library function, *ApiCmdBCStart*, to start execution of the BC as defined in Section 6.6. The next two subsections will describe the instructions required for each framing mode.

6.5.1 Standard Framing Mode

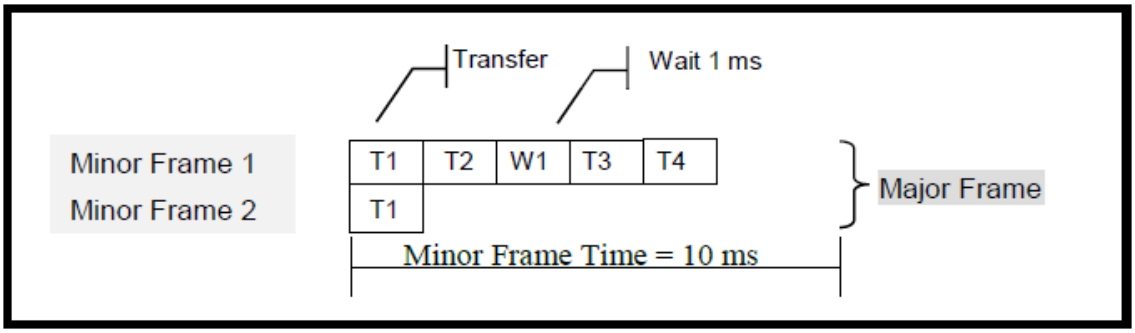
In Standard Framing mode, the MIL-STD-1553 device supports up to 128 transfers per minor frame and 512 minor frames per major frame. With Standard Framing mode, you will first be defining all the minor frames (*ApiCmdBCFrameDef*) then defining which of those minor frames will be placed in the major frame (*ApiCmdBCMFrameDefEx*). Further definition of these two API S/W Library function calls associated with the definition of the minor/major frames for this mode include:

- ***ApiCmdBCFrameDef*** - Defines the sequence of 1553 transfers within a minor frame. The Minor frame characteristics defined by this function include:
  - Minor Frame ID - an ID that you define for the minor frame (1 - 64 is available)
  - Number of Instructions (Transfers) in the Minor frame
  - Type of instruction - Generally this parameter is the 1553 Transfer ID, however, the API S/W Library provides for advanced programming of the minor frames other than transfer identification. Instructions which can be executed by the BC at the programmed location in the minor frame are as follows:

Constant	Description
API_BC_INSTR_TRANSFER	Execute BC Transfer
API_BC_INSTR_SKIP	BC Skip (skip a specified number of instructions)
API_BC_INSTR_WAIT	BC Wait (wait a specified delay time).
API_BC_INSTR_STROBE	BC external Strobe (output strobe pulse)

- ***ApiCmdBCMFrameDefEx*** - Defines the sequence of minor frames within the major frame using the Minor Frame IDs defined in the *ApiCmdBCFrameDef* function above. Therefore, this function should be executed after the *ApiCmdBCFrameDef* function above.

  The following example defines minor/major frames using the Standard Framing mode. This sample code demonstrates the setup of minor and major frames and the starting of the BC using 4 previously setup transfers with Transfer IDs of 1 through 4. The first minor frame includes 5 instructions; (1) execute Transfer 1, (2) execute Transfer 2, (3) execute a wait delay of 1 millisecond, (4) execute Transfer 3, (5) execute Transfer 4. The second minor frame has only one instruction; (1) execute Transfer 1. The second minor frame has only one instruction; (1) execute Transfer 1.



Define Minor Frames:

```
/* Minor Frame 1 Transfer sequence */
api_bc_frame.id       = 1;                          // Minor frame Identifier
api_bc_frame.cnt      = 5;                          // Instructions count
api_bc_frame.xid[0]   = 1;                          // Transfer 1
api_bc_frame.instr[0] = API_BC_INSTR_TRANSFER;
api_bc_frame.xid[1]   = 2;                          // Transfer 2
api_bc_frame.instr[1] = API_BC_INSTR_TRANSFER;
api_bc_frame.xid[2]   = 4000;                       // 1 ms delay in 250ns increments
api_bc_frame.instr[2] = API_BC_INSTR_WAIT;
api_bc_frame.xid[3]   = 3;                          // Transfer 3
api_bc_frame.instr[3] = API_BC_INSTR_TRANSFER;
api_bc_frame.xid[4]   = 4;                          // Transfer 4
api_bc_frame.instr[4] = API_BC_INSTR_TRANSFER;
ApiCmdBCFrameDef(ulModHandle,0,&api_bc_frame);

/* Minor Frame 2 Transfer sequence */
api_bc_frame.id       = 2;                          // Minor frame Identifier
api_bc_frame.cnt      = 1;                          // Instructions count
api_bc_frame.xid[0]   = 1;                          // Transfer 1
api_bc_frame.instr[0] = API_BC_INSTR_TRANSFER;
ApiCmdBCFrameDef(ulModHandle,0,&api_bc_frame);
```

Define Major Frame:

```
/* Minor Frame sequence in Major Frame */
api_bc_mframe_ex.cnt     = 2;    // Number of Minor Frames
api_bc_mframe_ex.fid[0] = 1;    // Minor Frame 1
api_bc_mframe_ex.fid[1] = 2;    // Minor Frame 2
ApiCmdBCMFrameDefEx(ulModHandle,0,&api_bc_mframe_ex);
```

6.5.2 BC Instruction Table Mode

In BC Instruction Table mode, the number of minor frames and major frames supported is limited only by the amount of memory allocated for the BC Instruction Table list. In this mode, the user defines both minor and major frames within one Instruction Table (*ApiCmdBCInstrTblGen*). Each minor frame defined in the Table is associated with a user-defined label. If you are using BC Instruction Table mode to setup your minor/major frames, all programmed actions the BC is to take is to be entered into the BC Instruction Table manually using the appropriate BC Instruction mode functions defined below.

There are two API S/W Library function calls associated with the definition of the minor/major frames and minor frame timing for this mode including:

- ***ApiCmdBCInstrTblIni*** - Initializes the BC Instruction Table mode. This function internally calls *ApiCmdSysMemLayout* to obtain the size and start address of the BC Instruction List from the Global memory layout.

- ***ApiCmdBCInstrTblGen*** - Clears/Converts/Writes the BC Instruction Table which contains the minor frame(s) and the major frame information/definition.

  Clears - Initializes the BC Instruction Table to all zeros. (Zeros are considered No-Op instructions by the BC controller.)

  Converts - Converts the user-defined BC Instruction Table entries into an array of firmware instruction long-words (which are written into the BC Instruction list area of Global Memory using the Write command). Prior to the execution of this command the user must define the BC Instruction Table entries using *TY_API_BC_FW_INSTR* structure for each BC Instruction Table entry. BC Instruction Table entries include the following:

  – Label - a unique user-defined 16-Bit value used as address

  – Firmware Operational code (Opcode) - instructions to be executed within the minor fame. These instructions consist of one or more of the codes shown in the following table:

Constant	Description
API_BC_FWI_XFER	Execute BC Transfer (defined using *ApiCmdBCXferDef*)
API_BC_FWI_CALL	Call Subtable - allows you to jump to a subtable/minor frame definition identified with a label.
API_BC_FWI_RET	Return from Subtable - used to return to the main BC Instruction Table entry following the related *API_BC_FWI_CALL* opcode.
API_BC_FWI_JUMP	Jump to Subtable / Instruction - absolute jump to an Instruction Table entry identified by a label.
API_BC_FWI_SKIP	Relative Branch (Skip) - skip a user-specified number of instructions
API_BC_FWI_WTRG	Wait for BC Trigger input - ties the external pulse to start of minor frames, or starts execution of the major frame based on the external pulse
API_BC_FWI_STRB	Strobe BC Trigger output - instruction to output a strobe signal
API_BC_FWI_DJZ	Decrement and Jump on Zero - When using non-cyclic major frame (as specified in *ApiCmdBCStart*) you can jump to a label in the BC Instruction Table when the major frame counter is decremented to zero.
API_BC_FWI_WMFT	Wait for next Minor Frame Time slot - instruction to wait until the next minor frame time slot begins, then continue with the following entry in the BC Instruction Table
API_BC_FWI_HALT	BC Operation Halt - Halt the BC
API_BC_FWI_DELAY	Delay - delay the execution of the next entry in the BC Instruction Table by a user-specified time
API_BC_FWI_CMFT	Change Minor Frame Time - instruction to change the minor frame time "on-the-fly" to a user-specified value.
API_BC_FWI_RESMF	Reset Major Frame - instruction to swap between several different major frames in one BC setup.

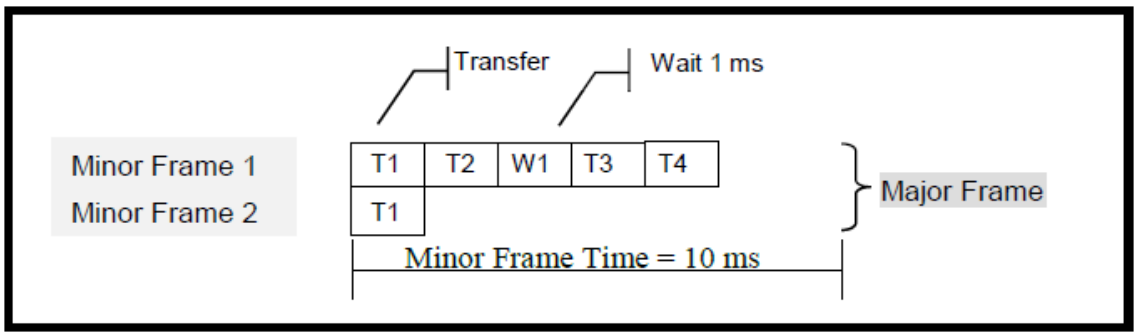
  – Instruction parameters - parameters required for the Firmware Instruction

  Writes - Writes the converted BC Instruction Table to the Instruction List global memory area.

Two other API S/W Library functions can be called in order to read the contents of the Global memory BC Instruction List including: *ApiCmdBCInstrTblGetAddrFromLabel* which obtains the address of the BC Instruction List for a user-specified label, and *ApiReadBlockMemData* to read the Global Memory's BC Instruction List at the address returned by *ApiCmdBCInstrTblGetAddrFromLabel*.

The following example defines minor/major frames using the BC Instruction Table mode. This sample code demonstrates the setup of minor and major frames and the starting of the BC using 4 previously

setup transfers with Transfer IDs of 1 through 4. The first minor frame includes 5 instructions; (1) execute Transfer 1, (2) execute Transfer 2, (3) execute a wait delay of 1 millisecond, (4) execute Transfer 3, (5) execute Transfer 4. The second minor frame has only one instruction; (1) execute Transfer 1. (Note: This is the same minor/major frame architecture setup as defined in the example for Standard Framing mode in Section 6.5.1.)



Define Minor/Major Frames:

```
TY_API_BC_FW_INSTR instr_tbl[7];
AiUInt32 ctbl[7];

/* Get BC resources by initializing to BC Instruction Table mode*/
ApiCmdBCInstrTblIni(ulModHandle,0);

/* Clear BC Instruction Table structure */
ApiCmdBCInstrTblGen(ulModHandle,0,
                    0,              // mode=CLEAR
                    8,              // # of entries
                    0L,             // dest_cnt
                    0L,             // dest_offset
                    instr_tbl,      // instructions array
                    ctbl,           // the converted table in firmware format
                    &line,
                    &rval);

/* Setup BC Instruction Table */
instr_tbl[0].label  = 0x1111;               // Label for first minor frame
instr_tbl[0].op     = API_BC_FWI_XFER;
instr_tbl[0].par1   = 1;                    // Transfer 1
instr_tbl[1].op     = API_BC_FWI_XFER;
instr_tbl[1].par1   = 2;                    // Transfer 2
instr_tbl[2].op     = API_BC_FWI_DELAY;
instr_tbl[2].par1   = 4000;                 // 1 ms delay in 250ns increments
instr_tbl[3].op     = API_BC_FWI_XFER;
instr_tbl[3].par1   = 3;                    // Transfer 3
instr_tbl[4].op     = API_BC_FWI_XFER;
instr_tbl[4].par1   = 3;                    // Transfer 4
instr_tbl[5].op     = API_BC_FWI_WMFT;      // Wait for Next Minor Frame Time Slot
```

```
instr_tbl[6].label   = 0x2222;              // Label for second minor frame
instr_tbl[6].op      = API_BC_FWI_XFER;
instr_tbl[6].par1    = 1;                   // Transfer 1
instr_tbl[7].op      = API_BC_FWI_WMFT;     // Wait for Next Minor Frame Time Slot
Instr_Tbl[8].op      = API_BC_FWI_JUMP;     // Setup this major frame to be cyclic
Instr_Tbl[8].par1    = 0x1111;              // by jumping back to the first min frame

// Convert and Write BC Instruction Table to memory image in ctbl
ApiCmdBCInstrTblGen( ulModHandle,0,
                     3,                     // mode=CONVERT & WRITE
                     9,                     // # of entries
                     8,                     // dest_cnt
                     0L,                    // dest_offset
                     instr_tbl,             // instructions array
                     ctbl,                  // the converted table in firmware format
                     &line, &rval);
```

Note:
If you want to setup a cyclic major frame then the last instruction has to be *API_BC_FWI_JUMP* (back to the start of the major frame).

6.6 Starting the Bus Controller

The BC can be started after the minor/major frame definition has been performed as defined in Section 6.5. The function, *ApiCmdBCStart*, starts the execution of pre-defined BC transfers/instructions and defines minor frame timing as defined below:

- **Minor Frame Time** - the time allotted by the BC for each minor frame transmission. Figure 6.6.0-I shows how the Minor Frame Time parameter is used by the BC to control the timing of the minor frames. In essence, the timing starts at the beginning of the minor frame, the minor frame transfers/instructions are executed back-to-back then the BC waits for the Minor Frame Time (specified by the user) to expire

- **Bus Controller Start Mode** - Defines the method the BC will use to begin the transmission of the major/minor frames including one of the modes as defined in the table below. The Autoframing method, as shown in Figure 6.6.0-II, of inserting minor frames into the major frame is used for all start modes, with the exception of *API_BC_START_EXTERNAL_PULSE*. Figure 6.6.0-III and Figure 6.6.0-IV show two scenarios using an external pulse to frame the minor frames. The first scenario Figure 6.6.0-III shows how the rising edge of the pulse controls the transmission of the minor frame. The second scenario adds a wait instruction (using the *ApiCmdBCFrameDef* function) as the first instruction the BC executes after the rising edge of the pulse is detected.

Constant	Description
API_BC_START_IMMEDIATELY	Start BC operation / Transfer execution immediately
API_BC_START_EXT_TRIGGER	Start BC operation / Transfer execution by external BC trigger input
API_BC_START_RT_MODECODE	Start BC operation / Transfer execution by RT Modecode Dynamic Bus Control
API_BC_START_EXTERNAL_PULSE	Drive minor framing from external pulse (BC trigger input)
API_BC_START_SETUP	Prepare BC Instruction sequence (= Minor and Major Frame sequence) in the Board Global memory for a fast start with *API_BC_START_FAST*.
API_BC_START_FAST	Start BC operation / Transfer execution immediately fast. Note that *API_BC_START_SETUP* and *API_BC_START_FAST* shall be used in conjunction.
API_BC_START_INSTR_TABLE *	Start BC operation / Transfer execution with predefined BC Instruction Table (refer to *ApiCmdBCInstrTblGen*)
API_BC_START_CONTINUE_ON_BC_HALT *	Continue BC operation (only applicable with predefined BC Instruction Table, BC operation must be already started with *API_BC_START_INSTR_TABLE*)

Note:
Constants marked with * are applicable to BC Intruction Table Mode. All other modes are applicable for Standard Framing Mode.

- **Count** of times the major frames will execute. The user can specify 0 for in order to continuously execute the major frame (in Standard Framing mode only), or the user can specify a major frame count to indicate the number of times the major frame will be transmitted.

Figure 6.6.0-I: Minor Frame Timing Control Diagram

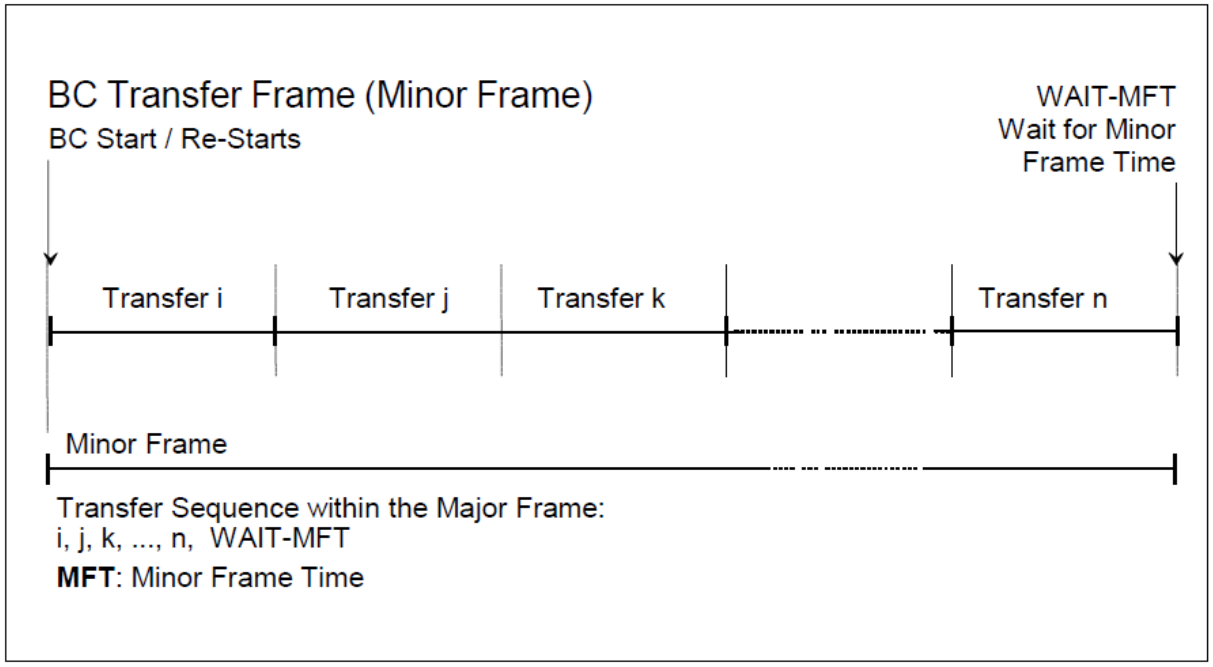


Figure 6.6.0-II: Minor Frames within the Major Frame using Autoframing

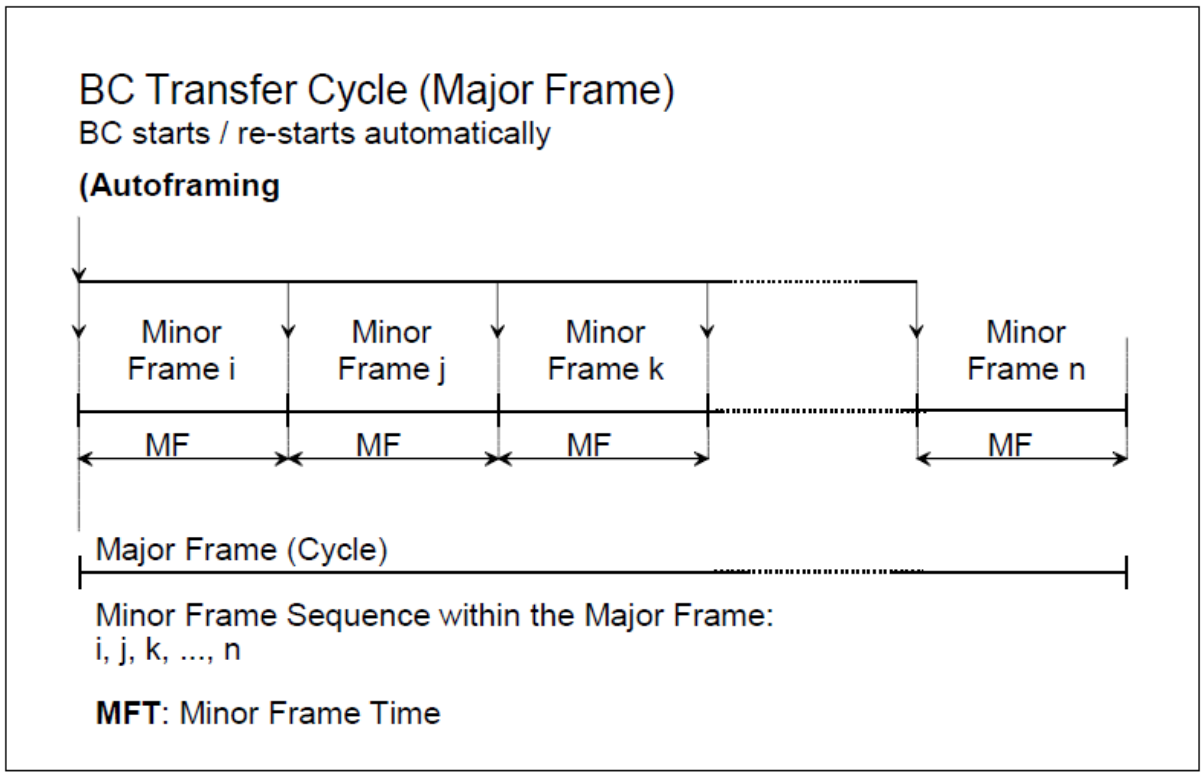


Figure 6.6.0-III: Minor Frames within the Major Frame using External Pulse

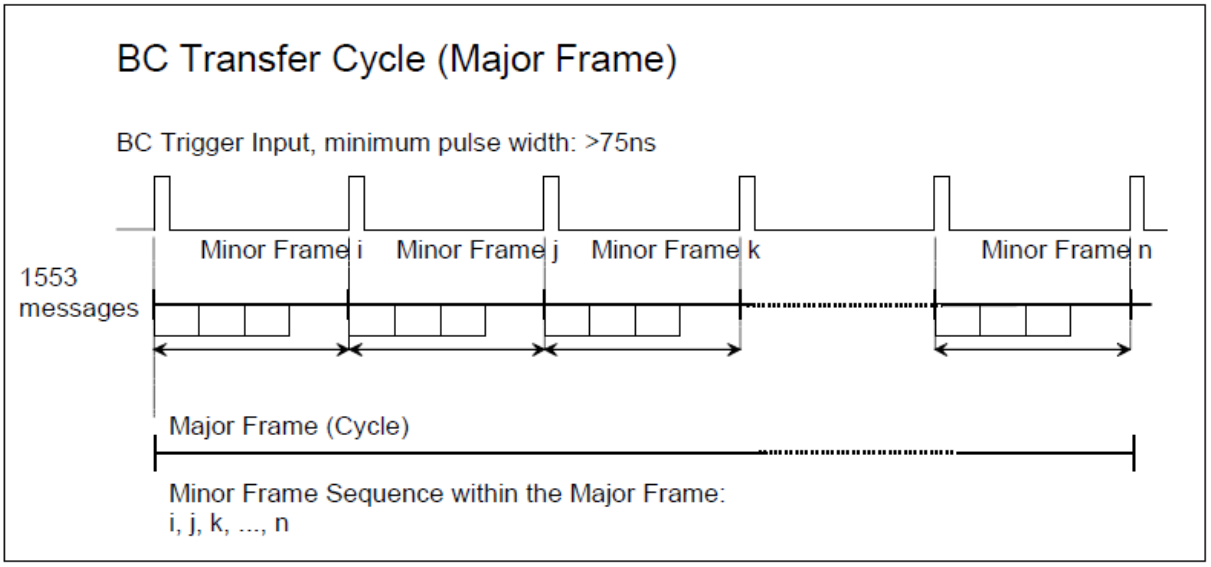
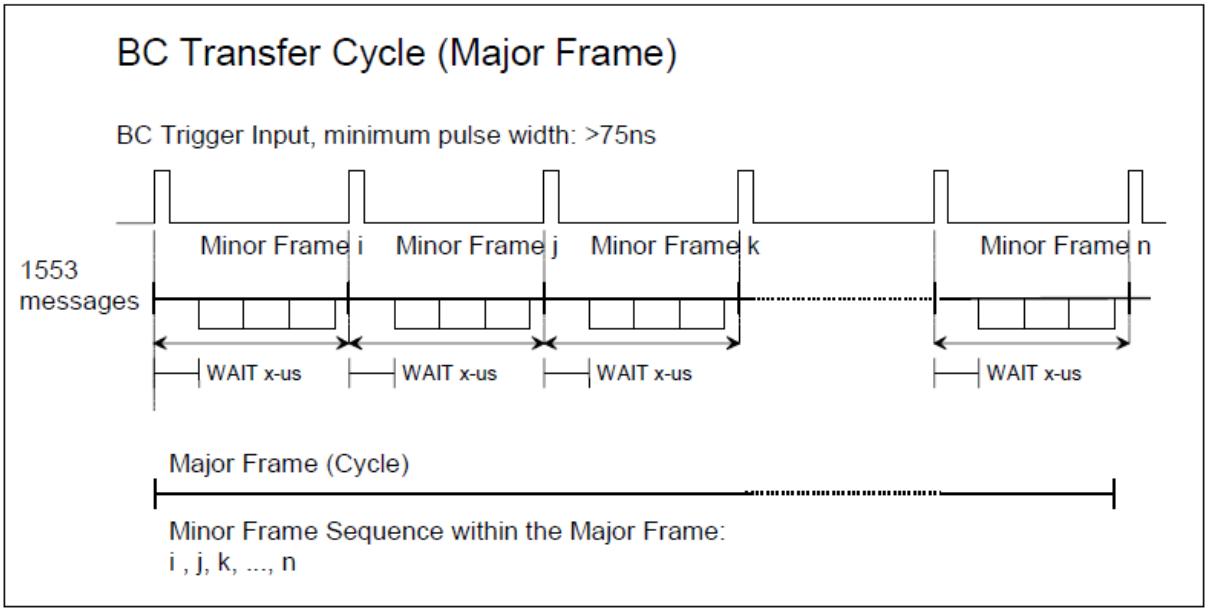


Figure 6.6.0-IV: Minor Frames within the Major Frame using External Pulse and a Wait Instruction



Following are three examples of starting the BC; one for Standard Framing mode, and the other two for BC Instruction Table mode. In each example, the minor frame time will be set to last 10 milliseconds.

Define Minor Frame Timing and Start the Bus Controller (for Standard Framing Mode)

```
frame_cnt   = 0;     // 0 = cyclic or any number of major frame executions
frame_time  = 10;    // Minor Frame Time in milliseconds
ApiCmdBCStart( ulModHandle,0,
               API_BC_START_IMMEDIATELY,   // Start standard framing immediately
               frame_cnt,                  // cyclic
               frame_time,                 // frame time in ms
               0,                          // start address n/a in this mode
               &ul_MajFrameAddr,           // output value
               aul_MinFrameAddr);          // output value
```

When using BC Instruction Table mode, only the *API_BC_START_INST_TABLE* or *API_BC_START_CONTINUE_ON_BC* start modes are applicable. In the example below, the execution of the BC Instruction Table will start at the beginning of the table.

Define Minor Frame Timing and Start the Bus Controller (for BC Instruction Table Mode)

```
frame_cnt    = 0;  // Cab be used for BC Instruction Table mode API_BC_FWI_DJZ
frame_time   = 10; // Minor Frame Time in milliseconds
ApiCmdBCStart( ulModHandle,0,
               API_BC_START_INSTR_TABLE,    // Start instruction table mode
               frame_cnt,                   // N/A*
               frame_time,                  // frame time in ms
```

```
                0,                          // BC Instr Tbl start address
                &ul_MajFrameAddr,           // output value
                aul_MinFrameAddr);          // output value
```

Note:
To create cyclic execution in BC Instruction Table mode, the programmer must include the *API_BC_FWI_JUMP* instruction at the end of the BC Instruction Table major frame using the *ApiCmdBCInstrTblGen* function. The *API_BC_FWI_DJZ* instruction can be used in combintion with frame_cnt to implement a cyclic framing that stops after a given number of repetitions.

To start execution from a different instruction in the BC Instruction, the instruction must have a label defined. You would then need to get the address of the label (using *ApiCmdBCInstrTblGetAddrFromLabel*), and use that address in the *ApiCmdBCStart* function call as shown below. (This example uses the BC Instruction Table minor frame/major frame definition example shown in Section 6.5.2.)

Define Minor Frame Timing and Start the Bus Controller (for BC Instruction Table Mode with execution start at a label other than the first label in the table.)

```
frame_time = 10; /* Minor Frame Time in milliseconds */
ApiCmdBCInstrTblGetAddrFromLabel( ulModHandle, 0,
                                  0x2222      // label for 2nd minor frame,
                                  8           // # of entries */,
                                  instr_tbl,
                                  &saddr,     // output address of label
                                  &line);

ApiCmdBCStart( ulModHandle,0,
               API_BC_START_INSTR_TABLE,  // Start in instruction table mode
               0,                         // frame_cnt N/A
               frame_time,                // frame time in ms
               saddr,                     // address of lable 0x2222
               &ul_MajFrameAddr,          // output value
               aul_MinFrameAddr);         // output value
```

6.7 Acyclic 1553 Transfers

Acyclic 1553 transfers may be required in your application if there are certain conditions under which you may want to insert a sequence of transfer(s) "on the fly" (or at a pre-defined time) into the major frame. The new acyclic frame containing the sequence of transfers will be inserted into the major frame by the BC one time upon the completion of the BC transfer/instruction currently being executed. In order to prepare for the "on the fly" insertion, you must first have the transfers defined as described in Section 6.2. The following additional functions must be used including:

- **ApiCmdBCAcycPrep** - Defines the properties of the acyclic "on-the-fly" BC transfers/instructions to be inserted into the BC framing sequence. This instruction is basically identical to the *ApiCmdBCFrameDef* without the definition of a Minor Frame ID (since it will be sent "on the fly").

- **ApiCmdBCAcycSend** - Starts the insertion of the acyclic "on-the-fly" transfers using one of the following methods:
  - Insert immediately into the BC framing sequence upon transmission completion of the current BC transfer/instruction
  - Insert at a user-specified-time *
  - Insert after the current normal minor frame is completed and before the next normal minor frame starts. *

Note:
* Check Appendix B of the Reference Manual for a detailed overview of functions supported for different devices.

Examples of acyclic transfer programming using the BC functions described above can be found in the *ls_acyclic_sample.c* BSP sample. The following sample code demonstrates the insertion of two transfers immediately upon request using two previously setup transfers with Transfer IDs of 1 and 2. Once normal BC operations have started, the acyclic transfer can be sent.

Define the transfers that will be sent in the acyclic frame. A maximum of 127 Transfers can be sent in one Acyclic Frame.

```
// Prepare Acyclic Transfer sequence
api_bc_acyc.cnt      = 2;                          // # of instructions in frame
api_bc_acyc.xid[0]   = 1;                          // Transfer 1
api_bc_acyc.instr[0] = API_BC_INSTR_TRANSFER;      // Instruction type transfer
api_bc_acyc.xid[1]   = 2;                          // Transfer 2
api_bc_acyc.instr[1] = API_BC_INSTR_TRANSFER;      // Instruction type transfer
ApiCmdBCAcycPrep( ulModHandle,0, &api_bc_acyc );
```

***Wait until the BC has started execution ***

Send the Acyclic Transfer

```
ApiCmdBCAcycSend( ulModHandle,0, API_BC_ACYC_SEND_IMMEDIATELY, 0, 0 );
```

Note:
Acyclic transfers can be performed when using either the Standard Framing mode or the BC Instruction Table mode for minor/major frame definition. The function returns before all acyclic data has been sent!

6.8 BC Interrupt Programming

As introduced in Section 6.3.1, the BC is capable of producing interrupts upon the occurrence of certain events. Interrupt Handler(s) must be created to process the interrupts which occur after the BC has been started and an interrupt occurs. Some possible BC Interrupt Handler functions may include: (1) refilling the message buffer at the end-of-transfer interrupt, and/or (2) reporting transfer errors on a transfer

error interrupt. The functions required to setup BC interrupts and interrupt handler execution include the Library Administration and Initialization functions as defined in Section 4.9, and one or more of the BC function calls (as your application requires) defined as follows:

- ***ApiCmdBCXferDef*** - Setup the BC to perform an interrupt on any of the following conditions:
  - Interrupt on End of Transfer
  - Interrupt on Transfer Error
  - Interrupt on Status Word exception
- ***ApiCmdBCFrameDef*, *ApiCmdBCAcycPrep*, or *ApiCmdBCInstrTblGen*** - Within the sequence of transfer instructions defined within a minor frame, you can setup an instruction to generate an interrupt.
  - Interrupt upon occurrence of a transfer instruction defined as BC Skip instruction with interrupt enabled
- ***ApiCmdBCStart*** – when setup for non-cyclic major frame transmission
  - Interrupt after the user-specified count of major frames have been transmitted and BC is halted.

Once you have configured the BC to generate an interrupt and you have created an Interrupt Handler function to handle the interrupt, then start the BC using the *ApiCmdBCStart* function to start data flow. If the BC determines that an interrupt has occurred, the BC will initiate a call to the Interrupt Handler function and provide information about the interrupt as defined in the *ApiInstIntHandler* handler definition in the associated Reference Manual.

The following sub-sections describe how to setup the BC transfer to create an interrupt.

6.8.1 How to Setup the 1553 Transfer to Cause an Interrupt

As described above, the BC can be setup to interrupt during a transfer on one of the following conditions:

- Interrupt on End of Transfer - an interrupt will be logged to the interrupt loglist when the transfer is complete (including status word/frame)
- Interrupt on Transfer Error - an interrupt will be logged to the interrupt loglist when a transfer error is detected
- Interrupt on Status Word exception - an interrupt will be logged to the interrupt loglist when the Staus Word received by the BC AND'd with the Status Word Exception Mask indicates a non-zero condition. When configuring your transfer to interrupt on a Status Word Exception, you must set the Status Word Exception Mask to indicate the Status Word Exception the BC is to look for. See Figure 6.8.1-I for details.

The following table shows some examples on how to enable different interrupt conditions within the transfer descriptor. In the Status Word Exception Interrupt Transfer defined, the Status Word Exception Mask is setup to look for a "Busy" condition in the Status Word:

BC Transfer Definition Parameters defined using *ApiCmdBCXferDef*			
Transfer structure	**End of Transfer Interrupt**	**Transfer Error Interrupt**	**Status Word Exception Interrupt**
tic	API_BC_TIC_ON_XFER_END	API_BC_TIC_INT_ON_XFER_ERR	API_BC_TIC_ON_STATUS_EXCEPT
sxwm			0x0008 (See Figure 6.8.1-I)
rsp			API_BC_RSP_AUTOMATIC

Figure 6.8.1-I: Status Word Exception Mask

Status Word Exception Mask									
15.....11	**10**	**9**	**8**	**7....5**	**4**	**3**	**2**	**1**	**0**
5	1	1	1	3	1	1	1	1	1
Remote Terminal Address	Message Error	Instrumentation	Service request	Reserved	Broadcast Command received	Busy	Sub System Flag	Dynamic Bus Control Acceptance	Terminal Flag

Section 5.2.1 contains an example on how to program the BC interrupt to reload the transmit buffer data.

6.8.2 How to Setup the Minor Frame Transfer Sequence to Cause an Interrupt

Although used less frequently, the BC can be setup to generate an interrupt between transfers in a minor frame or acyclic frame. This can be accomplished using the functions *ApiCmdBCFrameDef* (for minor frame transfer sequence definition in Standard Framing mode), *ApiCmdBCInstrTblGen* (for minor frame transfer sequence definition in BC Instruction Table mode), and *ApiCmdBCAcycPrep* (for acyclic frame transfer sequence definition).

- For *ApiCmdBCFrameDef* and *ApiCmdBCAcycPrep*, the two function parameters associated with this setup include the transfer instruction, "inst", and the transfer ID "xid". These two parameters should be setup as follows:
  - inst - set to the Skip Instruction (inst = API_BC_INSTR_SKIP)
  - xid - set the Skip Count to 1 (which basically is interpreted as a "No operation" instruction) and enable the interrupt.
- For *ApiCmdBCInstrTblGen* the two function parameters associated with this setup include the transfer instruction, "op" and the instruction parameter "par2". These two parameters should be setup as follows:
  - op - set to the Skip Instruction (inst = API_BC_FWI_SKIP)
  - par2 - set INT = 1 to enable the interrupt.

Note:
Interrupts generated by the skip instruction are reported as API_INT_BC_BRANCH type in the interrupt callback handler.

6.9 Status Word Exception Handling

The BC can be programmed to monitor bits in the Status word received from an RT SA/Mode code. If any of the Status word bits (programmed to be monitored) are set in the Status word received from the RT SA/Mode code, the BC can also be programmed to halt the BC or generate an interrupt such that a user-developed interrupt handler can provide further processing.

There is one function associated with programming the BC to handle Status word Exceptions: *ApiCmdBCXferDef*. (Handling of the Service Request in the RT Status word is a special case and is discussed in Section 6.9.1) As defined in Section 6.2, the *ApiCmdBCXferDef* BC function will utilize the *TY_API_BC_XFER* structure to define the properties of a 1553 transfer. Information contained in this structure is not only used to create the Command word, Data word(s)/Mode Code, it is also used to define the process for handling the Status word generated by the RT(s) SA/Mode code associated with the transfer. The *TY_API_BC_XFER* structure parameters associated with Status word exception handling are shown in the following table.

BC Transfer Definition Parameters defined using *ApiCmdBCXferDef*	
Transfer structure	**Definition**
tic	**Transfer Interrupt Control** - setup for generation of a BC interrupt upon end of transfer, transfer error or status word exception (See Section 6.8)
hlt	**Halt** - setup to halt the BC upon end of transfer, transfer error, status word exception or any interrupt
sxh	**Status Word exception handling** - defines the process to execute upon occurrence of a Status Word Service Request
sxwm	**Status Word Exception Mask** - defines the bits in the status word received from the RT that will be checked by the BC
rsp	**Response** - defines the response the BC expects from the RT such as basic response as defined by *ApiCmdDefMilbusProtocol*, no status word 1 expected, or no status word 2 expected.

The Status Word Exception Mask (swxm) is used to mask (AND mask) the status word response of the RT SA/Mode code. The Status Word Exception Mask is used for both responses received by the BC for RT-to-RT transfers. If the result of the mask process is not zero, an interrupt is asserted, if enabled via the tic parameter and/or the BC is halted, if enabled via the hlt parameter. Unexpected responses (a non-zero result of the mask process) are not counted as errors and cause no Error Interrupt or Retry.

The table below provides some examples of how to setup the parameters for *ApiCmdBCXferDef* for Status word exception handling. Note that in each case the Expected Response control (rsp) is setup to check both Status Word 1 and Status Word 2 (RT-RT transfers only). This parameter, however, can be modified to check only Status word 1 or only Status word 2. In the following table, three examples are setup. The first example shows the parameter setup for configuring the BC transfer such that the BC will check the Message Error bit (bit 10) in the Status word and interrupt if that bit is set in the Status word received by the BC. The second example configures the BC to halt the BC if the Instrumentation error bit (bit 9) is set in the Status word. The third example configures the BC transfer such that the BC will interrupt if either the Message Error bit (bit 10) or the Busy bit (bit 3) is set in the Status word

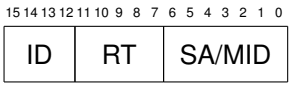
BC Transfer Definition Parameters defined using *ApiCmdBCXferDef*			
Transfer structure	**Interrupt on Message Error in RT Status Word**	**Halt the BC on Instrumentation Error in RT Status Word**	**Interrupt on Busy set or Message Error in RT Status Word**
hlt	API_BC_HLT_NO_HALT	API_BC_HALT_ON_STAT_EXCEPT	API_BC_HLT_NO_HALT
tic	API_BC_TIC_ON_STATUS_EXCEPT	API_BC_TIC_NO_INT	API_BC_TIC_ON_STATUS_EXCEPT
sxwm	0x0400 (See Figure 6.8.1-I)	0x0200 (See Figure 6.8.1-I)	0x0408 (See Figure 6.8.1-I)
rsp	API_BC_RSP_AUTOMATIC	API_BC_RSP_AUTOMATIC	API_BC_RSP_AUTOMATIC

6.9.1 RT Service Request Processing

The BC can be programmed to transmit a Vector Word Mode code (Mode code 16) when the Service Request bit in the RT Status word (Status Word 1 only) is set. Setup required for this process to occur includes the following:

- **ApiCmdBCIni** - Enable Service Request / Vector Word Mode Control (svrq)
- **ApiCmdBCXferDef** - Setup the Status word exception mask (swxm) with the Service Request bit set (bit 8). Configure the Status Word Exception Service Request handling control parameter (sxh = 1) for the BC to generate automatic Transmit Vector Word Mode code (16) when the Service Request bit is set.

As response on the Mode code, the RT transmits a Vector Word to the BC, which is interpreted and handled by the BC following the description given in the 'AVS Databus Usage Report R-J-403-V-1209 Par7.2'. The format of the Vector Word expected to be sent by the RT after receipt of the Transmit Vector Word Mode code is shown in the following bit field.



- **ID** [Bit 15 - 12] Vector Word ID
- **RT** [Bit 11 - 7] RT-Address
- **SA/MID** [Bit 6 - 0] Subaddress / MID.

The Vector Word IDs and BC automatic response is shown in the following table. In order for the BC to respond as requested by the RT's Vector Word, a BC transfer must be setup as defined in the last column of next Table.

BC Response to RT Vector Word		
Vector Word ID	**BC Response**	**BC Setup Required**
Request Single RX/TX (1000/1001)	A pre-defined transfer is executed once immediately (acyclic) after receipt and decoding of the vector word.	*ApiCmdBCXferDef* - setup a vector word driven transfer for this request with parameter sxh = 1.

Continued on next page

Continued from previous page

Vector Word ID	BC Response	BC Setup Required
Request Multiple RX/TX (1010/1011)	A pre-defined transfer is appending to the end of the current BC minor frame (Cyclic execution until disabled).	*ApiCmdBCXferDef* - setup a vector word driven transfer for this request with parameter sxh = 2
Delete RX/TX (1100/1101)	The related BC transaction is disabled by deleting it from the end of the related BC minor frame. Re-enabling requires receipt of the corresponding vector word code (Request Multiple RX/TX 1010/1011). Only BC transfers which have been enabled by a previous 'Request Multiple RX/TX' vector word can be disabled.	

Note:
Only BC-RT and RT-BC transfer types can be defined to be 'vector word driven transactions'.

The BC function *ApiCmdBCSrvReqVecStatus* can be used to obtain the status of the number of Service Requests issued by the RT and to obtain the Last Vector Word received from the RT.

7 Remote Terminal Programming

The remote terminal (RT) is a device designed to interface various subsystems with the 1553 data bus. The interface device may be embedded within the subsystem itself, or may be an external interface to tie a non-1553 compatible device to the bus. As a function of the interface requirement, the RT receives and decodes commands from the BC, detects any errors and reacts to those errors. The RT must be able to properly handle both protocol errors (missing data, extra words, etc.) and electrical errors (waveform distortion, rise time violations, etc). RTs are the largest segment of bus components. Up to 31 remote terminals can be connected to the data bus and each remote terminal can have 31 subadresses. The remote terminal shall not speak unless spoken to first by the bus controller and specifically commanded to transmit. The commands may include data or request for data (including status) from RTs. Command word, Data word and Status word formats are shown in section 2.

If your application requires the simulation of an RT, this section will provide you with the understanding of the RT programming functions required for use in your application. Programming the RT may require the use of more than just the RT functions. Also needed may be the Buffer and FIFO functions. This section will discuss some of the typical scenarios a programmer would encounter that would require the use of the Buffer, FIFO and RT functions. These scenarios include:

- Initializing the RT
- Defining RT Subaddresses
  - RT transfers using single buffers
  - RT transfers using multiple buffers
  - RT transfers using FIFOs
  - RT transfers with dynamic data
- RT transfers with error injection
  - Into Status/Data word
  - Via Status Word Response
- RT Interrupt programming

7.1 Initializing the RT

Initialization of the RT must be the first RT setup function call issued. This function call, *ApiCmdRTIni*, required for each RT to be initialized, will perform setup of the RT global transfer and Status word response characteristics for a specified RT including:

- **RT Address** - Address (0 - 31) of the simulated RT
- **RT Operation Control** - This control parameter configures the RT for one of the following two operational modes:
  - RT Simulation - In this mode the RT will behave in the manner you have set it up to behave.

  - RT Mailbox Monitoring - In this mode, the RT will capture RT message data on a subaddress level without affecting bus traffic (i.e. without generating a response on the bus). This mode is used to monitor non-simulated "external" RT's.

- **Response Time** - defines the default value defining the time it will take for the RT to respond with a status word. As shown in Figure 2.0.0-III, the response time is the time between the BC Command/Data word and the RT Status word. This value can be programmed from 4.0µs to 63.75µs in 0.25µs steps.

  Note:
  There is an additional Response Time configuration function, *ApiCmdRTRespTime*, that allows you to change the defaulted response time after basic initialization has been performed.

  Note:
  Not all boards support response times down to 4.0 µs. Check the Reference Manual for restrictions on various board types.

- **RT Next Status Word** - defines the value of the status word that the RT will respond with after the BC Command/Data word is received. See figure 2.0.0-II for a description of the status word.

The following code example uses API S/W Library constants to initialize the RTs, 1, and 2 for Simulation enabled, Response time default set to 8µs, and the Next Status word is setup to disable any errors or service requests. (The only bits that are set in the Next Status word indicate the RT address.)

```
// Remote Terminal Initialization Commands
ApiCmdRTIni(ulModHandle, 0, 1/*RT*/, API_RT_ENABLE_SIMULATION, 0, 8, 0x0800);
ApiCmdRTIni(ulModHandle, 0, 2/*RT*/, API_RT_ENABLE_SIMULATION, 0, 8, 0x1000);
```

Note:
Default operation of the RT is to respond to Command Words from both the primary and secondary busses.

7.2 Defining RT Subaddress/Mode Code for Communication with the BC

For MIL-STD-1553 BC-to-RT, RT-to-RT and RT-to-BC transfers, the user must first assign an RT Buffer Header ID to the RT to enable the processor to identify the location of status and event data generated for each transfer. The status and event data queues associated with the assigned Buffer Header will contain the command and status information required to process the 1553 transfer as well as the pointers to the buffers used to transfer the 1553 Data Words.

The message buffers will be assigned to transmit/receive the Data words within the 1553 transfer. If there are no Data words within the 1553 transfer, a message buffer will still need to be assigned. (However, the API S/W Library does not prevent the user from using the same buffers in more than one transfer, therefore, the same message buffer can be assigned for transfers that do not require the transmission/reception of Data word(s)). Each message buffer has a unique ID which must be assigned by the user. See section 5.1 for details about the buffer handling.

In addition, the characteristics of the RT SA must be defined.

To configure an RT to respond to each type of transfer will require the use of the following RT function calls:

- **ApiCmdRTBHDef** - this RT function will define an RT Buffer Header structure. As shown in Figure 5.1.0-I, the RT Buffer Header structure enables the processor to identify the location of the message buffers used and status and event data generated for each RT message transfer. The RT Buffer Header information to be setup by this function includes the following:
  - **RT Buffer Header ID** – the ID of the RT Buffer Header structure.
  - **Message Buffer ID** - the ID of the first buffer in the Global RAM message buffer pool to be used for the transfer of the Data words. A Message Buffer ID of 20 would indicate that the 20th buffer in the Global RAM Buffer Pool will be used. See Figure 5.1.0-I for a diagram of the structure of these message buffers.
  - **Buffer Queue Size** - the number of Global RAM message buffers to be used for the transfer of the Data words. One or more buffers can be used for the transfer. You will always need to assign at least one message buffer from the Global RAM Buffer Pool for your transfer. For example, assigning API_QUEUE_SIZE_8 for this transfer indicates that 8 contiguous buffers would be used. Using the example of Message Buffer ID of 20 above, and a queue size of 8, message buffers 20 - 27 would be used for the transfer.
  - **Buffer Queue Mode** - specifies the order in which multiple buffers as specified in the Buffer Queue Size, will be filled. In most cases, users will choose to store the Data words into the Message Buffers in a cyclic fashion (API_BQM_CYCLIC).
  - **Data Buffer Store Mode** - will allow the user to indicate the actions to be taken in case of error in the transmission or reception of Data words such as whether to keep transmitting the same message buffer at transfer error, or continue with the next buffer in the queue for the next transmission.

Note:
Buffer Queue Size, Buffer Queue Mode , Data Buffer Store Mode, and Buffer Size and the Current Buffer Index can be modified "on the fly" (i.e. after the RT has been started) using the Buffer function call *ApiBHModify*.

- **ApiCmdRTSACon** - this function defines the subaddresses to be simulated and the mode codes the RT is required to respond to. This enables the RT to be able to properly respond to the Command/Action word with a user-defined Status word and/or Data word Transmission. For each of the RT's Subaddresses or Mode codes simulated by your application you will need to use this function call. All RT characteristics defined by this function are associated with the Buffer Header ID identified with *ApiCmdRTBHDef*. This function provides
  - **Subaddress Type** – allows the user to define whether the Subaddress specifed by the user (sa) (0 – 31) is enabling response functionality for a Mode code or is enabling a subaddress for data transfers.
  - **Subaddress Control** - provides a method to disable the individual SA, and setup interrupt processing including whether the RT SA will interrupt after the transfer is complete or interrupt on any transfer error.
  - **Buffer Locations** - It also associates the RT SA/Mode code to a pre-defined RT Buffer Header ID (see *ApiCmdRTBHDef* description above) such that the buffer location for all action (Data word receive/transmit), and status for that RT SA/Mode code is known.

– **Status Word Response** - provides the capability for a unique Status Word Response to be sent for each individual RT SA's or Mode codes. The Status Word response is defined by masking in any desired Status word bits to the Next Status Word defined using the global RT function, *ApiCmdRTIni*.

The following code is an example of setting up the RT for a BC-to-RT transfer to RT1, SA1 with a data word count of 4. The IDs assigned to this transfer include:

RT Assignments
RT Buffer Header ID = 1
Buffer ID = 10

This setup performed in this example will configure RT1, SA1 to receive Data words into a message buffer (buf_id = 10). With the *ApiCmdRTBHDef* function, the Queue size is set to 1, such that the same buffer will be used each time the RT receives Data words from the BC. Using the *ApiCmdRTSACon* function, RT1 Receive SA1 is associated with RT Header ID = 1, the subaddress is enabled with no interrupt (*API_RT_ENABLE_SA*), the Status Word Mask control and mask is set to logically "OR" the value of 0 with the Next Status word specified in the *ApiCmdRTIni* function. (In essence, the RT1 Receive SA1 will use the default Next Status word instead of defining a unique Status word.)

```
rt_hid = 1; buf_id = 10;

ApiCmdRTBHDef(ulModHandle, 0,
              rt_hid,
              buf_id,
              0, 0,
              API_QUEUE_SIZE_1,
              0, 0, 0, 0, 0,
              &api_rt_bh_desc );

ApiCmdRTSACon(ulModHandle, 0,
              1/*RT*/, 1/*SA*/,
              rt_hid/*HID*/,
              API_RT_TYPE_RECEIVE_SA,
              API_RT_ENABLE_SA,
              0/*rmod*/,
              API_RT_SWM_OR,
              0/*swm*/);
```

The following table provides examples of how to change the parameters in the *ApiCmdRTSACon* function to setup different types of RT transfers. The BSP sample folder contains many examples of setting up multiple RTs as well.

ApiCmdRTSACon	RT	SA/MC	SA Type
Broadcast Xfer with word count of 2	31	1 - 30	RECEIVE_SA
RT2 SA3 to BC Xfer with any word count	2	3	TRANSMIT_SA
Mode code 17 Xfer to RT05 SA0 (synchronize)	5	17	RECEIVE_MODECODE

Note:
If you want to setup the RT such that it will detect illegal Mode codes, you will need to configure the RT using the *ApiCmdRTSACon* function for every illegal Mode code and set the RT's Status word to indicate Message Error. See Section 7.3.2

7.2.1 RT Transmit/Receive Message Data Word Generation/Processing

Now that you are familiar with the method used to define the characteristics of the RT transfers generated by a simulated RT, we can now discuss how to setup and place data into the message buffers assigned for RT Data word transmissions for RT-to-BC and RT-to-RT type transfers. Also described will be how to setup and obtain data from the message buffers used for RT Data word reception for BC-to-RT and RT-to-RT type transfers.

7.2.2 For RT-to-BC and RT-to-RT Type Transfers (RT Transmit)

The API S/W Library provides several methods to insert real-time/dynamic/fixed user data into the Global RAM 1553 Message Buffers used by the transmitting side of the RT transfer and specified in the *ApiCmdRTBHDef* function described in the previous section. Regardless of how many buffers have been allocated in the Global RAM message buffer pool for the transmission of RT data, any of the following methods can be used to insert Data words into the Message Buffer(s). Following is a summary of the different scenarios you may choose:

- Insert fixed data into the Message Buffer(s) using the *ApiCmdBufDef* Buffer function.
- Insert dynamically updating data into the Message Buffer(s) using the *ApiCmdRTDytagDef* function. This function provides dynamically changing data words within the transmit message buffer using one of the following dynamic data generation modes/functions:
  - **Function mode:** provides for up to two dynamic data words per transfer using any combination of the following functions as pictured in Figure 6.3.1-I.
    1. Positive Ramp
    2. Negative Ramp
    3. Positive Triangle
    4. Negative Triangle
    5. Transmit a Data word from a different Message Buffer
  - **Tagging mode:** provides for up to four dynamic data words per transfer using Sawtooth Tagging. The Sawtooth Tagging mode provides an incrementer by 1, which is performed on the user-specified location with each execution of the associated RT transfer. The location to be incremented can be setup with an initial value to be incremented each transfer, or the existing value can be incremented. The options are to increment any combination of the following byte or words:
    1. 16-Bit Sawtooth
    2. 8-Bit Sawtooth LSB (lower byte of word)
    3. 8-Bit Sawtooth MSB (higher byte of word)

- Assign a FIFO (ASP Shared RAM) to the transfer. (Each FIFO consists of 128 32-word buffers. There are 32 FIFOs.) Pre-fill the FIFO with application data, the data transmitted in the Global RAM message transmit buffer will be obtained from the FIFO. Re-fill the FIFO as needed. (See Figure 6.3.1-II)
- Assign Dataset buffers (ASP Shared RAM) to the transfer. (There are 4095 32-word buffers in the Dataset Buffer pool.) Pre-fill the Dataset buffer(s) with application data, the data transmitted in the Global RAM message transmit buffer will be obtained from the Dataset buffers. Refill the Dataset buffers as needed.
- Create an interrupt handler routine and setup the interrupt control defined for the transfer to interrupt on end-of-transfer. Upon end-of transfer interrupt, the interrupt handler will be called at which time the buffer can be filled with new data. See Section 7.4 for more information on RT Interrupt handling.

The BC, Buffer, and/or FIFO function calls required for each scenario described above are summarized in the following table.

Data Source	Functions Used	BSP Examples
Fixed Data	1. *ApiCmdRTBHDef* 2. *ApiCmdRTSACon* 3. *ApiCmdBufDef*	LS BC RT BM Sample
With Dynamic Data Words	1. *ApiCmdRTBHDef* 2. *ApiCmdRTSACon* 3. *ApiCmdBufDef* to setup non-dynamic data 4. *ApiCmdRTDytagDef* to setup 1-4 dynamic data words	Ls Dynamic Data Sample
Using FIFOs	1. *ApiCmdRTBHDef* 2. *ApiCmdRTSACon* 3. *ApiCmdFifoIni* to initialize the FIFO 4. *ApiCmdRTSAAssignFifo* to assign the FIFO to the transfer 5. *ApiCmdFifoReadStatus* to determine how much FIFO data to reload. 6. *ApiCmdFifoWrite* to fill the FIFO with data	Ls FIFO Sample
With Dynamic Dataset Buffers	1. *ApiCmdRTBHDef* 2. *ApiCmdRTSACon* 3. *ApiCmdRamWriteDataset* to fill the dataset buffers with data to be used in the message buffers 4. *ApiCmdSystagDef* to assign the Dataset buffers to the transfer	Ls Dynamic Data Sample

7.2.3 For BC-to-RT and RT-to-RT Type Transfers (RT Receive)

Before the RT receives the Data words from the BC or an RT, as part of the setup process, you may choose to clear your receive buffer to avoid any Data word transmission confusion. The receive buffers

can be cleared before use by using the *ApiCmdBufDef* function.

After the RT has received Data words from the BC or an RT, software will need to be added to process the data. Processing data received by the RT can be accomplished in one of two ways: polling at pre-defined intervals to examine the RT data, or setting up the transfer to interrupt at end-of-transfer. To accomplish the interrupt on end-of-transfer method of processing the data, an interrupt handler routine must also be developed to handle the interrupt which will occur after all Data words have been received. Upon end-of transfer interrupt, the interrupt handler will be called at which time the buffer can be read and processed as required by the application. Interrupt handling is discussed further in Section 7.4.

7.2.4 RT Transmit/Receive Mode Code Generation/Processing

The RT can be programmed to respond to any Mode code command including the reserved Mode codes. As discussed in Section 7.2, enabling or disabling Mode codes is achieved using the *ApiCmdRTSACon* function. This function will allow the RT to process up to 32 receive and 32 transmit Mode codes.

Note:
No difference is made between received Mode code commands using the subaddress 0 or 31 if the RT operates in MIL-STD-1553B protocol mode. If the RT operates in MIL-STD-1553A protocol mode, only Subaddress zero is used for received Mode Commands.

If your application requires that the RT be capable of processing Mode codes, this section will provide further detail into this process. If the RT operates in the MIL-STD-1553B protocol mode, special Mode code handling is provided for the Mode codes as follows:

- **Transmit Status Word (00010)** - The Last Status word sent for each simulated RT is maintained by the onboard RT interface software. If a simulated RT is setup to receive this Mode code using the *ApiCmdRTSACon* function the RT interface will automatically send the Last Status word for the specific RT upon receipt of this Mode code. If the last transfer was a Broadcast transfer, the RT interface will set the "Broadcast Received" bit in the Status word response.
- **Transmit Last Command Word (10010)** - The Last Status word sent for each simulated RT is maintained by the onboard RT interface software. In addition, the Last Command word received by each simulated RT is maintained by the onboard RT interface software. If the simulated RT is setup to receive this Mode code using the *ApiCmdRTSACon* function the RT interface will automatically send the specific RTs Last Status word followed by a single Data word containing the Last Command word upon receipt of this Mode code. If the last transfer was a Broadcast transfer, the RT interface will set the "Broadcast Received" bit in the Status Word response and fetch the Last Command word from RT31 (the Broadcast RT).
- **Transmitter Shutdown (00100)** - If the simulated RT is setup to receive this Mode code using the *ApiCmdRTSACon* function, then upon receipt of this Mode code the RT interface will automatically disable the transmission on the alternate Bus for this specific RT. If this Mode code is broadcasted (with RT31 setup to receive this Mode code), the RT interface will disable the transmission on the alternate Bus for all simulated RTs.
- **Override Transmitter Shutdown (00101)** - If the simulated RT is setup to receive this Mode code using the *ApiCmdRTSACon* function, then upon receipt of this Mode code the RT interface will automatically enable the transmission on the alternate Bus for this specific RT. If this Mode

code is broadcasted (with RT31 setup to receive this Mode code), the RT interface will enable the transmission on the alternate Bus for all simulated RTs.

- **Reset Remote Terminal (01000)** - If the simulated RT is setup to receive this Mode code using the *ApiCmdRTSACon* function, then upon receipt of this Mode code the RT interface will automatically enable the transmission on both Busses for this specific RT. If this Mode code is broadcasted (with RT31 setup to receive this Mode code), the RT interface will enable the transmission on both Busses for all simulated RTs.

Note:
The special Mode code handling described above is not provided when the RT is operating in Mailbox Monitoring mode.

For all other Mode codes not automatically processed by the RT interface, yet are still required for processing by your application, the application developer will need to configure the simulated RT to receive the required Mode code and generate an interrupt upon transfer. This setup can be accomplished using the *ApiCmdRTSACon* function as described in Section 7.2. The user-developed interrupt service routine can be called to process the Mode code for the simulated RT. See Section 7.4 for further information regarding RT Interrupt Handling.

The following table provides a list of the Mode codes defined in MIL-STD-1553B. The codes shaded are codes which are simulated in the AIM 1553 module's RT interface if the RT operates in the MIL-STD-1553B protocol mode.

T/R Bit	Mode Code	Function	Data Word	Broadcast
1	00000	Dynamic Bus Control	N	N
1	00001	Synchronize	N	Y
1	00010	Transmit Status Word	N	N
1	00011	Initiate Self Test	N	Y
1	00100	Transmitter Shutdown	N	Y
1	00101	Override Transmitter Shutdown	N	Y
1	00110	Inhibit Terminal Flag Bit	N	Y
1	00111	Override Inhibit Terminal Flag Bit	N	Y
1	01000	Reset RT	N	Y
1	01001	Reserved	N	TBD
...	...	...	...	...
1	01111	Reserved	N	TBD
1	10000	Transmit Vector Word	Y	N
0	10001	Synchronize with Data	Y	Y
1	10010	Transmit Last Command	Y	N
1	10011	Transmit BIT Word	Y	N
0	10100	Selected Transmitter Shutdown	Y	Y
0	10101	Override Selected Transmitter Shutdown	Y	Y
TBD	10110	Reserved	Y	TBD
...	...	...	...	...
TBD	11111	Reserved	Y	TBD

7.2.5 RT Receive Broadcast Message Processing

To implement the capability for your simulated RT to receive Broadcast messages you will need to setup RT31 to receive application required Mode codes and/or Data Broadcast messages using the functions discussed in Section 7.2. The RT interface will provide automatic processing for RT31 of the Mode codes listed in Section 7.2.4. All other Mode codes not automatically handled by the RT interface will need to be processed by your application. The application developer will need to configure the RT31 to receive these required Mode codes and generate an interrupt upon transfer. This setup can be accomplished using the *ApiCmdRTSACon* function as described in Section 7.2. See Section 7.4 for further information regarding RT Interrupt Handling.

The user-developed interrupt service routine can be called to process the Broadcasted Mode code for the simulated RT(s) in order to performing the functions necessary to simulate the Mode code command issued.

7.3 RT Transfers Error Injection

There are two methods that you can use to inject errors into an RT transmission:

- Use the *ApiCmdRTSAConErr* function for error injection in any Status word or Data word transmitted by an RT
- Manipulate the Status word Busy or Message Error bits using the Status word masking ability in either the *ApiCmdRTIni* function, the *ApiCmdRTSACon* function or the *ApiCmdRTNXW* function.

These two methods are described in the following sections.

7.3.1 RT Transfers Error Injection into Status/Data Word

RT transmissions can be configured for error injection in any Status word or Data word transmitted by an RT. The RT is capable of injecting one of the following errors for a defined transfer:

- **Status Sync Error** - changes the transmitted Status word sync pattern to one specified by the user
- **Data Sync Error** - changes the transmitted Data word sync pattern to one specified by the user
- **Parity Error** - creates a parity error for the Status word or specified Data word
- **Manchester stuck at high error** - creates a Manchester stuck at high error for a specified Status word, or Data Word at a specified bit position
- **Manchester stuck at low error** - creates a Manchester stuck at low error for a specified Status word, or Data Word at a specified bit position

- **Gap error** - inserts specified Gap after defined Status or Data word
- **Word Count High** - transmits the number of Data words defined for the original transfer plus one
- **Word Count Low** - transmits the number of Data words defined for the original transfer minus one
- **Bit Count High** - transmits a specified number (1-3) additional bits for specified Status word or Data word.
- **Bit Count Low** - transmits a specified number (1-3) less bits for specified Status word or Data word.
- **Alternate Bus Error** - responds on the wrong bus
- **Zero Crossing Low Deviation Error** - implements zero crossing low deviation at a specified Status word or Data word position, bit position with four predefined deviation values.
- **Zero Crossing High Deviation Error** - implements zero crossing high deviation at a specified Command word or Data word position, bit position with four predefined deviation values.

To setup for error injection, the *ApiCmdRTSAConErr* function should be used after the RT has been setup to transmit data using the *ApiCmdRTBHDef* and *ApiCmdRTSACon* functions. The following error injection sample code will setup the transfer to inject a Data Sync Error on the third data word.

Set RT-to-BC RT01 Transmit SA02 WC15 (Inject Data Sync Error in 3rd data word)

```
wpos     = 3; // Inject Error on the 3 data word
bpos     = 0; // not used
bc_bits = 0; // not used
ty_api_rt_err.type       = API_ERR_TYPE_DATA_SYNC; // error injection type
ty_api_rt_err.sync       = 0x30;                   // set invalid data sync 110000
ty_api_rt_err.contig     = 0;                      // not used
ty_api_rt_err.err_spec   = 0;
ty_api_rt_err.err_spec  |= (wpos << 16);           // wpos
ty_api_rt_err.err_spec  |= (bpos << 8);            // bpos
ty_api_rt_err.err_spec  |= (bc_bits << 0);         // bc_bits
ApiCmdRTSAConErr( ulModHandle, 0,
                  1 /*RT*/, 2/*SA*/, rt_hid /*HID*/,
                  API_RT_TYPE_TRANSMIT_SA,
                  &ty_api_rt_err );
```

7.3.2 RT Transfer Error Emulation via Status Word Response

The Status word sent by an RT upon receipt of a BC Command/Data word can be manipulated to indicate that the RT is Busy or that it detected a Message Error. If either of these bits (See Figure 2.0.0-II, Status Word Bits) is set by the user, the RT SA/Mode code will only respond with the Status word, and not the requested data transmission/receipt. This can be accomplished using one of the following two methods:

- When initializing the RT using the *ApiCmdRTIni* function, the Next RT Status word parameter can be initialized with either of the bits set. However, this will cause the bit(s) to be set for Subaddresses and Mode code initialized for that RT. The following example initializes RT1 with the busy bit (4th bit) set for the Next Status word.

```
// Remote Terminal Initialization of RT1
// with the busy bit set in Next Status word
ApiCmdRTIni(ulModHandle,0, 1/*RT*/,API_RT_ENABLE_SIMULATION, 0, 8, 0x0808);
```

- When setting up the RT Subaddress/Mode code using the *ApiCmdRTSACon* function, the Status Word Mask control (smod) and Status Word Mask (swm) parameters can be setup to raise the Busy or Message Error bits of the Status word. The following example shows RT1 Transmit SA2 setup such that the it's Status Word response Message Error bit will be set (11th bit). The Status word Mask Control (smod) is set to use logical "OR" function as the masking mechanism.

```
// RT1 Transmit SA2 initialized
// with the Message Error bit set in Next Status word
ApiCmdRTSACon( ulModHandle, 0, 1/*RT*/, 2/*SA*/, rt_hid/*HID*/,
               API_RT_TYPE_TRANSMIT_SA,
               API_RT_ENABLE_SA, 0 /*rmod*/,
               API_RT_SWM_OR /*smod*/,
               0x0400 /*swm*/);
```

7.4 RT Interrupt Programming

As introduced in Section 7.2.1, the RT is capable of producing interrupts upon the occurrence of certain events. Interrupt Handler(s) must be created to process the interrupts which occur after the RT has been started and an interrupt occurs. Some possible RT Interrupt Handler applications may include: (1) refilling a transmit buffer at the end of a transfer interrupt, (2) gathering the data words received in the receive message buffer at the end of the transfer and/or (3) reporting transfer errors on a transfer error interrupt. The functions required to setup RT interrupts and interrupt handler execution include the Library Administration and Initialization functions as defined in Section 4.9, and the RT function call defined as follows:

- **ApiCmdRTSACon** - Setup the RT Subaddress to perform an interrupt on any of the following conditions:
  - Interrupt on End of Transfer
  - Interrupt on Transfer Error

Once you have configured the RT(s) to generate an interrupt and you have created an Interrupt Handler function to handle the interrupt, then start the RT using the *ApiCmdRTStart* function to start data flow. If the RT determines that an interrupt has occurred, the RT will initiate a call to the Interrupt Handler function and provide information about the interrupt as defined in the *ApiInstIntHandler* handler definition in the associated Reference Manual.

The following section describes how to setup an RT transfer to create an interrupt.

7.4.1 How to Setup the RT Transfer to Cause an Interrupt

To setup the interrupt the parameter Subaddress Control (con) of the *ApiCmdRTSACon* function is used. The Subaddress Control parameter can be set to one of the two following values:

- **API_RT_ENABLE_SA_INT_XFER** can be used as parameter Subaddress Control in order to enable the interrupt on End of Transfer
- **API_RT_ENABLE_SA_INT_ERR** can be used as parameter Subaddress Control in order to enable the interrupt on Transfer Error

8 Bus Monitor Programming

When data needs to be recorded from the bus one has to consider two options. The first option is to read the data message by message, the second option is to read big chunks of data efficiently in bulk. For the first option we recommend to implement an approach based on RT or BC interrupts where the data is read from the receive data buffer. For the second option we recommend an implementation based on Data Queues as described in Section 8.5.

The Data Queue Recording can be used in combination with Bus Monitor Triggers as described in Section 8.1 and Bus Monitor Filters as described in Section 8.2.

8.1 Additional Bus Monitor Trigger

In recording mode the BM triggers on any message by default. If you need a different Start Trigger Event you will need to setup at least one trigger. This section will describe the following:

- **Trigger Definition (using *ApiCmdBMTCBIni*)** - defines the types of triggers that are available and how to setup a Trigger Control Block (TCB)
- **Starting the "Data Capture" Process (using *ApiCmdBMFTWIni*)** - defines the software functions required to create a Start Trigger Event
- **Arming the Trigger (using *ApiCmdBMTIWIni*)** - defines the functions required to communicate to the Bus Monitor which triggers to evaluate.

Note:
Stopping the "Data Capture" is obsolete and not part of this description. The prefered solution is to record all data and search afterward or on the fly for interesting events

8.1.1 BM Trigger Definition

The Bus Monitor is capable of monitoring bus traffic using up to two dynamic triggers in parallel to determine the start of data capture. Triggers provide the user with the capability to monitor bus traffic for the occurrence of a specific error condition (such as parity error) and/or a discrete external trigger received at the BM Trigger input pin (See the corresponding Hardware Manual). Dynamic triggers provide the user with the capability to monitor the bus traffic for a sequence of events. An example of a sequence of events could be: SEQ1-a word received on the primary bus, SEQ2-the word is a status word, and SEQ3-the word has bit 8 set.

Each trigger requires that the user first configure a Trigger Control Block (TCB) which contains information about the conditions of the trigger. All triggers use the function ApiCmdBMTCBIni to configure their TCB. The user has the capability of pre-defining up to 254 TCBs, then using them as the BM application requires. This section will describe how to setup a TCB, however, to tell the BM the scheme to be used to start the "Data Capture" process and which TCBs to evaluate, you will need to issue two additional commands to the BM as defined in Section 8.1.1.3 and Section 8.1.1.4.

The following table contains the list of parameters associated with the TCB. The values for each parameter are dependent upon the Type of Trigger you are setting up. The following sections will discuss the parameter setup for different Triggers.

Trigger Control Block Parameter Definitions (*ApiCmdBMTCBIni / TY_API_BM_TCB*)		
Struct Element	**Description**	**Trigger type**
tt	Trigger Type - Trigger on: • 0 - Error Condition • 1 - External Event • 2 - Received Word • 3 - Data Value *	ALL
sot	Generate External Strobe on Trigger	ALL
tri	Generate Interrupt on Trigger	ALL
inv	Invert Result of Limit Check	Data Value
tres	Trigger Reset - the bits in the Monitor Status Trigger pattern to be reset when the trigger condition is met	ALL
tset	Trigger Set - the bits in the Monitor Status Trigger pattern to be set when the trigger condition is met	ALL
tsp	Trigger Specification	Data Value & Received Word
next	Next Trigger Control Block *	Data Value & Received Word
eom	Next Trigger Control Block on End of Message	Data Value & Received Word
tdw	Trigger Data word	Error Condition & Receive Word
tmw	Trigger Mask - defines bits of word relevant to Received word or Data value trigger	Data Value
tuli	Trigger Upper Limit - for range checks of Data Value Triggers	Data Value
tlli	Trigger Lower Limit - for range checks of Data Value Triggers	Data Value

Note:
* Check the Reference Manual for a detailed overview of functions supported for different devices.

As shown in the table above, there are 4 parameters that apply to all types of triggers. These parameters will be described in this section as follows:

- **Generate External Strobe on Trigger** - when set, if this TCB is active as one of the two possible Triggers, and the condition specified in the TCB is met, the BM will output a strobe signal on the external BM Trigger output pin.
- **Generate Interrupt on Trigger** - when set, if this TCB is active as one of the two possible Triggers, and the condition specified in the TCB is met, the BM will generate a TCB Interrupt and pass the TCB number to the BM Interrupt Handler Routine (user program)
- **Trigger Reset** - This parameter tells the BM what bits to reset in the Monitor Status Word if the trigger condition is met. See Section 8.1.1.3 for a more detailed description of the Trigger Start processing performed by the BM.

- **Trigger Set** - This parameter tells the BM what bits to set in the Monitor Status Word if the trigger condition is met. See Section 8.1.1.3 for a more detailed description of the Trigger Start processing performed by the BM.

8.1.1.1 Bus Monitor Static Trigger Definition

A Static Trigger is configured using the *ApiCmdBMTCBIni* function. A static trigger is a dynamic trigger sequence with only one trigger:

- Trigger on Error Condition
- Trigger on External Event (strobe or pulse) detected on the BM input Trigger pin.

The following two sections describe the parameters in the Trigger Control Block that are associated with these triggers.

Trigger on Error Condition: You can setup the BM to trigger on any one or more error conditions. If you specify more than one error condition for the Trigger Control Block, the trigger will be considered valid if any one of the error conditions is detected. The types of errors that can be setup to cause a Trigger Start Event in the Bus Monitor are shown in the table below.

Error Conditions for Triggering the Bus Monitor		
Parameter Bit ID	**Bit #**	**Error Description**
ERR	15	Any Error (Logical OR of Bits 14 to 0).
ALTER	14	Alternate Bus Response Error
LCNT	13	Low Word count Error
HCNT	12	High Word count Error
STAT	11	Status Word Exception Error*
TADDR	10	Terminal Address Error / RT RT Protocol Error
GAP	9	Early Response or Gap too short
ILLEGL	8	Illegal Command Word/ Reserved Mode Code Error
TX	7	Transmission on both MILbus channels
IWGAP	6	Interword Gap Error
ISYNC	5	Inverted Sync Error
PAR	4	Parity Error
LBIT	3	Low Bit Count Error
HBIT	2	High Bit Count Error
MANCH	1	Manchester Coding Error
NRESP	0	Terminal No Response Error

Note:
* The default Status word Exception mask is set to 0x07ff with the *ApiCmdBMIni* function (all status bits are checked). If you want the BM to trigger on only a subset of bits in the Status word, you must setup the Status Word Exception Mask using the BM function *ApiCmdBMSWXMIni* accordingly.

The subset of parameters in the Trigger Control Block Structure that define an Error Trigger include the parameters shown in the following table.

Trigger Control Block Parameter Definitions Applicable to Error Condition Trigger (*ApiCmdBMTCBIni / TY_API_BM_TCB*)		
Struct Element	**Description**	**Trigger type**
tt	Trigger Type - Trigger on: • 0 - Error Condition • 1 - External Event • 2 - Received Word • 3 - Data Value	0
sot	Generate External Strobe on Trigger	If desired
tri	Generate Interrupt on Trigger	If desired
tres	Trigger Reset - the bits in the Monitor Status Trigger pattern to be reset when the trigger condition is met	
tset	Trigger Set - the bits in the Monitor Status Trigger pattern to be set when the trigger condition is met	
tdw	Trigger Data word	

The following code sample sets up a Static Trigger Control Block with an Error Condition trigger to trigger on a Parity error and a Low Word Count error. When the trigger condition is met, the BM will not reset any bits in the Monitor Status Trigger pattern, but will set bits 0x0F in the Monitor Status Trigger pattern.

```
// Setup Static Trigger – Error Condition
// init TCB 3 for Trigger on Parity error and a Low Word Count error
api_bm_tcb.tt   = API_BM_TRG_ERR_CONDITION;
api_bm_tcb.sot  = API_DIS; // External Trigger
api_bm_tcb.tri  = API_DIS; // Interrupt on Trigger
api_bm_tcb.inv  = API_DIS; // Inversion of Limit Check
api_bm_tcb.tres = 0x00;    // Monitor Status Trigger pattern Reset Bits
api_bm_tcb.tset = 0x0F;    // Monitor Status Trigger pattern Set Bits
api_bm_tcb.tsp  = 0x00;    // Trigger spec
api_bm_tcb.next = 0xFF;    // next TCB (disabled for Static trigger)
api_bm_tcb.eom  = 0xFF;    // next TCB End of Message control (disabled)
api_bm_tcb.tdw  = 0x2010;  // Trigger Data Word – indicating check for parity and
// low word count
api_bm_tcb.tmw  = 0xFFFF;  // Trigger mask word
api_bm_tcb.tuli = 0x0000;  // Trigger upper limit
api_bm_tcb.tlli = 0x0000;  // Trigger lower limit
ApiCmdBMTCBIni(ulModHandle, 0, 3 /*tid*/, API_ENA, &api_bm_tcb);
```

Trigger on External Event Condition: The External Event Condition Static Trigger will trigger on an external strobe or pulse detected on the BM input Trigger pin. This type of Static Trigger is the least complex to setup. The subset of parameters in the Trigger Control Block Structure that define an External Event Trigger include the parameters shown the following table.

Trigger Control Block Parameter Definitions for External Event Trigger (*ApiCmdBMTCBIni / TY_API_BM_TCB*)		
Struct Element	**Description**	**Trigger type**
tt	Trigger Type - Trigger on: • 0 - Error Condition • 1 - External Event • 2 - Received Word • 3 - Data Value	1
sot	Generate External Strobe on Trigger	If desired
tri	Generate Interrupt on Trigger	If desired
tres	Trigger Reset - the bits in the Monitor Status Trigger pattern to be reset when the trigger condition is met	
tset	Trigger Set - the bits in the Monitor Status Trigger pattern to be set when the trigger condition is met	

The following code sample sets up a Static Trigger Control Block with an External Event trigger. When the trigger condition is met, the BM will not reset any bits in the Monitor Status Trigger pattern, but will set bits 0x0F in the Monitor Status Trigger pattern.

```
// Setup Static Trigger - External Event -
// External strobe or pulse detected on the BM input Trigger pin
// init TCB 4
api_bm_tcb.tt   = API_BM_TRG_EXTERNAL_EVENT; // Trigger Type
api_bm_tcb.sot  = API_DIS; // External Trigger
api_bm_tcb.tri  = API_DIS; // Interrupt on Trigger
api_bm_tcb.inv  = API_DIS; // Inversion of Limit Check
api_bm_tcb.tres = 0x00;    // Monitor Status Trigger pattern Reset Bits
api_bm_tcb.tset = 0x0F;    // Monitor Status Trigger pattern Set Bits
api_bm_tcb.tsp  = 0x00;    // Trigger spec
api_bm_tcb.next = 0xFF;    // next TCB (disabled for Static trigger)
api_bm_tcb.eom  = 0xFF;    // next TCB End of Message control (disabled)
api_bm_tcb.tdw  = 0xFFFF;  // Trigger data word
api_bm_tcb.tmw  = 0xFFFF;  // Trigger mask word
api_bm_tcb.tuli = 0x0000;  // Trigger upper limit
api_bm_tcb.tlli = 0x0000;  // Trigger lower limit
ApiCmdBMTCBIni(ulModHandle, 0, 4 /*tid*/, API_ENA, &api_bm_tcb);
```

8.1.1.2 Bus Monitor Dynamic Trigger Definition

A Dynamic Trigger is a sequence of triggers containing more than one trigger. A Dynamic Trigger is also configured using the *ApiCmdBMTCBIni* function. Two Dynamic triggers can be active at one time. In addition to the already defined error trigger and external event trigger this chapter will introduce:

- Trigger on Received word

- Trigger on Data Value.

If your Dynamic Trigger involves more than one trigger condition, then multiple TCBs will need to be defined for the dynamic trigger and linked together using the Next TCB parameter in the TCB. The first trigger in the sequence will reference the TCB of the next trigger to evaluate when the first trigger condition is met, and so on. Following is a High Level Language example of a Dynamic Sequence Trigger using pre-defined TCBs (1 - 4):

Dynamic Sequence Trigger (T0):

```
IF TCB1 [Received word = Command Word = Data value]
THEN
        Set Bit 0 of the Monitor Status Trigger pattern
        Evaluate TCB2 (Next TCB)
ELSE    ReARM TCB1 (EOM TCB)

IF TCB2 [Data Word 3 is in range (100-1000)]
THEN
        Set Bit 1 of the Monitor Status Trigger pattern
        Evaluate TCB3 (Next TCB)
ELSE    Return to TCB1 Evaluation (EOM TCB)

IF TCB3 [Received word = Command Word = Data Value]
THEN
        Set Bit 2 of the Monitor Status Trigger pattern
        Evaluate TCB4 (Next TCB)
ELSE    Return to TCB1 Evaluation (EOM TCB)

IF TB4 [Data Word 6 Bit 5 AND Bit 8 Set]
THEN
        Set Bit 3 of the Monitor Status Trigger pattern
        Evalute TCBxx (Next Index, ARM TBxx)
ELSE    Return to TCB3 Evaluation (EOM Index)
```

The following two sections describe the parameters in the Trigger Control Block that are associated with these triggers.

Trigger on Received Word: The Trigger on Received word enables the user to setup the Bus Monitor to search for a Command word (1 or 2), Status word or Data word on the Primary or Secondary Bus with a specified value. The bits defined for setup in the Trigger Specification (tsp) include the Received words as defined in the following table:

Received Words Triggering the Bus Monitor		
Parameter Bit ID	**Bit #**	**Receive Word Description**
RXA	5	Word received on Primary Bus

Continued on next page

Continued from previous page

Parameter Bit ID	Bit #	Receive Word Description
RXB	4	Word received on Secondary Bus
CW2	3	Word is second Command Word for RT RT transfer
ST	2	Word is Status Word
DW	1	Word is Data Word
CW	0	Word is Command Word

The subset of parameters in the Trigger Control Block Structure that define a Trigger on Received word include the parameters shown in following table.

Trigger Control Block Parameter Definitions (*ApiCmdBMTCBIni / TY_API_BM_TCB*)		
Struct Element	**Description**	**Trigger type**
tt	Trigger Type - Trigger on: • 0 - Error Condition • 1 - External Event • 2 - Received Word • 3 - Data Value	3
sot	Generate External Strobe on Trigger	If desired
tri	Generate Interrupt on Trigger	If desired
tres	Trigger Reset - the bits in the Monitor Status Trigger pattern to be reset when the trigger condition is met	
tset	Trigger Set - the bits in the Monitor Status Trigger pattern to be set when the trigger condition is met	
tsp	Trigger Specification	
next	Next Trigger Control Block	
eom	Next Trigger Control Block on End of Message	
tdw	Trigger Data word	
tmw	Trigger Mask - defines bits of word relevant to Received word or Data value trigger	

The trigger condition is valid, if the following expression becomes valid:

(The word is received on the Primary or Secondary Bus) **AND**
(The word is Command word 2 **OR** Status word **OR** Data word **OR** Command word) **AND**
(Compare Value Check == True)

The following code sample sets up a Dynamic Trigger Control Block (TCB 5) with an Received Word Condition trigger to trigger on reception of a Command word received on the Primary or Secondary Bus for RT1 Transmit SA1 with a 32 data word count. When the trigger condition is met, the BM will not reset any bits in the Monitor Status Trigger pattern, but will set bits 0x01 in the Monitor Status Trigger pattern. The BM will then begin to evaluate TCB 6 for the next word received by the BM which is indicated when you set the Next TCB (next) to 0xFF. If the Trigger condition is not fully met by the end of the transfer, the BM will continue to monitor TCB 5 which is indicated when you set the EOM TCB (eom) to 0xFF.

```
// Setup Dynamic Trigger – Received Word – Command word received on primary or
```

```
// secondary bus for RT1 SA1 with a 32-word data count
// init TCB 5 for Trigger on Command from RT1 Transmit SA1 with Data word count = 32
api_bm_tcb.tt   = API_BM_TRG_RECEIVED_WORD; // Trigger Type
api_bm_tcb.sot  = API_DIS; // External Trigger
api_bm_tcb.tri  = API_DIS; // Interrupt on Trigger
api_bm_tcb.inv  = API_DIS; // Inversion of Limit Check
api_bm_tcb.tres = 0x00;    // Monitor Status Trigger pattern Reset Bits
api_bm_tcb.tset = 0x01;    // Monitor Status Trigger pattern Set Bits
api_bm_tcb.tsp  = 0x31;    // Trigger spec bits set for Pri or Sec Bus & CW
api_bm_tcb.next = 0x06;    // next TCB
api_bm_tcb.eom  = 0xFF;    // next TCB End of Message control
api_bm_tcb.tdw  = 0x0C20;  // Cmd = RT1TSA1 with word count = 32
api_bm_tcb.tmw  = 0xFFFF;  // Trigger mask word - all bits applicable
api_bm_tcb.tuli = 0x0000;  // Trigger upper limit
api_bm_tcb.tlli = 0x0000;  // Trigger lower limit
ApiCmdBMTCBIni(ulModHandle, 0, 5 /*tid*/, API_ENA, &api_bm_tcb);
```

Trigger on Data Value Condition: The Trigger on Data Value Trigger enables the user to setup the Bus Monitor to evaluate a specific Data value in a word position value 1 to 32 which corresponds directly with Data Word Location 1 to 32 of a MILbus transfer. This type of Dynamic Trigger is best used in conjunction with a Received Word Trigger defined in the previous section in order to provide a filtering of the transfer message prior to data word evaluation. The subset of parameters in the Trigger Control Block Structure that define a Dynamic Data Value Trigger include the parameters shown in the following table.

Trigger Control Block Parameter Definitions Applicable to Data Value Trigger (*ApiCmdBMTCBIni / TY_API_BM_TCB*)

Struct Element	Description	Trigger type
tt	Trigger Type - Trigger on: • 0 - Error Condition • 1 - External Event • 2 - Received Word • 3 - Data Value	3
sot	Generate External Strobe on Trigger	If desired
tri	Generate Interrupt on Trigger	If desired
inv	Invert Result of Limit Check	If desired
tres	Trigger Reset - the bits in the Monitor Status Trigger pattern to be reset when the trigger condition is met	
tset	Trigger Set - the bits in the Monitor Status Trigger pattern to be set when the trigger condition is met	
tsp	Trigger Specification - Data Word Position	
next	Next Trigger Control Block	
eom	Next Trigger Control Block on End of Message	
tdw	Trigger Data word	
tmw	Trigger Mask - defines bits of word relevant to Received word or Data value trigger	
tuli	Trigger Upper Limit - for range checks of Data Value Triggers	

Continued on next page

Continued from previous page

Struct Element	Description	Trigger type
tlli	Trigger Lower Limit - for range checks of Data Value Triggers	

The received MILbus Word is masked (bitwise logical AND) with the Trigger Mask word (tmw). The result is compared with the Upper limit (tuli) and Lower limit (tlli). Proper setting of these values allow masking for certain bit fields as well as for dedicated values.

The trigger condition is valid, if the following expression becomes valid:

(Received Bus Word **AND** Trigger Mask word) >= Lower limit **AND**
(Received Bus Word **AND** Trigger Mask word) <= Upper limit

The following code sample sets up a Dynamic Trigger Control Block (TCB 6) with a Data Value Condition trigger to trigger on reception of a 4th Data word equal to 0x0033. It is designed to be used in sequence with the TCB 5 which was setup in the previous section If the BM determines that the 4th word does equal 0x0033, the BM will not reset any bits in the Monitor Status Trigger pattern, but will set bits 0x0E in the Monitor Status Trigger pattern. It will then be re-armed with TCB5 which is indicated when you set the Next TCB (next) to 0x05. If the 4th word does not equal 0x0033, no action is taken with the Monitor Stats Trigger word bits and the BM will be re-armed using TCB 5 which is indicated when you set the EOM TCB (eom) to 0x05.

```
// Setup Dynamic Trigger - Data Value - 4th Data word equal to 0x0033.
// init TCB 6 for Trigger on Data Value 4th Data word equal to 0x0033.
api_bm_tcb.tt   = API_BM_TRG_DATA_VALUE; // Trigger Type
api_bm_tcb.sot  = API_DIS; // External Trigger
api_bm_tcb.tri  = API_DIS; // Interrupt on Trigger
api_bm_tcb.inv  = API_DIS; // Inversion of Limit Check
api_bm_tcb.tres = 0x00;    // Monitor Status Trigger pattern Reset Bits
api_bm_tcb.tset = 0x0E;    // Monitor Status Trigger pattern Set Bits
api_bm_tcb.tsp  = 0x04;    // Check the 4th Data word for value in .tdw
api_bm_tcb.next = 0x05;    // next TCB
api_bm_tcb.eom  = 0x05;    // next TCB End of Message control
api_bm_tcb.tdw  = 0x0000;  // reserved for .tt = API_BM_TRG_DATA_VALUE
api_bm_tcb.tmw  = 0xFFFF;  // Trigger mask word - all bits applicable
api_bm_tcb.tuli = 0x0033;  // Trigger upper limit (Compare value = 0x0033)
api_bm_tcb.tlli = 0x0033;  // Trigger lower limit (Compare value = 0x0033)
ApiCmdBMTCBIni(ulModHandle, 0, 6 /*tid*/, API_ENA, &api_bm_tcb);
```

8.1.1.3 Starting the "Data Capture" Process

Each user-defined Trigger Control Block contains two parameters associated with starting the "Data Capture" process. These two parameters include:

- Trigger Set Bits - define the Trigger bits in the Monitor Status Word to set

- Trigger Reset Bits - define the Trigger bits in the Monitor Status Word to reset

Each time a TCB condition is met, the Bus Monitor will set/reset the Trigger bits (8 bits) of the Monitor Status Word (internal to the Bus Monitor) as the user specifies in the TCB parameters: Trigger Set Bits (tset) and Trigger Reset Bits (tres). (The 8 Trigger bits of the Monitor Status Word are referred to as the Monitor Status Trigger pattern.) This provides the user with the capability to "build" a "Data Capture" Start Trigger Event. Thus, providing the user with the capability to create an infinite number of scenarios to start the "Data Capture" process.

In order for this capability to work, the user must specify the final bit value that the Bus Monitor will associate as the Start Trigger Event. This bit value is defined within the Function Trigger Word using the *ApiCmdBMFTWIni* function. As shown in the table below, the Function Trigger Word defines the bit patterns used to create a Start Trigger Event. The BIU Processor uses the Function Trigger Word and the Monitor Status Trigger pattern to determine the start of the "Data Capture".

Bus Monitor Function Trigger Word			
31 24	**23 ..... 16**	**15 ..... 8**	**7 ..... 0**
Reserved	Reserved.	Start Trigger Mask	Start Trigger Compare

The **Start Trigger Event** condition is met if the state of the **Monitor Status Trigger** pattern masked (logical "AND") with the **Start Trigger Mask** is equal to the **Start Trigger Compare** pattern.

This feature is most useful for Dynamic triggers containing more than one sequence of TCBs or a combination of a single trigger and Dynamic Triggers. For instance, you may not want the "Data Capture" to start when the first TCB condition is met. You may want the "Data Capture" Start Trigger Event to occur after two or more TCB conditions have been met. In addition, you may want the occurrence of a trigger such as an external pulse or error condition.

In the previous section, we have example code for an External Event trigger (TCB 4), and an Error Condition trigger (TCB3). There are also two TCBs (TCB 5 and 6) which are linked together to form one Dynamic Sequence Trigger. The first two TCBs are defined with parameters "tset" = 0x0F. The Dynamic TCBs set the "tset" sequentially (TCB5 sets "tset" = 0x01, TCB6 sets "tset" = "0x0E") such that when both conditions are met the Monitor Status Trigger pattern equals 0x0F. In each trigger condition, the final trigger pattern to create a Start Event Trigger is 0x0F, therefore, the following code shows how to configure the Function Trigger Word in the BM to start "Data Capture" when the Monitor Status Trigger pattern equals 0x0F:

```
// Set the Function Trigger Word with Mask and compare Values
// the Stop Trigger Mask and Compare is not used in this example.
ApiCmdBMFTWIni( ulModHandle, 0,
                API_BM_WRITE_ALL,
                0x00 /*reserved*/,
                0xFF /*reserved*/,
                0x0F /*Start Trigger Mask*/,
                0x0F /*Start Trigger Compare*/);
```

8.1.1.4 Arming the Trigger

As stated earlier in this section, the Bus Monitor is capable of monitoring bus traffic using up to two trigger sequences in parallel. To enable the Bus Monitor to evaluate the Trigger(s) you have defined in your Trigger Control Blocks, you must setup the Trigger Index Word using the *ApiCmdBMTIWIni* function.

Issuing the *ApiCmdBMTIWIni* function command tells the BM which TCB(s) to evaluate for each the two triggers allowed to be evaluated simultaneously.

Bus Monitor Trigger Index Word			
31 24	**23 ..... 16**	**15 ..... 8**	**7 ..... 0**
Reserved	Reserved	Sequence Trigger 1 TCB Index	Sequence Trigger 0 TCB Index

In the previous section, we have example code for an single External Event trigger (TCB 4), an single Error Condition Trigger (TCB3). In addition, two TCBs (TCB 5 and 6) are linked together to form one Dynamic Sequence Trigger. If we want to enable the BM to evaluate two of these TCBs simultaneously we should issue the *ApiCmdBMTIWIni* function as follows:

```
// Set the Trigger Index Word with the indexes of the TCBs
ApiCmdBMTIWIni(ulModHandle, 0,
                API_BM_WRITE_ALL,
                0,  // reserved,
                0,  // reserved,
                3,  // sequence 0 TCB single error trigger
                5); // sequence 1 TCB dynamic trigger sequence
```

The BSP sample ls_bc_rt_bm_tcb_sample.c shows how to setup the Trigger Control Block for Trigger on Command and Trigger on data value.

8.2 Additional Bus Monitor Filter

There is one additional function that will aid the developer in filtering out bus traffic that is not required for evaluation or recording. This function is applicable to all Capture modes and is defined as follows:

- *ApiCmdBMFilterIni* - Enables/disables the recording of 1553 transfers based on user-specified RT Transmit and Receive Subaddresses and Mode codes. (When the BM is initialized using the *ApiCmdBMIni* function, all RT Transmit and Receive Subaddresses and Mode codes are enabled).

8.3 Recording Using Queuing

The BM can be setup to store the data to the Monitor Buffer such that 1553 transfers can be retrieved in a queuing fashion. This mode is the most simplistic mode to use. You would use Queuing if you want

to record the user-specified bus traffic and do not require the use of a trigger for data analysis and/or if you prefer the structured outputs available in this recording mode. The Recording with Queuing process requires the use of the following functions:

- ***ApiCmdBMIni*** - Initialize the BM.
- ***ApiCmdQueueIni*** - this function is used to initialize the BM Queuing process. This function takes the place or the *ApiCmdBMCapMode*.
- ***ApiCmdQueueStart*** - this function will start queuing the BM data to the Monitor Buffer. (This function takes the place of *ApiCmdBMStart*.)
- ***ApiCmdQueueRead*** - this function will return a Message entry on each call. It is a First-in, First-out manner. If no message is on the stack then the return value will indicate this. The Monitor Buffer pointer starts at the first entry of the queue and is automatically updated to the next transfer entry upon completion of the *ApiCmdQueueRead* function. At that time an additional *ApiCmdQueueRead* function can be issued to read the next entry.

The BSP sample ls_bc_rt_bm_sample.c shows this mode for the Bus Monitor.

8.4 Recording Using Data Queue Recording

The Data Queue Recording has been designed to copy big chunks of data in an efficient way. All Data Queue functions start with a ApiCmdDataQueue prefix. These functions are used in combination of the standard Bus Monitor functions which start with a ApiCmdBM prefix.

- **ApiCmdBMIni** - Initialize the BM
- **ApiCmdBMCapMode** - Set the BM capture mode to API_BM_CAPMODE_RECORDING
- **ApiCmdBMStart** - Start the BM
- **ApiCmdDataQueueOpen** - Open the Data Queue (DQ)
- **ApiCmdDataQueueControl** - Start the DQ with API_DATA_QUEUE_CTRL_MODE_START
- **ApiCmdDataQueueRead** - Read from DQ with size zero to check if data is available
- **ApiCmdDataQueueRead** - Read from DQ with size greater zero to copy the available data
- **ApiCmdBMHalt** - Stop the BM
- **ApiCmdDataQueueControl** - Stop the DQ with API_DATA_QUEUE_CTRL_MODE_STOP
- **ApiCmdDataQueueRead** - Read from DQ with size zero to check if data is available
- **ApiCmdDataQueueRead** - Read from DQ with size greater zero to copy the available data
- **ApiCmdDataQueueClose** - Close the DQ

8.5 1553 Data Queue Recording

A typical Data Queue Recording sequence can be found in the ls_recording_replay_sample.c and ls_bm_sample.c delivered with the BSP. These samples also show how to parse the data returned by the bus monitor.

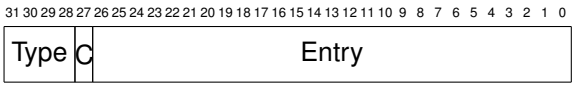
The function *ApiCmdDataQueueRead* returns the data in a binary format as described in section 8.6. If the data is post processed it is strongly recommended to always read all available data. This guaranties that the returned data does not contain any partial messages.

E.g. If the Bus Monitor receives a full size message which consists out of 144 bytes in the bus monitor. *ApiCmdDataQueueRead* will either return return 0 available bytes or 144 bytes. In the first case the message has not been complete. In the second case the message was completely received. This makes parsing and post processing the data much easier. If the application however reads a fixed size of byes it can not be guaranteed that the data returned contains complete messages.

8.6 Format of Data Stored in the 1553 Monitor Buffer

The Monitor stores all received words from the bus together with Time Tag information and possible error entries, as 32 bit Monitor Words in little endian format. The Monitor Words are described in section 8.6.1.

8.6.1 Monitor Words

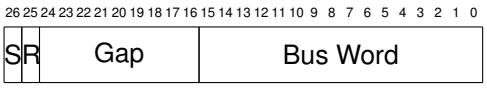


- **Type** [Bit 31 - 28] Describes what is stored in the Entry field
  - **0x0** Entry not updated
  - **0x1** Error Word Entry
  - **0x2** Time Tag Low Entry
  - **0x3** Time Tag High Entry
  - **0x8** Bus Word Entry - Command Word on Primary Bus
  - **0x9** Bus Word Entry - Command Word 2 on Primary Bus
  - **0xA** Bus Word Entry - Data Word on Primary Bus
  - **0xB** Bus Word Entry - Status Word on Primary Bus
  - **0xC** Bus Word Entry - Command Word on Secondary Bus
  - **0xD** Bus Word Entry - Command Word 2 on Secondary Bus
  - **0xE** Bus Word Entry - Data Word on Secondary Bus
  - **0xF** Bus Word Entry - Status Word on Secondary Bus
- **C** [Bit 27] Entry Connection flag. Set to one if an additional entry follows which is logically connected to this entry. This bit is always set, if during a single word receive operation more than one entry is written to the buffer. For example, this can happen if the first command word of a transfer

is received and the Time tag information is additionally stored in the buffer. Also, it can happen if an erroneous word is received and an the error entry is written to the buffer.

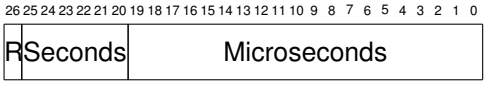
- **Entry** [Bit 26 - 0] The interpretation of this Entry depends on the Type field. See Sections 8.6.2, 8.6.3, 8.6.4 and 8.6.5 for details.

8.6.2 1553 Bus Word Entry



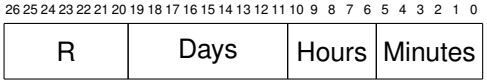
- **S** [Bit 26] Start Trigger Flag - If Start Trigger flag is set, the Bus Word Entry is related to the start trigger event which starts capturing of MILbus traffic.
- **R** [Bit 25] Reserved.
- **Gap** [Bit 24 - 16] Gap Time Value - The Gap Time value reports the time between the current and the previous received MILbus Words, in 0.25us steps. If the words are received continuously, the reported gap time will be 2us, according to the "MIL-STD1553 Gap time Measurement Definition". The range of the gap time values is 2us to 109.75us. If the reported gap time value is zero, the gap time was greater than 109.75us.
- **Bus Word** [Bit 15 - 0] Received MILbus Word - The received 16 bit MILbus Word which was received by the current operation. The word type (command, status or data) is defined by the Type field.

8.6.3 1553 Time Tag Low Entry



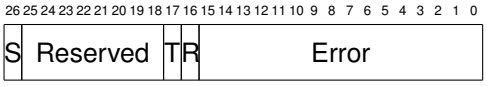
- **R** [Bit 26] Reserved.
- **Seconds** [Bit 25 - 20] Seconds of minute 0 to 59.
- **Microseconds** [Bit 19 - 0] Microsecond of second 0 to 999999.

8.6.4 1553 Time Tag High Entry



- **R** [Bit 26 - 20] Reserved.
- **Days** [Bit 19 - 11] Day of year 1 to 365.
- **Hours** [Bit 10 - 6] Hour of the day 0 to 23.
- **Minutes** [Bit 5 - 0] Minutes of hour 0 to 59.

8.6.5 1553 Error Word Entry



- **S** [Bit 26] Start Trigger Flag - If Start Trigger flag is set, the Error Word Entry is related to the start trigger event, which starts capturing of MILbus traffic.
- **Reserved** [Bit 25 - 18] Reserved.
- **T** [Bit 17] Transmit RT - This bit position is set if the Error Word Entry is related to a transmit command.
- **R** [Bit 16] Receive RT - This bit position is set if the Error Word Entry is related to a receive command.
- **Error** [Bit 15 - 0] The Error Type field indicates error type which was detected during the receive operation of the related, received word. [1]
  - **Bit 15** ANY-ERROR (logical OR of Bits 14 to 0)
  - **Bit 14** Alternate Bus Response Error
  - **Bit 13** Low Word count Error
  - **Bit 12** High Word count Error
  - **Bit 11** Status Word Exception [2]
  - **Bit 10** Terminal Address Error / RT-RT Protocol Error
  - **Bit 9** Early Response or Gap Time too short Error
  - **Bit 8** Illegal Command Word / Reserved Mode Code Error
  - **Bit 7** Transmission on both MILbus channels
  - **Bit 6** Interword Gap Error
  - **Bit 5** Inverted Sync Error
  - **Bit 4** Parity Error
  - **Bit 3** Low Bit Count Error
  - **Bit 2** High Bit Count Error
  - **Bit 1** Manchester Coding Error
  - **Bit 0** Terminal No Response Error

[1] If an error is detected during a RT- RT- transfer, it will be related to the currently active RT. If the monitor detects a protocol error during a RT-RT- transfer, it will be related to both RT's.

[2] The default Status word Exception mask is set to 0x07ff with the ApiCmdBMIni function (all status bits are checked). If you want the BM to detect an error on only a subset of bits in the Status word, you must setup the Status Word Exception Mask using the BM function ApiCmdBMSWXMIni accordingly.

9 Replay Programming

If Replay is enabled, the BIU Processor physically replays a file in the format as recorded during the Monitor Operation. Before Replay can be started, the first entries of the file must be copied to the Replay buffer area with ApiWriteRepData. With each call of this function half of a Replay buffer can be written. Call this function twice in order to fill the complete buffer. The replay buffer size in the Global RAM is 20000 Hex for each stream. Once started, the BIU Processor reads and transmits the data from the Replay Buffer. If programmed, an interrupt can be asserted each time half of the Replay buffer is transmitted. This provides for double buffer type refill strategy. When the Replay Buffer end is reached, the processor will wrap around to the replay buffer start address. This operation continues until the number of entries, as specified in the Replay Entry Count location are processed or a monitor entry indicating "Entry not Updated" is found.

Note:
ApiWriteRepData does automatically write to the half buffer which is currently not replayed.

The replay mode can be configured to disable replay for a selected RT. If the selected RT is disabled for Replay, an external RT should be connected to the bus to respond to the BC commands to that RT. Special handling is provided to cope with differences in the response time between the recorded and the actual external RT.

The protocol type (MIL-STD-1553A or MIL-STD-1553B) must match the protocol type of the recorded RT's.

Since the Replay function is the bus master and issues the command words on the MIL-STD-1553 bus it can not be combined with BC operation. Any attempt to enable the Replay mode together with the BC mode will be rejected. However, the Replay mode can be combined with RT and Bus Monitor Mode. RTs can be setup to mailbox the data from the replay file or from external RT's, or actively respond to commands issued from the replay module.

Note:
The Replay mode does not reproduce any recorded error conditions nor does it provide special error handling for external RT's during the replay process. The only exception handling is a timeout feature which inhibits a lock-up if external RT's are used which do not respond.

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

- Initialization (*ApiCmdReplayIni*) - provides initialization of the following:
  - Enable/disable half buffer transmitted interrupt
  - Time Tag replay setup:
    - Replay of the IRIG time tag
    - Replay only the Low IRIG Time Tag (seconds and microseconds)
    - Disable time tag replay
- Definition of the amount of data (in bytes) to be replayed
- Copy the recorded Monitor data to the Replay Buffer area in Global RAM using the *ApiWriteRepData* System function. This function allows you to write a half buffer size (0x10000) or less to the currently inactive half Replay Buffer.

- An optional choice now would be to disable any RTs for which you do not wish the data to be replayed. If you disable an RT, you will need to connect an External RT to respond to the BC commands to the RT which are present in the data to be replayed. The *ApiCmdReplayRT* function provides the disable RT capability.
- Start the Replay using the *ApiCmdReplayStart* function.
- Refill the Replay Buffer:
  - If a Half-Buffer transmitted interrupt has been enabled, and an interrupt handler has been developed to handle the interrupt, then the Replay Buffer can be refilled.
  - *As an alternative to the interrupt the function* ApiCmdReplayStatusRead *can be used to poll the half buffer full interrupt counter*.

The *ls_recording_replay_sample.c* demonstrates how to fill both halves of the Replay Buffer, then start the Replay function. The Entry count of the Replay Buffer is continuously monitored, and when the count = 0, the Replay is stopped. Each time the replay interrupt count is incremented (a half Replay buffer was replayed to the bus) another half Replay buffer is loaded.

Note:
The Replay function is not available on all modules. Check Appendix B of the Reference Manual for a detailed overview of functions supported for different devices.

10 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.

10.1 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 *ApiGetErrorMessage* to provide a human readable error description.

10.2 Using Counters

Reading and evaluating transmission and error counters of BC, RT and BM can help understand problems on the bus. Use the functions *ApiCmdBCStatusRead*, *ApiCmdRTStatusRead* and *ApiCmdBMStatusRead* to obtain the status information.

10.3 Monitoring the Bus Traffic

Enable the monitoring of the stream or attach a separate board to the bus. This may be a different board, for example with the PBA.pro running on it. Understanding the traffic on the bus can help to understand the nature of the problem.

10.4 Contacting Support

Collect data about the board. What is the correct name of the board, which are the versions on board? The output of the print board versions sample delivered with the BSP should always be provided in support requests to speed up the problem determination. To get this output start the sample executable delivered with the BSP with the sample number one. E.g.

```
1553_sample_project.exe local 0 1 1
```

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

11 Migrating from API 22 to API 24

This section is designed to help in migrating applications running with BSP 11.x (Library 22) to BSP 12.x.y (Library 24). The gap between 22 and 24 is due to an internal version 23 which was only released in a special BSP. The interface of library 22 and 23 is identical.

11.1 Library Interface

- **Recompile** The application needs to be recompiled in order to match the calling convention and struct alignment of the new library.
- **Rename** The Windows DLL and LIB have been renamed to contain the major version in the name. Through this we want to ensure that your application is only started with a compatible library. The linker settings in the application project needs to be adapted.

11.2 Incompatible Function Prototypes

- **Return Value** The return value of all functions indicating success or failure by an integral value is now AiReturn which is a 32-bit integer value. Probably, your compiler will complain if you're assigning this value to 16-bit variables as was common in older API versions. This can temporarily be ignored, as the values returned by all functions are below 0xFFFF. It is recommended to take care of this compiler warnings, as values higher than 0xFFFF are possible in future. Please note that some of the individual error values have also changed.
- **ApiCmdIni** The default mode parameter 0 which initialized the whole board was changed to be equal to the read only mode. This function should no longer be called and is now obsolete. If you need any of the information provided by the information struct, please consider using ApiCmdSysGetBoardInfo instead.

  In the very rare case that you need to switch back to the old behavior you can use API_INIT_MODE_ALL as a fall back.
- **ApiCmdReset** The mode API_RESET_ONLY_IF_NOT_ALREADY_RESETTED was removed. To detect if the board is already opened and initialized by another application please use ApiGetDriverInfo/ul_OpenConnections. This function returns the number of independent tasks that have an open handle to the device. If the value is greater than one some other task is already working on the device and some functions should be used with care. See the sample main function for a example on how to utilize this feature.

  Please note that this function does not count instances of threads that share a common library instance.
- **ApiInstIntHandler** The function signature for the callback registered with ApiInstIntHandler changed. When using this function in an application, the registered callback implementation must be adapted manually. We change the calling convention of the callback from stdcall to cdecl. The struct parameter with the interrupt information was change from pass by value to pass by pointer.

  See ls_interrupt_sample.c for a example implementation.

- **ApiProvideScopeBuffers, ApiCreateScopeBufferList** The callback function signature for the scope buffer is now called with cdecl calling convention. When using this function in an application, the registered callback implementation must be adapted manually.
- **ApiCmdGetIrig/ApiCmdSetIrig** The structure representing the IRIG time has been changed. The external/internal status and control has been separated into the additional functions ApiCmdGetIrigStatus/ApiCmdSetIrigStatus.
- **ApiCmdDataQueueOpen** The prototype of this function has been simplified. The ids for the different queues have been unified. The appropriate buffering mode for client or server is now detected internally. The function returns the size of the buffer which is accessed with ApiCmdDataQueueRead.
- **ApiCmdDataQueueRead** The prototype of this function has been simplified.
- **ApiCmdDataQueueControl,ApiCmdDataQueueFlush,ApiCmdDataQueueClose** The prototype of this functions has been simplified. All struct parameters are now direct input values of the function.

11.3 Obsolete Function Prototypes

The following functions have been declared as obsolete and might be removed by a future incompatible BSP.

- **ApiOpen** Please consider using ApiOpenEx instead.
- **ApiCmdIni** Please consider using ApiCmdSysGetBoardInfo instead.
- **ApiReadBSPVersion** The version scheme of several software components has been changed and is not fully supported by this function. New devices have versions that are not returned by this function. Please use ApiReadVersion or ApiReadAllVersions instead.
- **ApiReadBSPVersionEx** The version scheme of several software components has been changed and is not fully supported by this function. New devices have versions that are not returned by this function. Please use ApiReadVersion or ApiReadAllVersions instead.
- **ApiReadRecData** Please use the Data Queue recording functions instead.
- **ApiCmdBMStackEntryFind,ApiCmdBMStackEntryRead,ApiCmdBMStackpRead** Please use the Data Queue recording functions instead.
- **ApiCmdBMIniMsgFltRec,ApiCmdBMReadMsgFltRec** Please use the Data Queue recording functions instead. If message filtering on the device is required, the Data Queue recording can be combined with FW level filtering. See ApiCmdBMFilterIni in the reference manual.

11.4 Removed Memory Functions With Replacements

This functions have been used to access the board memory. Please use the functions ApiReadMemData and ApiWriteMemData with API_MEMTYPE_GLOBAL instead.

- ApiCmdRamRead
- ApiCmdRamReadByte
- ApiCmdRamReadLWord
- ApiCmdRamReadWord
- ApiCmdRamWrite
- ApiCmdRamWriteByte
- ApiCmdRamWriteLWord
- ApiCmdRamWriteWord

11.5 Removed Functions Without Replacements

This functions have been removed and there is no functional replacement.

- ApiGetTcomStatus
- ApiSetTgEmul/ApiGetTgEmul
- ApiPrintfOnServer
- ApiGetOpenErr
- ApiCmdCalTransCon
- ApiCmdTimerIntrCheck
- ApiCmdBiuIntrCheck

List of Abbreviations

DQ Data Queue

LIST OF FIGURES