United Electronic Industries

The High-Performance Alternative

DNA/DNR-1553-553 MIL-STD-1553 Communications Interface — User Manual

Dual-channel, MIL-STD-1553 Communications Interface
for the PowerDNA Cube and RACKtangle Chassis

May 2010 Edition

PN Man-DNA-1553-0510

Version 1.6

See the UEI website for complete terms and conditions of sale:
http://www.ueidaq.com/company/terms.aspx

Contacting United Electronic Industries

Mailing Address:

27 Renmar Ave.
Walpole, MA 02081
U.S.A.

For a list of our distributors and partners in the US and around the world, please see http://www.ueidaq.com/partners/

Support:

Telephone:(508) 921-4600
Fax:(508) 668-2350

Also see the FAQs and online "Live Help" feature on our web site.

Internet Support:

Support:support@ueidaq.com
Web-Site:www.ueidaq.com
FTP Site:ftp://ftp.ueidaq.com

Product Disclaimer:

WARNING!

DO NOT USE PRODUCTS SOLD BY UNITED ELECTRONIC INDUSTRIES, INC. AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS.

Products sold by United Electronic Industries, Inc. are not authorized for use as critical components in life support devices or systems. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. Any attempt to purchase any United Electronic Industries, Inc. product for that purpose is null and void and United Electronic Industries Inc. accepts no liability whatsoever in contract, tort, or otherwise whether or not resulting from our or our employees' negligence or failure to detect an improper purchase.

NOTE: Specifications in this document are subject to change without notice. Check with UEI for current status.

Table of Contents

Table of Figures

Chapter 1 Introduction

This document outlines the feature-set and describes the operation of the DNx-1553-553 Communication Interface boards. The DNA- version is designed for use with a PowerDNA Cube data acquisition system. The DNR- version is designed for use with a DNR-12 RACKtangle or DNR-6 HalfRACK rack-mounted systems. Both versions handle 1553 messaging over a dual redundant MIL-1553 bus. Please ensure that you have the PowerDNA Software Suite installed before attempting to run examples.

1.1 Organization of this manual

The DNx-1553-553 User Manual is organized as follows:

- **Introduction**
  This chapter provides an overview of the document content, the device architecture, connectivity, and logic of the layer. It also includes connector pinout, notes, and specifications.
- **Programming with the High-Level API**
  This chapter describes the use of the UeiDaq Framework High-Level API for programming the board. It includes information such as how to create a session, configure the session for 1553 bus communication, and interpret results.
- **Programming with the Low-Level API**
  This chapter describes the use of low-level API commands for configuring and using the DNx-1553-553 series boards.
- **Appendix A: Accessories**
  This appendix provides a list of accessories available for DNx-1553-553 board(s).
- **Index**
  This is an alphabetical listing of the topics covered in this manual.

Document Conventions

To help you get the most out of this manual and our products, please note that we use the following conventions:

Tips are designed to highlight quick ways to get the job done, or to reveal good ideas you might not discover on your own.

NOTE: Notes alert you to important information.

CAUTION! *Caution advises you of precautions to take to avoid injury, data loss, and damage to your boards or a system crash.*

Text formatted in **bold** typeface generally represents text that should be entered verbatim. For instance, it can represent a command, as in the following example: “You can instruct users how to run setup using a command such as **setup.exe**.”

Frequently Asked Questions

For frequently answered questions, application notes, and support, visit us online at:

http://www.ueidaq.com/faq/

1.2 The MIL-1553 Interface Boards

The DNx-1553-553 interface boards have the following features:

- 2 independent channels/ports
- Dual redundant bus interfaces
- Each channel is independently software-configurable as a Bus Controller (BC), a Remote Terminal (RT), a Bus Monitor (MT) or combination Remote Terminal/Monitor (RT/MT)
- Transformer-coupled Bus Interface standard, (Direct coupling software selectable)
- Supports 1553A and 1553B protocols (Notice 1 and/or 2)
- Multiple RT simulation up to 31 RTs
- Completely independent bit rate settings for every port
- 350 Vrms isolation between 1553 bus, other I/O ports, and chassis
- Selective Message Monitoring in MT mode based on RT address or Mode Code

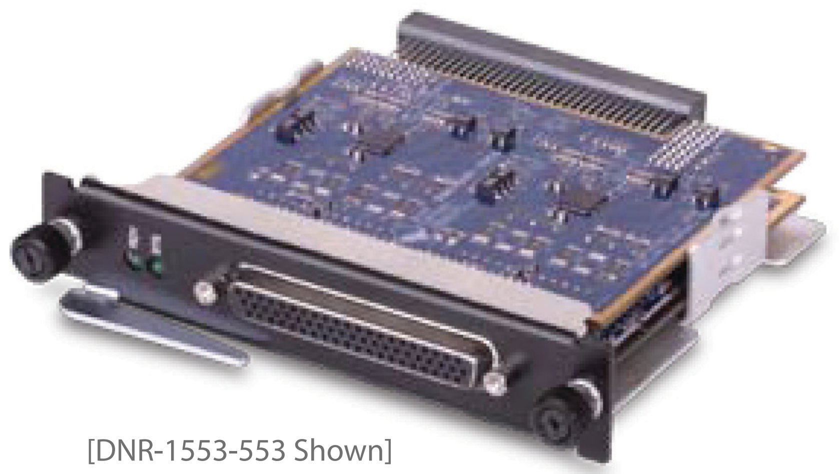

[DNR-1553-553 Shown]
DNA version is functionally identical with different bus connector and different front panel

Figure 1-1. Photo of DNR-1553-553 Interface Module

1.3 What is MIL-STD-1553?

MIL-STD-1553 is a military standard that defines mechanical, electrical, and operating characteristics of a serial data communication bus for the U.S. Department of Defense.It is now commonly used for both military and civilian applications in avionics, aircraft, and spacecraft data handling. A 1553 system typically uses a dual redundant, balanced-line, physical layer with a differential network interface with time division multiplexing, half-duplex, command/ response data communication protocol with up to 31 remote terminal devices. It was first used in the F-16 fighter aircraft and is now widely used by all branches of the U.S. military and NATO.

The current standard, MIL-STD-1553B was introduced in 1978, the goal of which was to define explicitly how each option should function, so that compatibility among manufacturers could be guaranteed.

1.3.1 Physical Layer

A single 1553 bus consists of a shielded twisted-wire pair with 70 - 85 ohm impedance at 1 MHz. If a coaxial connector is used, the center pin is used for the high Manchester bi-phase signal. All transmitter and receiver devices connect to the bus either through coupling transformers or directly through stub connectors and isolation transformers, as shown in **Figure 1-2**. Stubs can be a maximum of 1 foot in length for direct coupling or a maximum of 20 feet for transformer coupling. To limit reflections, the data bus must be terminated by resistors equal to the cable characteristic impedance (within ±2%). **Figure 1-2** shows one of the two buses. Each transceiver is also connected in the same way to the second (redundant) bus. Note that although a 1553 system may also have additional redundant buses, the DNx-1553-553 interface module is designed for systems with two buses only.

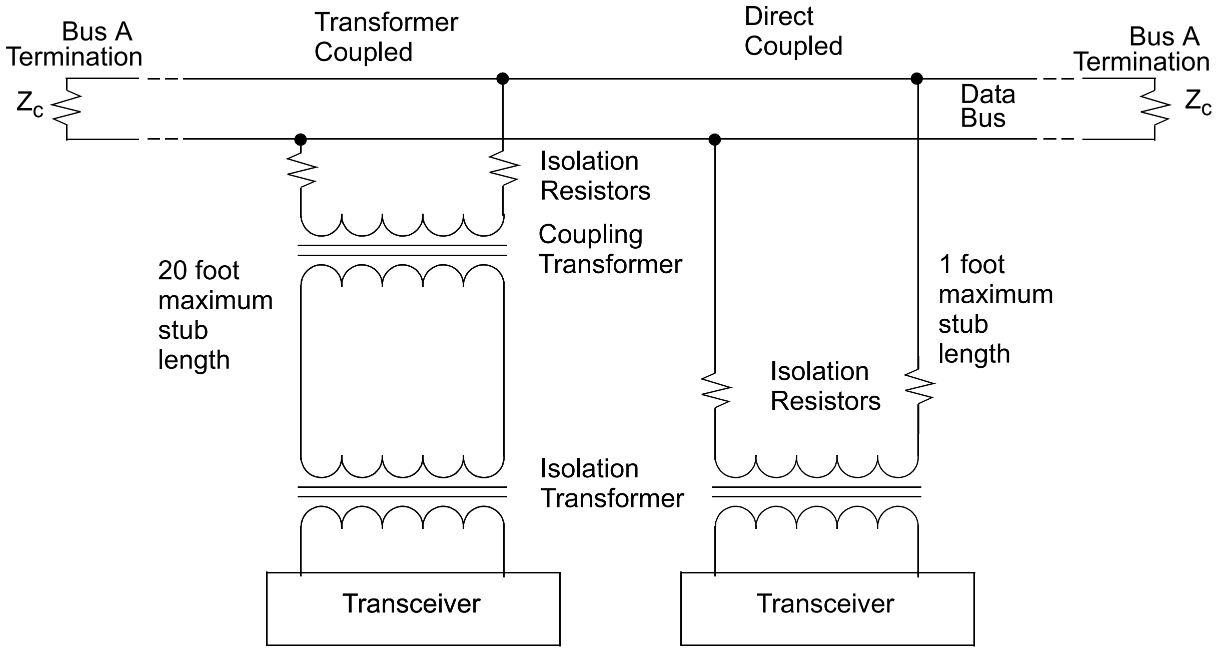


Figure 1-2. Terminal Connection Types (One Bus Shown)

Although multi-stub couplers may be used in place of individual couplers according to the standard, this does not apply when the DNx-MIL-1553 board is used because the DNx-1553-553 interface module is designed with software-selectable direct or transformer coupling and has termination resistors built in. The data bus, therefore can be connected directly to the I/O connector of the interface module.

1.4 Bus Protocol

All 1553 messages on the bus contain one or more 16-bit words, classified as command, data, or status word types. Each word is preceded by a 3 µs sync pulse and is followed by an odd parity bit. Note that since the sync pulse — 1.5 µs low followed by 1.5 µs high — cannot occur in a Manchester code, it is therefore unique. The words in a message are transmitted with no gap between words, but a 4 µs gap is inserted between successive messages. All devices must start transmitting a response to a command within 4 to 12 µs. If they do not start transmitting within 14 µs, they are considered to have not received the command message.

A typical system is illustrated in **Figure 1-3**.

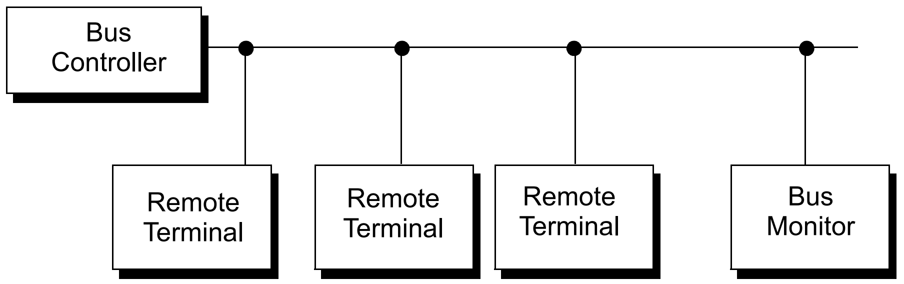


Figure 1-3. Typical MIL-STD-1553 Bus

All communication on the bus is controlled by the master bus controller (BC). A Remote Terminal (RT) can receive or transmit only in response to a command from the bus controller. The sequence of words in the message from the controller to a terminal for transfer of data is:

*master.command(terminal) --> terminal.status(master) -->
master.data(termiinal) --> master.command(terminal) -->terminal*

where the notation is *<source>.<word_type>(destination)>.*

The message to initiate terminal-to-terminal communication is:

*master.cmmand(terminal_1) --> terminal_1.status(master) -->
master.command(termiinal_2) -->terminal_2.status(master) -->
master.command(terminal_1) --> terminal_1.data(terinal_2) -->
master.command(terminal_2) --> terminal_2.status(master)*

The sequences shown above ensure the terminal is functioning and ready to receive data; the status request at the end of a transfer confirms that the data was received and accepted. This sequence is the basis for the high reliability of a 1553 data transfer. Note, however, that the sequences described do not illustrate the actions that are taken under fault conditions, which are more complex than those shown in the example.

Note that a remote terminal cannot initiate a data transfer by itself. It can only request a data transfer in response to a poll by the master controller. Priority of such requests is determined solely by the frequency of polling designed into the system. If a terminal does not respond to a poll, this indicates existence of a system fault.

Five types of transactions can occur between a Bus Controller (BC) and Remote Terminals (RT), as follows:

- **Receive Data**
  The Bus Controller (BC) sends one 16-bit command word to an addressed terminal, followed by 1 to 32 16-bit data words. The selected terminal then sends a single 16-bit Status Word back to the Bus Controller.
- **Transmit Data**
  The Bus Controller (BC) sends one 16-bit command word to an addressed terminal, followed by 1 to 32 16-bit data words. The selected terminal then sends a single 16-bit Status Word back to the Bus Controller, immediately followed by 1 to 32 data words.
- **Broadcast Data**
  The Bus Controller (BC) sends one command word with a Terminal Address of 31, which signifies a broadcast type of command, followed by 1 to 32 data words. All terminals accept the data, but do not send a response. This function is used for system updates such as time of day, etc.
- **Mode Code**
  The Bus Controller sends one command word with a subaddress of either 0 or 31, signifying a Mode Code command type. Depending on which command is sent, the command may or may not be followed by a single word. The Remote Terminal responds with a Status Response word, which may or may not be followed by a single data word.
- **RT to RT Transfer**
  The Bus Controller sends a Receive Data command followed by a Transmit Data command to a specific terminal. The terminal sends a Status Word followed by 1 to 32 data words to a specified receiving terminal. The receiving terminal the sends its Status Word, indicating that it received the transmitted data

1.5 Word Formats

The 1553 standard defines three word types:

- Command
- Data
- Status

Each word type has a specific format within a common structure. All words are 20 bits in length and the first three bits are a synchronization field, which enables the decoding clock to re-sync at the beginning of each new word. The next 16 bits contain the information, in a format that varies with the word type. The last bit in the word is a parity bit, which is based on odd parity for a single word.

All bit encoding is based on bi-phase Manchester II format, which provides a self-clocking waveform. The signal is symmetrical about zero and is therefore compatible with transformer coupling.

In Manchester coding, signal transitions occur only at the center of a bit time. A logic “0” is defined as a transition from negative to positive level; a logic “1” is the reverse. Note that the voltage levels on the bus are ***not*** the information signal; all information is contained in the timing and direction of the ***zero crossings*** of the signal on the bus.

The terminal hardware provides the encoding and decoding of the various word types. The encoder also calculates parity. For received messages, the decoder signals the logic what sync type a word is and whether or not parity is valid. For transmitted messages, input to the encoder defines what sync type to place at the beginning of a word. The encoder calculates parity automatically for each word.

The formats for each word type are illustrated in **Figure 1-4**.

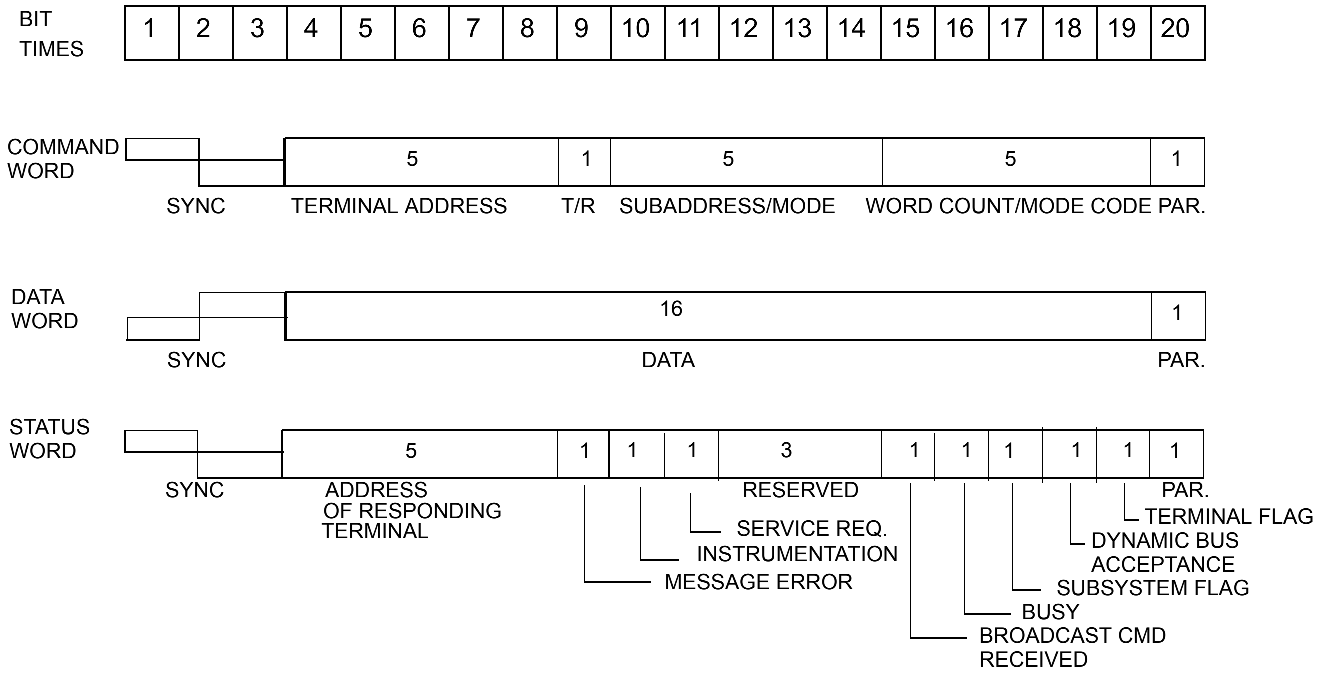


Figure 1-4. 1553 Word Formats

1.5.2 Command Word

A Command Word format uses the first 5 bits for the address of the Remote Terminal (0 to 31). The sixth bit is 0 for Receive and 1 for Transmit. The next 5 bits indicate the subaddress/mode code bits. If this field is a 00000B or 11111B, the command is a Mode Code Command. All other bits direct the data to specific functions in the subsystem. The next 5 bits define the Word Count or Mode Code to be performed. If this field is 00000B or 1111B, the field defines a mode code to be performed. If it is not, the field defines the number of data words to be transmitted or received (depending on the T/R bit). A word count field of 00000B means 32 data words.

The last bit is word parity. Only odd parity is used.

1.5.3 Data Word

A data word contains the information being transferred in a message. The first 3 bit times contain a data sync, which is opposite to that used for a command or status word.

Data words can be transmitted by either a remote terminal (transmit command) or a bus controller (receive command). The remote terminal is the reference point.

The next 16 bits may be used however the designer wishes. The only standard requirement is that the most significant bit must be transmitted first.

The last bit is an odd parity bit.

1.5.4 Status Word

A remote terminal responds to a valid message by transmitting a status word. The status word tells the bus controller whether or not a message was received properly and what the state of the terminal is.

The status word is cleared after receiving a a valid command word. After the status word is cleared, the bits are set again if the conditions that set the bits initially still exist.If an error is detected in the data, the Message Error bit is set and transmission of the status word is suppressed. Transmission of the status word is also suppressed whenever a broadcast message is received.

The first 5 bits of the status word (bits 4-8) are the Terminal Address. The remote terminal sets these bits to the address to which it has been programmed. The bus controller examines these bits to ensure that the responding terminal is the one to which the command word was addressed.

The next bit (9) is the Message Error bit, which is set by the terminal on detection of an error or an invalid message. Whenever this bit is set, none of the data received in the message is used. When an error is detected, the remote terminal must suppress transmission of the status word.

The Instrumentation bit (10) differentiates a command word form a status word (both have the same sync pattern). The instrumentation bit in a status word is always set to ”0”. When used, the instrumentation bit in a command word is always set to “1”. SInce this bit is the most significant bit of the subaddress field, using it as an instrumentation bit reduces the number of available subaddresses from 30 to 15. Because of this limitation, most systems today use techniques other than the instrumentation bit to differentiate between command and status words.

The Service Request bit (11) enables a terminal to inform the bus controller that it needs to be serviced. A “1” in this bit indicates that service is needed. It is typically used when the bus controller is polling the terminals.

Bits 12 - 14 are reserved for future use and must be set to “0”. Any other value is an error.

A “1” in Bit 15 indicates that the terminal received a valid broadcast command. Whenever a terminal receives a valid broadcast command, the terminal sets this bit to “1” and suppresses transmission of its status word.

A “1” in Bit 16 (“busy” bit) tells the Bus Controller that the terminal cannot act on a command to move data between the terminal and subsystem. This bit is typically not used in modern system designs and is discouraged by Notice 2 of the standard.

A “1” in Bit 17 is used as an indicator of existence of a fault in a subsystem. A “1” in Bit 18 indicates that the remote terminal has received a mode code and has accepted control of the bus. After setting this bit, the remote terminal becomes the bus controller.

A “1” in Bit 19 (the Terminal Flag) indicates to the bus controller that a fault exists in the remote terminal hardware.

1.6.5 Remote Terminal (RT)

Any device that is not a Bus Controller or a Bus Monitor is, by definition, a Remote Terminal.

A Remote Terminal can be used as an interface between the bus(es) and a subsystem or as a connector between this bus and another 1553 bus. A subsystem is the sender or user of the information transferred on the bus. A remote terminal contains all the components needed to transfer the data from the sender source to the destination user.

1.6.6 Bus Controller (BC)

The Bus Controller (BC) manages the flow of data on the buses. Only one bus controller can be active at a given time. A bus controller may be one of three types: (1) Word Controller, (2) Message Controller, or (3) Frame Controller.

A Word controller, which handles one word at a time, is seldom used today because of the processing burden it places on the subsystem.

A Message Controller handles one message at a time, interacting with the computer only when a message is complete or when a fault occurs.

A Frame Controller can process multiple message in a defined sequence, interrupting the computer only when the message stream is complete or after an error is detected.

1.6.7 Bus Monitor (MT)

A Bus Monitor is not able to transmit messages on the bus; its function is to monitor and record messages being transmitted on the bus without disrupting other devices. A bus monitor can be set up to record selected subsets of the messages on the bus. It can also be set up as a backup bus controller, ready to take over whenever needed.

1.6.8 Monitor Terminal with RT Address (MT/RT)

When a monitor with RT address (MT/RT) receives a command addressed to its terminal address, it responds as a remote terminal. For all other commands, it responds as a monitor. The remote terminal functions may be used to modify the operation of a specific monitor.

1.7 DNx-1553-553 Architecture

Figure 1-5 is a block diagram of the architecture of the DNx-1553-553 interface module.

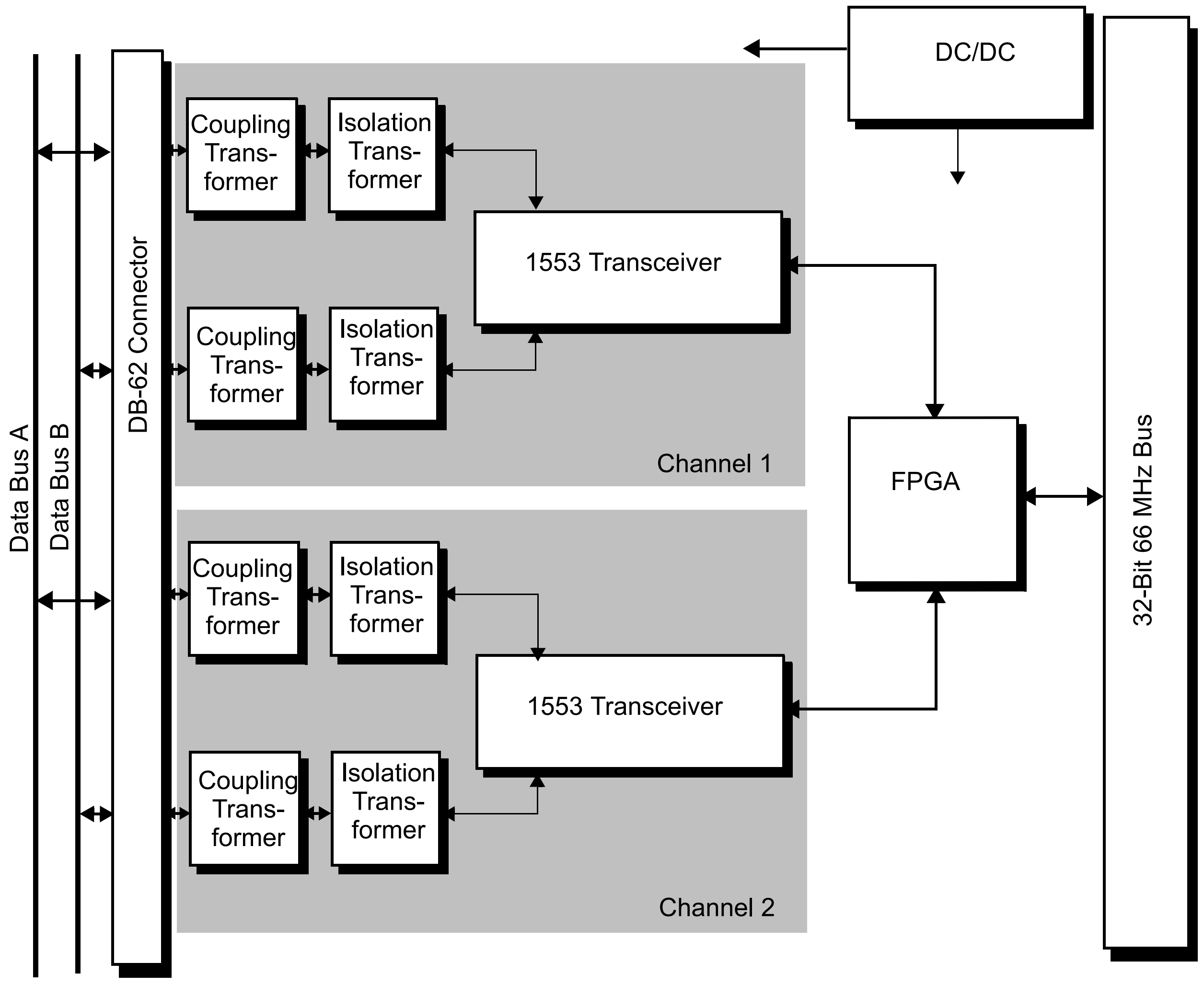


Figure 1-5. Block Diagram of the DNx-1553-553 Interface Module

1.7.1 Functional Description

As shown in **Figure 1-5**, the module has two independent 1553 channels, each connected to a dual-redundant data bus via coupling/isolation transformers or isolation transformers only, and each with an independent dedicated transceiver. The coupling transformers and the transformer taps can be selected by opto-isolated relays under program control. (Note that different transformer ratios are used for direct or transformer coupling.) Each channel can function independently as a Bus Controller, Remote Terminal, or Bus Monitor, as determined by the software (subject to the restrictions of the applicable MIL standard).

When the channel is operating as a transmitter, messages are sent from the host through the DNx-1553-553 module to the selected Data Bus. The address contained in the message itself determines the destination of the message. The transceiver transmitter accepts Manchester-encoded biphase data and converts it to differential voltages that are passed to the bus through the isolation and coupling transformers (or through an isolation transformer circuit only, if direct coupling is selected).

When the channel is operating as a receiver, the process described above is reversed.

1.8 Specifications

Technical Specifications:

General Specifications	
Number channels/ports	2, Independent
Channel configuration	Dual redundant interfaces
Specification compliance	MIL-STD-1553a or MIL-STD-1553b including notices 1 & 2
Configuration	Bus Controller (BC), Remote Terminal (RT) or Bus Monitor (BM). *Either channel may be set as BC, RT or BM*
Interface	Transformer or direct coupling (software selectable)
Isolation	350 Vrms
Power Consumption	5 W (not including load)
Bus Controller (BC) Specs	
Configuration	Independent Ports
Communication support	BC to RT, RT to BC, RT to RT
Messaging protocols	Standard Mode Codes, Broadcast messages
Message timing	Scheduled or asynchronous with two levels of priority
Programmability	Major/minor frame timing, intermessage gap times, time out and late response, BC retries
Error handling	Automatic error detection and recovery.
Remote Terminal (RT) Specs	
Modes	Single or multiple RT emulator (up to 31 different RTs) RT - RT xfers with simulated RTs may be implemented with user software.
RT/BM joint mode	Allows the unit to act as an RT while logging data as an BM
Error handling	Automatic error detection and insertion.
Bus Monitor (BM) Specs	
Monitor modes	Full or selective monitoring by RT address
Monitored parameter	In addition to bus data, BM mode time tags data and capture Word/Message/Error status and RT response time
Environmental	
Operating Temp. (tested)	-40°C to +85°C
Operating Humidity	0 - 95%, non-condensing
MTBF	275,000 hours
Vibration *IEC 60068-2-6* *IEC 60068-2-64*	5 g, 10-500 Hz, sinusoidal 5 g (rms), 10-500 Hz, broad-band random
Shock *IEC 60068-2-27*	50 g, 3 ms half sine, 18 shocks @ 6 orientations 30 g, 11 ms half sine, 18 shocks @ 6 orientations
Altitude	0 - 70,000 feet (0 - 21,336 m)

Figure 1-6. DNx-1553-553 Specifications

1.9 Software

Refer to Chapters 2 and 3 for information on how to program the DNx-1553-553 Interface Module.

UEIDAQ Framework is a programming facility that allows you to connect transparently to a cube or RACKtangle and then configure the DNx-1553-553 module to communicate with other 1553 devices. Attach to the 1553-bus(es) and then send and/or receive 1553 packets.

1.10 Wiring & Connectors

Figure 1-7 illustrates the pinout of the DB-62 connector on the DNx-1553-553 Interface Module.

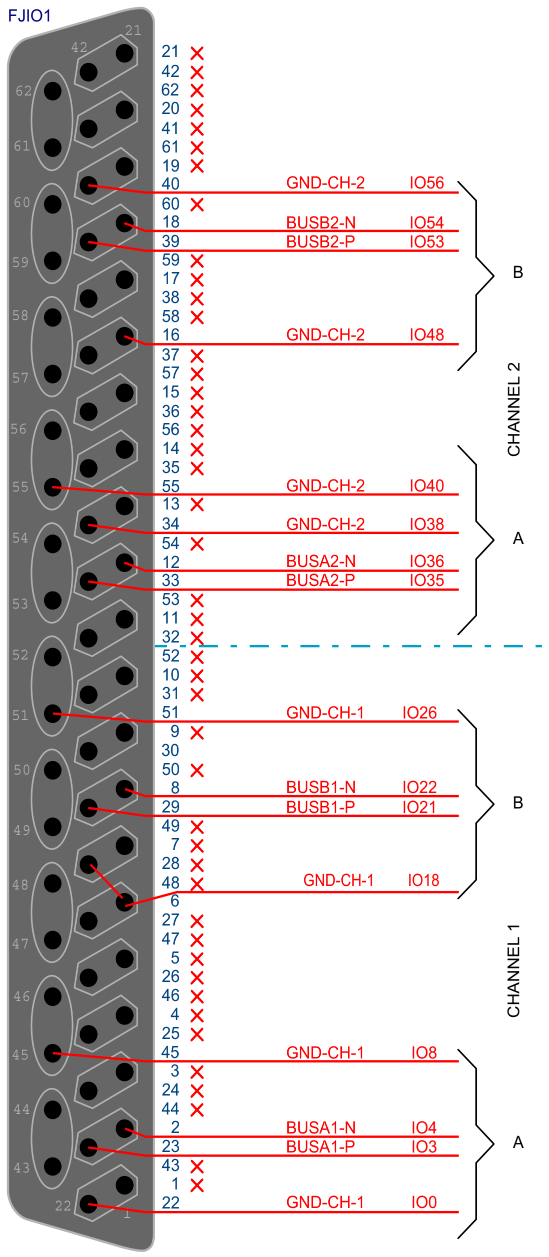
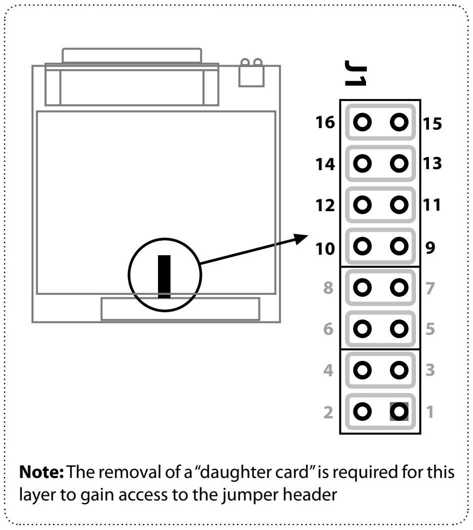


Figure 1-7. Pinout Diagram for DNX-MIL-1553 Layer

1.11 Jumper Settings for Module Position

Figure 1-8 shows the physical layout of baseboard of the DNx-1553-553 Interface Module, highlighted to show the 16-pin jumper block for setting the module position within the PowerDNA Cube.

NOTE: Layer position jumpers are not provided with the DNR versions of the DNx-1553-553. The physical position of the layer within the DNR RACKtangle^{TM} enclosure is determined automatically by the system.


J1
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
Note: The removal of a "daughter card" is required for this layer to gain access to the jumper header

See **Figure 1-9** for placement of jumpers for various layer positions in a Cube.

Figure 1-8. Jumper Block on Base Board for DNA-MIL-1553 Layer Position

1.11.0.1 Jumper Settings

A diagram of the jumper block is shown in **Figure 1-9**. To set the layer position jumpers, place jumpers as shown in **Figure 1-9**.

NOTE: Since all layers are assembled in Cubes before shipment to a customer, you should never have to change a jumper setting unless you change a layer from one position to another in the field.

		Layer's Position as marked on the Faceplate*					
		I/O **1**	I/O **2**	I/O **3**	I/O **4**	I/O **5**	I/O **6**
Jx Pins	9-10	■	○	■	○	■	○
	11-12	■	■	○	○	■	■
	13-14	■	■	■	■	○	○
	15-16	■	■	■	■	■	■

* *All I/O Layers are sequentially enumerated from top to the bottom of the Cube*

○ - **Open** ■ - **Closed**

Figure 1-9. Diagram of DNA-MIL-1553 Layer Position Jumper Settings

Chapter 2 Programming with the High Level API

This section describes how to program a DNx-1553-553 layer using the UeiDaq Framework High-Level API. UeiDaq Framework is object-oriented and its objects can be manipulated in the same manner in various development environments, such as Visual C++. Visual Basic, LABView, or DASYLab.

UeiDaq Framework is bundled with examples for supported programming languages. They are located under the UEI Programs group in

Start >> Programs >> UEI >> Framework >> Examples

The following subsections focus on the C++ API, but the concept is the same regardless of the programming language used.

Please refer to the "UeiDaq Framework User Manual" for more information on using other programming languages.

2.1 Creating a Session

The Session object controls all operations on your PowerDNA device. Therefore, the first task is to create a session object:

```
CUeiSession session;
```

2.2 Create MIL-1553 Ports

MIL-1553 ports are configured using the session object's method ***"CreateMIL1553Port",*** as follows:

```
// port 0 - bus controller
CUeiMIL1553Port* pPort0 = session.CreateMIL1553Port(
                                        "pdna://192.168.100.2/dev0/milb0",
                                        UeiMIL1553CouplingTransformer,
                                        UeiMIL1553OpModeBusController
                                        );

// port 1 - remote terminal
CUeiMIL1553Port* pPort1 = session.CreateMIL1553Port(
                                        "pdna://192.168.100.2/dev0/milb1",
                                        UeiMIL1553CouplingTransformer,
                                        UeiMIL1553OpModeRemoteTerminal
                                        );
```

Each created port can be used in one of three modes (but there can be only one Bus Controller port):

- ***UeiMIL1553OpModeBusMonitor*** – a Bus Monitor port allows you to receive ongoing activity on the bus using a ***CUeiMIL1553Reader*** object. In this mode of operation, the ***CUeiMIL1553Writer*** object also allows you to send unscheduled continuous data on the bus. In ***UeiMIL1553OpModeBusMonitor***, a user can monitor a bus and send messages using ***BusWriter***. Bus monitor is enabled in all configurations, including RT and BC.

- ***UeiMIL1553OpModeRemoteTerminal*** – An RT port allows you to program remote terminals and to send/receive data from the remote terminal data memory. In ***UeiMIL1553OpModeRemoteTerminal***, up to 32 RTs (including the 31st broadcast RT) and 30 SAs per each RT (SA0 and SA31 are reserved for mode commands).
- ***UeiMIL1553OpModeBusController*** – A Bus Controller port allows you to program a bus controller /scheduler and to send and receive data from and to bus controller data memory. In ***UeiMIL1553OpModeBusController*** mode, a user can program bus controller operation.

Note that ***BusWriter*** does not work in RT or BC mode and that Bus Monitor works in all modes.

Bus Controller and Remote Terminal mode cannot be used on the same port simultaneously, but can be used on the same layer on different ports of an aircraft simulation MIL-1553 network.

Each port created can be used in one of four coupling modes:

- ***UeiMIL1553CouplingDisconnected*** – port is completely disconnected from the bus.
- ***UeiMIL1553CouplingTransformer*** – normal mode of operation.
- ***UeiMIL1553CouplingLocalStub*** – isolation coupler of the layer, which requires a special version of the hardware.
- ***UeiMIL1553CouplingDirect*** – direct connection without isolation transformer. Sometimes used in laboratories when a coupled network is not available. This mode is normally not recommended.

The normal coupling is ***UeiMIL1553CouplingTransformer***.

The ***UeiMIL1553CouplingLocalStub*** coupling option requires a local stub to be populated on the 1553 board (this is an extra cost option). Direct coupling is used sometimes a in laboratory environment but is extremely rare, probably in cases in which there is no network and an RT is connected directly to the board using two wires.

Note that you will need to create one reader and one writer per port to access port data in any mode of operation.

A user can select which bus on which to transmit data using the ***CUeiMIL1553Port::SetRxBus()*** method. Note that for transmission of messages, either bus A or bus B should be selected (default is bus A). ***CUeiMIL1553Port::SetRxBus()*** selects which bus to listen to. Table 2-1 on page 17 describes the bus settings that are allowed for various modes of operation:

Table 2-1. Selection of Transmit Bus

Settings	BM (BW)		RT		BC	
	Listen (Rx)	**Transmit (Tx)**	**Listen (Rx)**	**Transmit (Tx)**	**Listen (Rx)**	**Transmit (Tx)**
Rx A	A only		A only		A only	
Rx B	B only		B only		B only	
Rx Both	A and B (normal)		A and B (normal)		A and B (normal)	
Tx A		A only		A only		A only
Tx B		B only		B only		B only
Tx Both		prohibited		A or B (reply on the bus on which command was received)		A or B (first use A and then B upon bus error, if error recovery is enabled)

2.3 Configure Timing

On MIL-1553 ports, messages are represented in a ***tUeiMIL1553*Frame*** structures. Note that the same approach is used for all modes of operation. Bus monitor, bus controller, and remote terminal timing depend on the frame type specified.

```
session.ConfigureTimingForMessagingIO(1, 0);
session.GetTiming()->SetTimeout(1000);
```

Asynchronous operations with MIL-1553-553 layers are currently not implemented. If a user needs to operate RT or BM asynchronously, the best way is to use it in a separate thread.

2.4 Creating a Reader Object and Writer Object for each Port

Before a user can communicate with the layer reader and (for everything except a BM), writer objects need to be created for each port, as shown below:

```
CUeiMIL1553Reader* readers = new CUeiMIL1553Reader(session.
GetDataStream(), session.GetChannel(ch)->GetIndex());
CUeiMIL1553Writer* writers[ch] = new CUeiMIL1553Writer(session.
GetDataStream(), session.GetChannel(ch)->GetIndex());
```

By doing this, you are creating MIL-1553-553 specific Reader and Writer classes and connecting them to the appropriate data stream and channel.

The default behavior of reader and writer objects is to block until the specified number of frames is ready to be transferred. You can also configure those objects to work asynchronously. The method used to program readers and writers asynchronously is very dependent on the programming language. You can find more information on how to do this in the Reference manual for each development environment.

CUeiMIL1553Reader and ***CUeiMIL1553Writer*** are polymorphic. There are multiple overloaded implementations of these functions that can accept different types of frames. The type of frame dictates the data passed and the operation to be performed.

The following types of frames are defined:

- ***CUeiMIL1553BCSchedFrame*** class – program major and minor BC frames.

This class is used to construct and manipulate ***tUeiMIL1553BCSchedFrame*** to simplify its use. It can be used with both ***CUeiMIL1553Reader::read()*** and ***CUeiMIL1553Writer::write()***

```
class CUeiMIL1553BCSchedFrame : public tUeiMIL1553BCSchedFrame
```

- ***CUeiMIL1553BCCBDataFrame*** class – assemble and store data into a BC control block.

This class is used to construct and manipulate ***CUeiMIL1553BCCBDataFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Writer::write().***

```
class CUeiMIL1553BCCBDataFrame : public tUeiMIL1553BCCBDataFrame
```

- ***CUeiMIL1553BCCBStatusFrame*** class – request and manipulate BC control block status data as well as the status and data the RT replied with.

This class is used to construct and manipulate ***CUeiMIL1553BCCBStatusFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Reader::read().***

```
class CUeiMIL1553BCCBStatusFrame : public tUeiMIL1553BCCBStatusFrame
```

- ***CUeiMIL1553RTFrame*** class – write or read RT "send" and "transmit" data areas.

This class is used to construct and manipulate a ***CUeiMIL1553RTFrame*** to simplify its use. It can be used with both ***CUeiMIL1553Reader::read()*** and ***CUeiMIL1553Writer::write()***

```
class CUeiMIL1553RTFrame : public tUeiMIL1553RTFrame
```

- ***CUeiMIL1553BMFrame*** class – read BM messages separated by idle state of the bus.

This class is used to construct and manipulate a ***CUeiMIL1553BMFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Reader::read()***.

```
class CUeiMIL1553BMFrame : public tUeiMIL1553BMFrame
```

- ***CUeiMIL1553BMCmdFrame*** class – read BM messages separated by 1553 protocol commands.

This class is used to construct and manipulate ***CUeiMIL1553BMCmdFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Reader::read().***

```
class CUeiMIL1553BMCmdFrame : public tUeiMIL1553BMCmdFrame
```

- ***CUeiMIL1553RTStatusFrame*** class – write or read RT "send" and "transmit" data areas.

This class is used to construct and manipulate ***CUeiMIL1553RTStatusFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Reader::read().***

```
class CUeiMIL1553RTStatusFrame : public tUeiMIL1553RTStatusFrame
```

- ***CUeiMIL1553RTControlFrame*** class – run-time control of RT.

This class is used to construct and manipulate ***CUeiMIL1553RTControlFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Writer::write().***

```
class CUeiMIL1553RTControlFrame : public tUeiMIL1553RTControlFrame
```

- ***CUeiMIL1553FilterEntry*** class – run-time control of RT.

This class is used to construct and manipulate ***CUeiMIL1553FilterEntry*** to simplify its use. It can be used only with ***CUeiMIL1553Writer::write().***

```
class CUeiMIL1553FilterEntry : public tUeiMIL1553FilterEntry
```

- ***CUeiMIL1553RTParametersFrame*** class – specify RT parameters during initialization

This class is used to construct and manipulate ***CUeiMIL1553RTParametersFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Writer::write()***.

```
class CUeiMIL1553RTParametersFrame : public tUeiMIL1553RTParametersFrame
```

- ***CUeiMIL1553TxFifoFrame*** class – write data to BusWriter.

  This class is used to construct and manipulate ***CUeiMIL1553TxFifoFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Writer::write()***.

```
class CUeiMIL1553TxFifoFrame : public tUeiMIL1553TxFifoFrame
```

- ***CUeiMIL1553BCStatusFrame*** class – receive BC status information

  This class is used to construct and manipulate ***CUeiMIL1553BCStatusFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Writer::read()***.

```
class CUeiMIL1553BCStatusFrame : public tUeiMIL1553BCStatusFrame
```

- ***CUeiMIL1553BCControlFrame*** class – control BC execution process.

  This class is used to construct and manipulate ***CUeiMIL1553BCStatusFrame*** to simplify its use. It can be used only with ***CUeiMIL1553Writer::write()***.

```
class CUeiMIL1553BCControlFrame : public tUeiMIL1553BCControlFrame
```

Frame type ***UeiMIL1553FrameTypeBusMon*** is used to receive data from the bus monitor. Each command and status word on the bus is stored in a separate frame.

Call the session object's method **"ConfigureTimingForMessagingIO"** to perform message communication, provided that the device allows it.

2.5 Starting the Session

You can start the session by calling the session object's method "Start":

```
// Start the session
mySession.Start();
```

Note that if you don't explicitly start the session, it will be automatically started the first time you try to transfer data using a reader or writer object.

2.6 Reading/ Writing Data from/to a Device

To write or read data to/from a MIL-1553-553 board, do the following:

```
CUeiMIL1553RTDataFrame* pFrame = new CUeiMIL1553RTDataFrame;
writer->Write(numFramestoWrite, pFrame, &numFramesWritten);
reader->Read(numFramestoRead, pFrame, &numFramesRead);
```

Note that the first read or write from a 1553 channel configures all operations and starts the layer.

2.6.1 Reading Bus Monitor

Bus Monitor is the simplest function to use.

To do so, first create a new bus monitor frame:

```
CUeiMIL1553BMFrame* bmFrm = new CUeiMIL1553BMFrame;
```

Then read bus monitor data from that accumulated in the 1553 bus monitor buffer (it can accumulate up to 1024 32-bit data words).

```
readers[1]->Read(1, bmFrm, &numFramesRead);
```

You can either work with the frame members directly or use helper methods to display data as strings:

```
if (numFramesRead) std::cout << bmFrm->GetBmDataStr() << std::endl;
```

Raw bus monitor data is represented as follows:

First 32-bit word: —

bit 31: parity error on the bus, if any
bit 30: set to 1 for command or status
bits 29 thru 16: time in 15.15ns interval since previous command or status.
bits 15 thru 0: command or status as received from the bus.

If there is data following the command, it is represented in the following format:

bit 31: parity error on the bus, if any
bit 30: set to 0 for data word
bits 29 thru 16: time in 15.15ns interval since previous command or status.
bits 15 thru 0: data as received from the bus.

If timestamps are enabled using the ***CUeiMIL1553Port::EnableTimestamping()*** method, (timestamps are enabled by default), the last two words contain a 32 bit “absolute” timestamp of the message in 10us resolution (timestamps are reset when the session starts) and the various flags defining the current bus status and which bus (A or B) the command was received on. See “PowerDNA API Reference Manual” for further detail.

A Bus Monitor works as follows:

Each time a command/status word is decoded on the 1553 A/B bus, it is validated against an RT filter. If the RT address is not included in the monitoring, all command/status and consequential data are ignored. If the RT address is included into the monitoring, the timestamp is stored into the internal register and the command/status with gap timeout counter is stored into the FIFO. After that, each received data word is stored into the FIFO until the next command/ status word is received and the process repeats itself. Once a data gap interval is detected OR another control/status word is received, the following optional information is stored into the BM FIFO:

- Timestamp word (30 LSBs of the timestamp) – optional
- Flags/status word (status information and extra bits of the timestamp if enabled)
- Bus IDLE tag, 0xC0000000, which indicates that the 1553 bus that was driving the BM went into the idle state. The idle tag is used internally in the firmware to separate messages and is not exposed in the datastream into ***CUeiMIL1553BMFrame***.
- Note that when BMALL bit is cleared (normal operation), the BM follows messages from bus A or B, but only one bus at the time. If for any reason, in violation of the 1553 protocol, any device sends data on both buses, only the transmission from one bus will be logged, data from the other bus will be processed upon the “first” bus going into the idle state; in the case in which more than one word was received prior to switching to the “second” bus, the data from the “second” bus will be corrupted.

BM data structure:

Command/Status, all 32-bits used, bit 30=1
Data word 0
Data word N
Optional: Timestamp[29:0] , bit 30=0
Optional: Flags[29:0] , bit 30=0

Command/Status/Data format in BM data

Bit	Name	Description
31	PARITY	One in this bit indicates parity error
30	WORD_TYPE	Word type (1=command/status, 0=data)
29-16	GAP1553	Gap interval on 1553 bus measured in 66MHz clocks, 248.2uS max; 0x3FFF indicates that gap counter has expired
15-0	DATA1553	1553 data from/to the decoders

Timestamp “LSB part” word format in BM data

Bit	Name	Description
31-30	ZERO	Upper two bits are always zeroes for the flags
29-0	TIME_LSB	30 LSBs of the timestamp tag

Flag word format in BM data

Bit	Name	Description
31-30	ZERO	Upper two bits are always zeroes for the flags
29	RES29	Reserved for future use
24	RES24	Reserved for future use
23	OVRE	=1 if decoder data overrun was detected
22	PE	=1 if parity error was detected
21	DBE	=1 if error detected during data bits reception
20	SBE	=1 if error detected during SYNC bit reception
19	ZCE	=1 if invalid combination is detected (positive line <> !negative line) ~ 150nS after zero crossing
18	SET	=1 if edge-edge timing is invalid for the SYNC bit
17	DET	=1 if edge-edge timing is invalid for the data bit
16	TOUT	=1 if timeout is detected while waiting for the edge on positive and negative input lines
15-1	TIME_MSB	15 MSBs of the 45-bit timestamp tag (currently bits15-3 are reserved and bits 2-1 contain upper two bits of the timestamp)
0	BUSID	1553 bus ID 1=A, 0=B

`SetTxBus(tUeiMIL1553PortActiveBus portBus)` can be used to select which bus to listen on – A or B or both. By default, a bus controller listens to communication on both buses.

2.6.2 Programming BusWriter Mode

In Bus Monitor mode, a user can send arbitrary data packets on the bus using the BusWriter mechanism.

To use BusWriter, an appropriate frame needs to be created first:

```
CUeiMIL1553TxFifoFrame* outFrm = new CUeiMIL1553TxFifoFrame;
```

Then, the frame needs to be filled with appropriate information:

```
outFrm->CopyData(messageSize, data);
outFrm->SetCommand(startRt, startSa, messageSize, UeiMIL1553CmdBCRT);
writer->Write(1, outFrm, &numFramesWritten);
```

The following command codes are defined:

```
UeiMIL1553CmdBCRT,  // Remote terminal to receive data from bus controller
UeiMIL1553CmdRTBC,  // Remote terminal to transmit data to bus controller
UeiMIL1553CmdRTRT,  // One remote terminal to transmit data to another remote
                    // terminal
UeiMIL1553CmdModeTxNoData, // Tx Status word
UeiMIL1553CmdModeTxWithData, // Remote terminal to transmit data and/or
                             // status word to bus controller
UeiMIL1553CmdModeRxWithData, // Remote terminal to receive data for mode
                            // command from bus controller
UeiMIL1553CmdBCRTBroadcast, // Remote terminal to receive broadcast data
                            // from bus controller
UeiMIL1553CmdRTRTBroadcast, // One remote terminal to broadcast data to
                            // other remote terminals
UeiMIL1553CmdModeTxNoDataBroadcast, // Mode command without data, remote
                            // terminals should not reply
UeiMIL1553CmdModeRxWithDataBroadcast, // Mode command with data, remote
                            // terminals should receive data
```

There are two overloaded methods of ***SetCommand():***

```
SetCommand(int Rt_, int Sa_, int WordCount_, tUeiMIL1553CommandType
Command_)
and
SetCommand(int Rt_, int Sa_, int Rt2_, int Sa2_, int WordCount_,
tUeiMIL1553CommandType Command_)
```

The second SetCommand method is used to send RT-RT type commands.

Another useful method is ***SetDelay(int Delay_)*** which inserts a delay in microseconds before execution of the command. ***SetDelay()*** can be used to create a certain communication pattern on the bus without needing to use a bus controller.

SetTxBus(tUeiMIL1553PortActiveBus portBus) can be used to select which bus to use – A or B.

2.6.3 Programming and Working with a Bus Controller

The Bus Controller is by far the most complicated functional element on a MIL-1553-553 layer.

The DNx-1553-553 layer supports full implementation of the BC — all 1553 transfer types are supported, major and minor frame timing is fully programmable, and the BC also implements recovery of failed transactions and auto-status requests for the RTs that are in the “dead” state.

A Bus Controller uses the following memory model (note that terms “descriptor” and “entry” are used interchangeably and both refer to the single record in the major or minor frame tables):

1. One major frame is defined. A major frame may contain up to 256 entries for the minor frames.
2. A user can define up to sixteen different minor frames (i.e., minor frame types). Each type allows you to perform one of two sets of up to 128 transactions on the bus (only one block of 128 elements is active at a time, a second may be updated from the host, allowing double-buffering support)
3. Major and minor frame descriptors are stored in the on-board FPGA memory. They contain control bits that are set by the host and status bits that represent important status information about execution of the particular descriptor. They are set by the BC.
4. For each of the descriptors in every minor frame, there is a dedicated 128 16-bit word area in the external memory called a Bus Controller Control Block (BCCB) that contains flags that define the type of the transfer, retry behavior, and other parameters. The area also contains validation data and status information for the transactions performed plus RT->RT/BC data as well.
5. Up to 16 (minor frame types)*256(minor frame entries) = 4096 unique BCCBs may be defined per channel and used on a single DNx-1553-553 layer, occupying 1048576 16-bit memory words. These BCCBs can be addressed statically (when the Framework automatically assigns a BCCB for each minor entry using the following formula:
   <minor_frame_type*256>+<minor_frame_entry_number>
   or they may be assigned by the user.

For data coherency support, two double-buffering techniques are defined for major frame processing called a “SWAP” cycle and “minor frame double-buffering”. They complement each other:

“SWAP” Cycle in a Major Frame

- Any of the major frame descriptors may be marked with a “SWAP” flag (***`UeiMIL1553MjSwapEnabled`***) . Marking the entry does not affect its current operation.
- “SWAP” cycle is initiated by writing ***`CUeiMIL1552ControlFrame`*** with the proper command set.
- An actual “SWAP” cycle takes place at the beginning of the next major frame cycle (takes an extra 16uS). During the SWAP cycle, all entries in the major frame descriptor table that are marked for “SWAP” have the current value of the “Enable” bit inverted, i.e., currently enabled entries will be disabled and currently disabled entries enabled. Entries where a “SWAP” flag was cleared stay unaffected. This allows replacing of all minor frames at the same time.

The “SWAP” operation could be performed as follows:

```
CUeiMIL1553BCControlFrame* bccontrol = new CUeiMIL1553BCControlFrame;
bccontrol->SwapMjEntries();
writer->Write(1, bccontrol, &numFramesWritten);
```

Minor Frame Double-Buffering

Each minor frame is split into two blocks, the first uses descriptors 0-127 and second uses descriptors 128-255. Only one sub-frame is actively used by the BC and, as a result, only half of the BCCBs are accessed by the BC engine. This allows the host to change the other half without worrying about overriding the same data that is currently being processed by the BC.

A “Sub-frame” is selected by the value that is written to the port. 16 LSBs define which part of the minor frame is selected, a 0 in the corresponding position indicates that descriptors 0-127 should be used, a 1 indicates 128-255. Note that an actual change takes place for all minor frames at the end of the major frame cycle after all descriptors are processed in order to set the initial value to this register and initiate a SWAP cycle for the BC. The current and pending sub-frame for each minor frame are accessible in the same register.

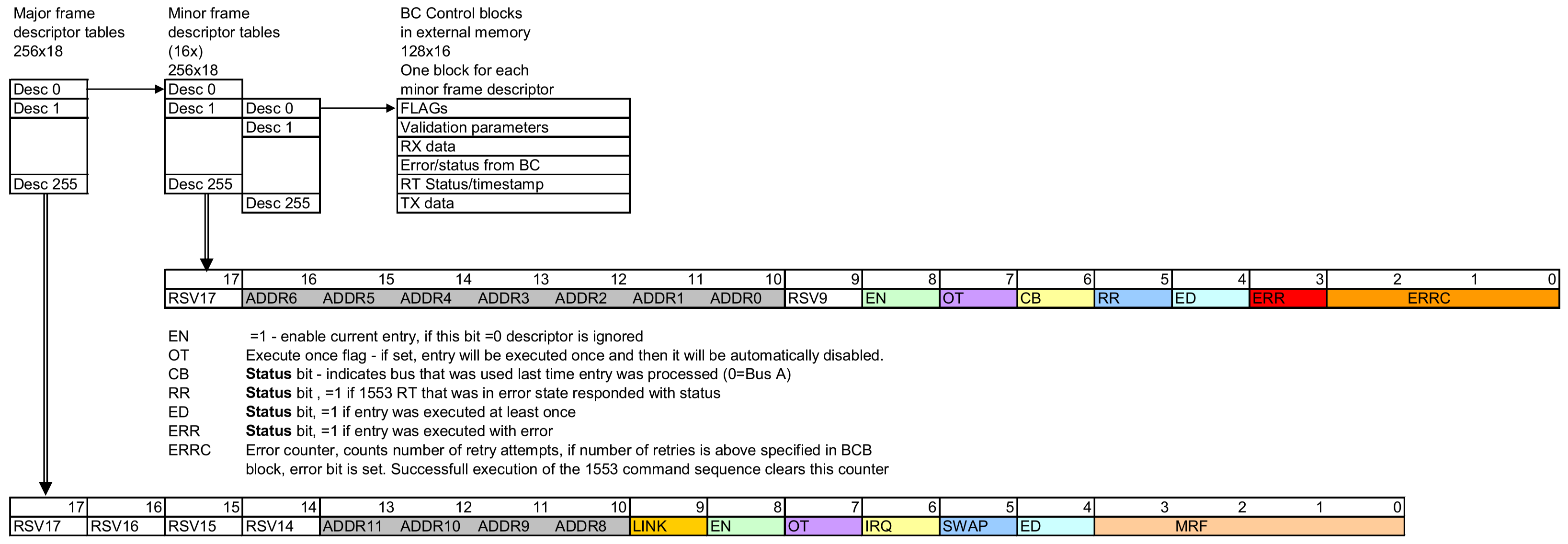
The following code demonstrates how to select block “1” (entries 128-255) for minor frames 2 and 3:

```
CUeiMIL1553BCControlFrame* bccontrol = new CUeiMIL1553BCControlFrame;
minor_blocks = (1L<<2)|(1L<<3);
bccontrol->SelectMnBlock(minor_blocks);
writer->Write(1, bccontrol, &numFramesWritten);
```

Double-buffering of the minor frames allows safe replacement of the data associated with each entry of the minor frame descriptor table.

A bus controller also provides a lot of information for a failed 1553 transaction that may be used for proper recovery – see a detailed description of the BCCB in the UeiDaq Framework Reference Manual.

Figure 2-1 is a graphical representation of the Bus Controller Memory Model. The notes listed below the figure contain explanatory information.


Major frame descriptor tables 256x18
Desc 0
Desc 1
Desc 255
Minor frame descriptor tables (16x) 256x18
Desc 0
Desc 1
Desc 255
Desc 0
Desc 1
Desc 255
BC Control blocks in external memory 128x16 One block for each minor frame descriptor
FLAGs
Validation parameters
RX data
Error/status from BC
RT Status/timestamp
TX data

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RSV17	ADDR6	ADDR5	ADDR4	ADDR3	ADDR2	ADDR1	ADDR0	RSV9	EN	OT	CB	RR	ED	ERR	ERRC		

EN	=1 - enable current entry, if this bit =0 descriptor is ignored
OT	Execute once flag - if set, entry will be executed once and then it will be automatically disabled.
CB	**Status** bit - indicates bus that was used last time entry was processed (0=Bus A)
RR	**Status** bit , =1 if 1553 RT that was in error state responded with status
ED	**Status** bit, =1 if entry was executed at least once
ERR	**Status** bit, =1 if entry was executed with error
ERRC	Error counter, counts number of retry attempts, if number of retries is above specified in BCB block, error bit is set. Successfull execution of the 1553 command sequence clears this counter

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RSV17	RSV16	RSV15	RSV14	ADDR11	ADDR10	ADDR9	ADDR8	LINK	EN	OT	IRQ	SWAP	ED	MRF			

EN	Enable descriptor, if this bit =0 descriptor is ignored and next entry is processed, until end of the descriptor table. Descriptor may be temporary disabled in order to replace data inside of the minor frame and/or minor frame descriptors
OT	Execute once flag - if set, entry will be executed once and then it will be automatically disabled. May be used for the minor frame filled with aperiodic messages
IRQ	Issue IRQ upon completion of the minor frame execution (SL553_PORT_BCI_UIRQ)
SWAP	Enable entry if disabled or disable if enabled upon SWAP request; SWAP request allows simultaneous change of the sequence in the major frame descriptor table at the beginning of the major frame cycle Write to SL553_PORT_BCSWP (0x21A4/0x28A4) initiates SWAP cycle upon next major frame clock
ED	**Status** bit, =1 if entry was executed at least once
MRF	Minor frame ID (0-15)
LINK	Link descriptor to the preceeding minor frame, do not wait for the clock

ADDR11-8 and ADDR6-0 - Index of the address of BCCB (bits 11-8 and 6:0 of 12-bit BCCB#), bit ADDRA7 is taken from SL553_PORT_BCFBS register

RSVx Reserved bits, must be written with 0 for the compatibility

Figure 2-1. Bus Controller Memory Model

Notes for Figure 2-1:

1. The minor frame CB status bit is also used by the BC state machine and should only be changed if for some reason, a BC should start transmitting on a different bus.
2. The RR bit is set by the BC if an RT that was in an “error” state replied with valid status. This bit along with ERR and ERRC bits should then be cleared to return the RT to normal operation.
3. An ERR bit, if set, must be cleared from the DNA side.
4. An ERRC bit is self-cleared if the number of retries is less than that allowed in the BCCB and the transaction on the 1553 bus completed without error. Otherwise, the DNA side should clear this counter.
5. A BC state machine operates without checking descriptor tables for coherency. It is recommended that you use the “SWAP” feature to update entries; i.e., keep two mirror copies of the major frame sequences and update the currently inactive one. This limits the number of unique minor frames by 8 or 15 (if only one minor frame is updated at a time).
6. ADDR bits allow each minor frame to access any of the available BCCB descriptors.

Bus Controller Programming Example

First, we need to allocate different frames to use with bus controller. We will need three types of frames for BC: BCCB Data, BCCD Status and BCCB Scheduler (one minor and one major):

```
CUeiMIL1553BCSchedFrame* major = new
CUeiMIL1553BCSchedFrame(UeiMIL1553BCFrameMajor);
CUeiMIL1553BCSchedFrame* minor = new
CUeiMIL1553BCSchedFrame(UeiMIL1553BCFrameMinor);
CUeiMIL1553BCCBDataFrame** fdata = new CUeiMIL1553BCCBDataFrame*[2];
CUeiMIL1553BCCBStatusFrame** fstatus = new CUeiMIL1553BCCBStatusFrame*[2];
```

The type of scheduler frame is assigned at the moment of creation by using a constructor with initialization. Let’s assume that we are going to have one minor frame with two entries in it and then construct BCCBs for both data and status:

```
fdata[0] = new CUeiMIL1553BCCBDataFrame(0, 0, 0);  // first and
fdata[1] = new CUeiMIL1553BCCBDataFrame(0, 1, 0);  // second minor frame
entries
fstatus[0] = new CUeiMIL1553BCCBStatusFrame(0, 0, 0); // ditto status
fstatus[1] = new CUeiMIL1553BCCBStatusFrame(0, 1, 0);
```

We will also need a BC control frame to start/stop and perform debug operations with the bus controller:

```
tUeiMIL1553BCControlFrame* bcControl = new tUeiMIL1553BCControlFrame;
```

The next step is to program a major frame entry:

```
major->AddMajorEntry(0, UeiMIL1553MjEnable);
writer->Write(1, major, &numFramesWritten);
```

In this case, we enable the first entry in the major frame. We assign minor frame zero to it and rely on the automatic allocation of BCCBs. In case a user wants to take BCCB allocation into its own hands, another member function can be called: ***`AddMajorEntry(minorId, entryMJ, BCCBSegment)`***, where ***`minorId`*** is a minor frame assigned, ***`entryMJ`*** are flags, but ***`BCCBSegment`*** is an upper 4 bits of a 12-bit BCCB index.

Now we need to program minor frame entries (two of them in our case):

```
minor->AddMinorEntry(UeiMIL1553MnEnable);
minor->AddMinorEntry(UeiMIL1553MnEnable);
writer->Write(1, minor, &numFramesWritten);
```

And BCCBs that correspond to that entry:

```
fdata[0]->SetCommand(UeiMIL1553CmdBCRT, Rt, Sa, WordCount);
fdata[0]->SetCommandBus(UeiMIL1553OpModeBusBoth);
fdata[0]->SetCommandDelay(100);
fdata[0]->SetRetryOptions(3, RetryType);
fdata[0]->CopyRxData(WordCount, data16);
writer->Write(1, fdata[0], &numFramesWritten);

fdata[1]->SetCommand(UeiMIL1553CmdRTBC, Rt, Sa, WordCount);
fdata[1]->SetCommandBus(UeiMIL1553OpModeBusBoth);
fdata[1]->SetCommandDelay(100);   // Delay in us
fdata[1]->SetRetryOptions(3, RetryType);
writer->Write(1, fdata[1], &numFramesWritten);
```

When you program a BCCB, you can select various retry options (see Reference Manual for further details)

```
tUeiMIL1553BCRetryType RetryType = (tUeiMIL1553BCRetryType)
(UeiMIL1553BCR_RNR|UeiMIL1553BCR_ESR|UeiMIL1553BCR_RE);
```

At this time, the BC controller is fully programmed and ready for operation. To switch the bus controller into operating mode, we need to use ***`UeiMIL1553BCControlFrame`***:

```
bcControl->SetEnableContinous(MajorClock, MinorClock);
```

The Major clock should be selected in such a way as to allow all minor frames to be performed within a single major clock.

```
MajorClock = 1.0;
MinorClock = 10.0;

writers[0]->Write(1, bcControl, &numFramesWritten);
```

You need to make sure that all commands programmed in the major and minor frames can be executed before the next major or minor clock. If a command is not completely executed at the time of the next clock, it is cancelled and either a major or the next minor frame is executed.

There are several ways a bus controller can be clocked. The normal way is to clock it continuously with major and minor clocks selected. Note that the minor frame clock is reset every time a major frame clock occurs. Thus, the first minor clock happens simultaneously with the major clock and the first major frame entry is executed.

For debugging, instead of running a bus controller continuously, a user can debug a bus controller in a step-by-step fashion by performing major and minor steps.

A major step is performed by writing a control frame which is set to ***bcControl->SetOneMajorStep().*** When this control frame is written, the bus controller makes one major frame step and executes the first major frame entry. To execute the next entry (i.e., execute the next minor frame) use ***SetOneMajorStep().*** You can always see the status of execution (i.e., bits in the minor frame) by reading ***CUeiMIL1553BCSchedFrame*** with properly set minor frame number, index, block, and the number of entries added to match the number of requested entries.

You can request information about what major and minor entries are currently executed by reading ***CUeiMIL1553BCStatusFrame.***

Once the bus controller is up and running, you can read the BCCB status and write BCCB data into it at any time (normally done in the cycle):

```
// Change and retrieve BC data each iteration
// Store data for BC "Receive" command
for (i = 0; i < WordCount; i++) data16[i] = (uInt16)(0x1000 + c + i);
fdata[0]->CopyRxData(WordCount, data16);
writer->Write(1, fdata[0], &numFramesWritten);

// Read data stored in BC "Transmit" command
reader->Read(1, fstatus[1], &numFramesRead);
std::cout << "BC= " << fstatus[1]->GetBcDataStr(WordCount) << std::endl;
```

This particular example updates data in a BCCB data frame in a cycle and reads and prints out the contents of the BCCB status frame.

There are multiple helper methods associated with both BCCB data and status classes. Please see UeiDaq Reference Manual for further details.

To stop or pause a bus controller operation, write a BC control frame set to: ***bcControl->SetDisable().***

Some fields in the BCCB status are bitfields. For example ***<errsts0>*** and ***<errsts1>*** are bitfields reporting on the error status, if any, that occurred while processing a BCCB. Look up ***tUeiMIL1553_BCB_ERRSTS0*** and ***tUeiMIL1553_BCB_ERRSTS1*** in the Reference Manual for definitions of the error reporting statuses.

A Bus Controller is equipped with a set of features that allows it to simplify communication with RTs.

For example, for each BCCB (i.e., for each command) you can set up the following options:

- ***fdata[0]->SetCommandDelay(100);*** (where 100 = delay in uS). This option allows you to set up a delay to be performed before executing a particular command. The maximum delay length (calculated in microseconds) is set to 20000us (20ms). This feature is used to create a peculiar pattern of BC commands on the bus.
- ***fdata[0]->SetRetryOptions(3, RetryType);*** (where “3” is the number of retries.) This option allows you to set up the number and type of retries if an RT fails to answer the BC command. Zero retries does not cause the bus controller to repeat a failed command at all, while seven retries will cause bus controller to retry that command forever.

The following retry types are defined:

```
typedef enum _tUeiMIL1553BCRetryType
{
UeiMIL1553BCR_IRT   =(1L<<14), // Retry on incorrect RT# in status
UeiMIL1553BCR_RUS   =(1L<<13), // Retry on unexpected status reception
UeiMIL1553BCR_RUD   =(1L<<12), // Retry on unexpected data reception
UeiMIL1553BCR_RWB   =(1L<<11), // Retry on wrong bus response
UeiMIL1553BCR_RIS   =(1L<<10), // Retry on illegal bits set in status
UeiMIL1553BCR_RBB   =(1L<<9),  // Retry on busy bit in status
UeiMIL1553BCR_RTE   =(1L<<8),  // Retry on bus timing error (late
                                      // reply)
UeiMIL1553BCR_RWC   =(1L<<7),  // Retry on word count (number of data
                                      // words received)
UeiMIL1553BCR_RE    =(1L<<6),  // Re-enable command transmission on
                                      // successful status reply
UeiMIL1553BCR_RNR   =(1L<<5),  // Retry on no-response
UeiMIL1553BCR_ERE   =(1L<<4),  // Enable re-transmit on alternative
                                      // bus each retry
UeiMIL1553BCR_ESR   =(1L<<3)   // Enable periodic status request
                                      // command when retry
} tUeiMIL1553BCRetryType;
```

While most retry types are self-explanatory, some of them require explanation:

- If the ***UeiMIL1553BCR_ESR*** bit is set, the bus controller will continue to retry this command with the status request command even if all retry attempts have expired. If at one moment the destination RT replies with the status, you can receive this notification in a BC status frame. If, however, flag ***UeiMIL1553BCR_RE*** is also set, the bus controller will switch back to the normal mode, i.e., sending a command to the RT that replied with the normal status after the hiccup.
- If the ***UeiMIL1553BCR_ERE*** flag is set, the BC will try the alternate bus after encountering one or another bus error accordingly with retry types selected.

Data Validation

A MIL-1553-553 bus controller is able to perform status and data validation and accept/reject data based on the data or status received.

CUeiMIL1553BCCBDataFrame::EnableDataCompare (int enable) enables or disables the comparing of data received (i.e., Tx command) with the data programmed in the BCCB data table. In this case, you can program the minimum and maximum (straight binary) acceptable values. The reply data received from the RT will be compared word by word to the minimum and maximum values programmed and will be rejected with an error bit set if it fails.

Use ***CUeiMIL1553BCCBDataFrame::CopyTminData(int Size, unsigned short* data)*** and ***CUeiMIL1553BCCBDataFrame::CopyTmaxData(int Size, unsigned short**** data) to fill the frame with the proper data words.

Bit ***UeiMil1553_BCB_ERRSTS0_DCF*** in ***tUeiMIL1553_BCB_ERRSTS0*** (belonging to the relevant BCCB) reports about data status compare failure.

Status Validation

A MIL-1553-553 bus controller is able to perform operations on data received from an RT status word (or both status words for an RT-RT command) and accept or reject a reply from an RT, depending on the received status.

To enable this option, you need to call ***CUeiMIL1553BCCBDataFrame::EnableStatusCompare(int number, unsigned short and_sts, unsigned short or_sts, unsigned short value, int enable),*** where "number" is either 0 for main status or 1 for the second RT-RT command status. Regardless of which status this is, the following logic operations will be performed on them:

```
((<received status> & <and_sts>)|<or_sts>) == value
```

If the result of this operation is TRUE, the status passed validation.

Using ***<and_sts>***, you can mask out irrelevant status bits and using ***<or_sts>*** you can set up bits in the received command to be able to pass validation for, say, different replies.

See bits ***UeiMil1553_BCB_ERRSTS0_S1F*** and ***UeiMil1553_BCB_ERRSTS0_S2F*** in ***tUeiMIL1553_BCB_ERRSTS0*** for information about a status compare failure.

Multi-Rate Programming

The MIL-1553 standard calls for the ability to run different frames at different frame rates.

The MIL-1553-553 layer provides two ways to support multiple rate programming. The first way is known as a "Link Bit" which allows execution of two or more major frame entries upon a single minor frame clock.

Let's assume that we have created three types of frames: one set of RTs needs to be updated at 8Hz, another set of RTs needs to be updated at 4Hz, and the slowest RT needs to be updated at 1Hz.

Let's consider the following major frame (major clock set to 1Hz and minor clock set to 8Hz) as shown in Table 2-2 on page 32:

Table 2-2. Multi-Rate Programming

Major Entry Index	Minor Frame Type	Flags	Effective Update Rate	Upon Clock Tick
0	0	Enable	8Hz	Major
1	1	Enable+Link	4Hz	
2	2	Enable+Link	1Hz	
3	0	Enable	8Hz	Minor
4	0	Enable	8Hz	Minor
5	1	Enable+Link	4Hz	
6	0	Enable	8Hz	Minor
7	0	Enable	8Hz	Minor
8	1	Enable+Link	4Hz	
9	0	Enable	8Hz	Minor
10	0	Enable	8Hz	Minor
11	1	Enable+Link	4Hz	
12	0	Enable	8Hz	Minor

Major frame entries 0, 1 and 2 are executed upon a major clock tick; entries 3 and 4, 5 and 6, etc. are executed upon a minor clock tick. There is no delay in execution between pairs of entries larger than those required to execute the corresponding minor frame.

The second way of doing the same thing is to create three minor frame types: Type 2 contains data for update rates 8Hz, 4Hz, and 1Hz, Type 1 contains entries for rates 8Hz and 4Hz, and Type 0 contains minor entries for 8Hz only.

For example, for –

Minor Frame Type 2:

Entry	Data for	BCCB Index	Effective BCCB
0	RT1 (8Hz)	Segment 0 + Index 0	0
1	RT2 (4Hz)	Segment 0 + Index 1	1
2	RT3 (1Hz)	Segment 0 + Index 2	2

Minor Frame Type 1:

Entry	Data for	BCCB Index	Effective BCCB
0	RT1 (8Hz)	Segment 0 + Index 0	0
1	RT2 (4Hz)	Segment 0 + Index 1	1

Minor Frame Type 0:

Entry	Data for	BCCB Index	Effective BCCB
0	RT1 (8Hz)	Segment 0 + Index 0	0

Now, we can fill major frame as follows:

Major Entry Index	Minor Frame Type	Flags	Effective Update	Upon Clock Tick
0	2	Enable	RTs 1,2,3	Major
1	0	Enable	RTs 1	Minor
2	1	Enable	RTs 1,2	Minor
3	0		RTs 1	Minor
4	1		RTs 1,2	Minor
5	0		RTs 1	Minor
6	1		RTs 1,2	Minor
7	0		RTs 1	Minor

As you can see, the content of the major frame entries is different. Due to the BCCB assignments, the data for each RT is stored/retrieved exactly into/from the same location.

2.6.4 Programming and Working with Remote Terminals

A MIL-1553-553 board can support up to 32 remote terminals with 32 subaddresses each. Not all remote terminal or subaddress can be used. RT31 is reserved as a ”broadcast” terminal which sends and receives broadcast commands (i.e., in the MIL-1553 standard, all terminals should receive a command if the source of the command is RT31). In our implementation, a broadcast command is always received by RT31 due to the fact that all RTs are under programmer control. SA0 and SA31 are also reserved for use with the special types of commands – mode commands, which are designed to provide control operations rather than simply to exchange data.

Programming an RT starts from creating a port:

```
CUeiMIL1553Port* pPort0 = session.CreateMIL1553Port(
                                    "pdna://192.168.100.2/dev0/milb0",
                                    UeiMIL1553CouplingTransformer,
                                    UeiMIL1553OpModeRemoteTerminal
                                    );
```

Next, create a reader and a writer:

```
CUeiMIL1553Reader* reader = new CUeiMIL1553Reader(session.GetDataStream(),
session.GetChannel(0)->GetIndex());
CUeiMIL1553Writer* writer = new CUeiMIL1553Writer(session.GetDataStream(),
session.GetChannel(0)->GetIndex());
```

By calling constructors with initialization, reader and writer objects are now connected to the proper data stream and MIL-1553-553 channel.

Now, you need to create frames that are going to be used to read and write RT data:

Input data frame (i.e., data from an Rx command):

```
CUeiMIL1553RTFrame* inFrm;
inFrm = new CUeiMIL1553RTFrame(Rt, Sa, Block, messageSize);
```

Output data frame (i.e., data for a Tx command):

```
CUeiMIL1553RTFrame* outFrm;
outFrm = new CUeiMIL1553RTFrame(Rt, Sa, Block, messageSize);
```

In this particular example, Rt and Sa are an RT and SA of interest, Block is the part of the RT data which is data that is going to be written to and ***messageSize*** should be equal to ***Word Count*** of the expected bus controller command for this RT.

The next (optional) step is to create status request and control frames as well as a filter entry.

```
CUeiMIL1553RTStatusFrame* rqFrm = new CUeiMIL1553RTStatusFrame(Rt);
where Rt is a
```

A status frame allows you to read ***(reader->Read(numFrames, statusFrame, &numFramesRead);)*** status of the particular RT. Let's look at the structure of the status frame (***tUeiMIL1553RTStatusFrame***).

A few members contain important information about the current status of RT:

<DataReady> has bits set for SAs that received data.

<DataSent> has bits set for SAs that transmit data (of course, upon appropriate BC command).

<ChStatus> represents the current status of the port bus. To decode this bitfield, a user needs to use ***tUeiMIL1553_CH_STS*** structure.

<Status0> is a combination of last command (upper 16 bits) and status (lower 16 bits) word of the last command. Note that this information is on per-RT basis and if another SA of the same RT received or sent a command, the previous command status information will be overwritten.

<Status1> is the last SYNC (upper 16-bit) and transmitter shutdown (lower 16-bit) data words if those commands were received.

In a default mode, all 32 RTs and all 32 SAs are functional once you start operations. Often, this mode is not a desirable mode of operation because only a certain subset of RT/SA needs to be in operation, especially when the application involves simulation of a part of an aircraft network.

To limit the scope of simulated RTs, you need to set up filtering:

```
CUeiMIL1553FilterEntry* filterFrm = new CUeiMIL1553FilterEntry;
```

Filter entries are accumulated in the Framework when the procedure of setting RT filtering looks as follows:

```
pPort0->ClearFilterEntries();

for(up to 360 entries) {
    filterFrm->Set(UeiMIL1553FilterByRt, Rt, 0);
    filterFrm->EnableCommands(TRUE, TRUE, TRUE);
    pPort0->AddFilterEntry(*filterFrm);
}
pPort0->EnableFilter(TRUE);
```

Filtering can be accomplished by the following criteria:

```
UeiMIL1553FilterByRt - Filter only by RT numbers, all SAs of any length are
                       enabled
UeiMIL1553FilterByRtSa - Each RT/SA combination should be declared
UeiMIL1553FilterByRtSaSize - Each RT/SA combination and size of Rx/Tx Min/
                       Max data should be declared
```

You cannot mix and match criteria. The Framework takes first entry filtering criteria and applies it to all consecutive entries programmed.

Next, you need to fill the Tx data area of the RT with data.

```
outFrm->CopyData(WordCount, data16);
writer->Write(1, outFrm, &numFramesWritten);
```

Once data is written to an RT Tx data area, the RT becomes active (if it was explicitly selected in the filter or filter wasn't set at all.)

To read data from an RT Rx data area, you read:

```
reader->Read(1, inFrm, &numFramesRead);
std::cout << "RT= " << inFrm->GetFrameStr() << " Data: " << inFrm->GetDataStr() <<
std::endl;
```

Please refer to the description of ***CUeiMIL1553RTFrame*** for more detailed information and helpful member functions.

Controlling an RT controller

`CUeiMIL1553RTControlFrame` allows multiple control options. You need to use writer to write this frame to change RT behavior on the fly. The best bet is to use helper functions that will help to fill the frame with proper values.

First, you need to create a control frame:

```
CUeiMIL1553RTControlFrame* ctrlFrame = new CUeiMIL1553RTControlFrame;
```

This frame could be used for various purposes, such as:

1. Enable/disable RTs:

```
crtlFrame->SetRt(Rt); // select Rt to enable/disable
ctrlFrame->SetEnable(TRUE); // or FALSE to disable this RT
writer->Write(numWrite, ctrlFrame, &numWritten);
```

The Framework will combine enable/disable flags from each frame in the **`Write()`** operation and write it to the RT controller in one block

2. Enable/disable RTs by mask.

```
crtlFrame->SetRt(Rt); // select Rt to enable/disable
ctrlFrame->SetEnableMask(Rt_mask); // set bit to 1 for each Rt to
                 // enable
writer->Write(1, ctrlFrame, &numWritten);
```

You need to write only one frame to enable/disable each RT because all RTs are represented as a bitmask.

3. Set block for each RT

```
ctrlFrame->SelectBlock(Rt, Block); // Block is 0 or 1
writer->Write(numWrite, ctrlFrame, &numWritten);
```

The Framework will combine enable/disable flags from each frame in the **`Write()`** operation and write it to the RT controller in one block.

4. Set validation entry directly:

```
ctrlFrame->SetValidEntry(Rt, Sa, validationEntry);
writer->Write(numWrite, ctrlFrame, &numWritten);
```

Validation entries are defined in **powerdna.h** as follows:

```
#define SL553_MEMVAL_LB_EN        (1L<<30) // =1 to loopback this Rx to Tx
#define SL553_MEMVAL_ERRI         (1L<<29) // =1 inject error into the
                 // traffic for this RT/SA. Code is in bits 0..9
#define SL553_MEMVAL_MD_TX_DW     (1L<<28) // =1 when mode data word is
                 // expected and T/R_N bit set to 1
#define SL553_MEMVAL_MD_RX_DW     (1L<<27) // =1 when mode data word is
                 // expected and T/R_N bit set to 0
#define SL553_MEMVAL_MD_IRQ_EN    (1L<<26) // Issue IRQ when corresponding
                 // mode command received
#define SL553_MEMVAL_MD_TX_EN     (1L<<25) // Enable corresponding mode
                 // command with T/R_N bit set to 1
#define SL553_MEMVAL_MD_RX_EN     (1L<<24) // Enable corresponding mode
                 // command with T/R_N bit set to 0
#define SL553_MEMVAL_TX_IRQ_EN    (1L<<23) // Enable TX IRQ on the selected
                                   // subaddress
```

```
#define SL553_MEMVAL_RX_IRQ_EN    (1L<<22) // Enable RX IRQ on the selected
                                     // subaddress
#define SL553_MEMVAL_TX_EN        (1L<<21) // Enable TX on the selected
                                     // subaddress
#define SL553_MEMVAL_RX_EN        (1L<<20) // Enable RX on the selected
                                     // subaddress
#define SL553_MEMVAL_TX_WCMAX_MSB (1L<<19) // Maximum word count for the TX
#define SL553_MEMVAL_TX_WCMAX_LSB (1L<<15) // 0=32; 1-31=1-31
#define SL553_MEMVAL_TX_WCMIN_MSB (1L<<14) // Minimum word count for the TX
#define SL553_MEMVAL_TX_WCMIN_LSB (1L<<10) // 0=32; 1-31=1-31
#define SL553_MEMVAL_RX_WCMAX_MSB (1L<<9)  // Maximum word count for the RX
#define SL553_MEMVAL_RX_WCMAX_LSB (1L<<5)  // 0=32; 1-31=1-31
#define SL553_MEMVAL_RX_WCMIN_MSB (1L<<4)  // Minimum word count for the RX
#define SL553_MEMVAL_RX_WCMIN_LSB (1L<<0)  // 0=32; 1-31=1-31
```

Setting validation entries directly is an advanced operation. It is described in more detail in the PowerDNA API Reference Manual.

2.7 Stopping the Session

You can stop the session by calling the session object's method ***"Stop"***:

```
// Stop the session
mySession.Stop();
```

Note that if you don't explicitly stop the session, it will be automatically stopped when the session object is destroyed or when it goes out of scope.

2.8 Destroying the Session

In C++, if you created the session object on the stack, it will automatically free its resources when it goes out of scope. As an alternative, you can force it to free its resources by calling the method ***"CleanUp"***, as shown below:

```
// Clean-up session
mySession.CleanUp();
```

If you dynamically created the session object, you need to destroy it to free all resources:

```
// Destroy session
delete(pMySession);
```

In C, you need to call ***"UeiDaqCloseSession"*** to free all resources:

```
UeiDaqCloseSession(mySession);
```

With .NET managed languages, the garbage collector will take care of freeing resources once the session object is not referenced anymore. You can also force the session to release its resources by calling the ***"Dispose"*** method.

Chapter 3 Programming with the Low-Level API

The DqAdv functions of the low-level API, which are included in this chapter, offer direct access to PowerDNA DaqBIOS protocol and allow you to access device registers directly. For additional information, please refer to the `API Reference Manual` document under:

Start » Programs » UEI » PowerDNA » Documentation

for all pre-defined types, error codes, and functions you can use with this layer.

NOTE: High-level UEI Framework support for this layer is not available in the current release of this product.

3.1 Low-Level DqAdv Functions

The DNx-1553-553 Interface Module (Layer) is designed to support MIL-STD-1553A and MIL-STD-1553B interfaces. Two dual redundant independent channels are available. The layer can support up to 32 remote terminals (RTs), one Bus Monitor (BM), and one Bus Controller (BC).

The layer has two independent channels; each channel incorporates all that is needed to communicate with a dual redundant 1553 bus. Bus coupling (transformer/direct) is software-selectable.

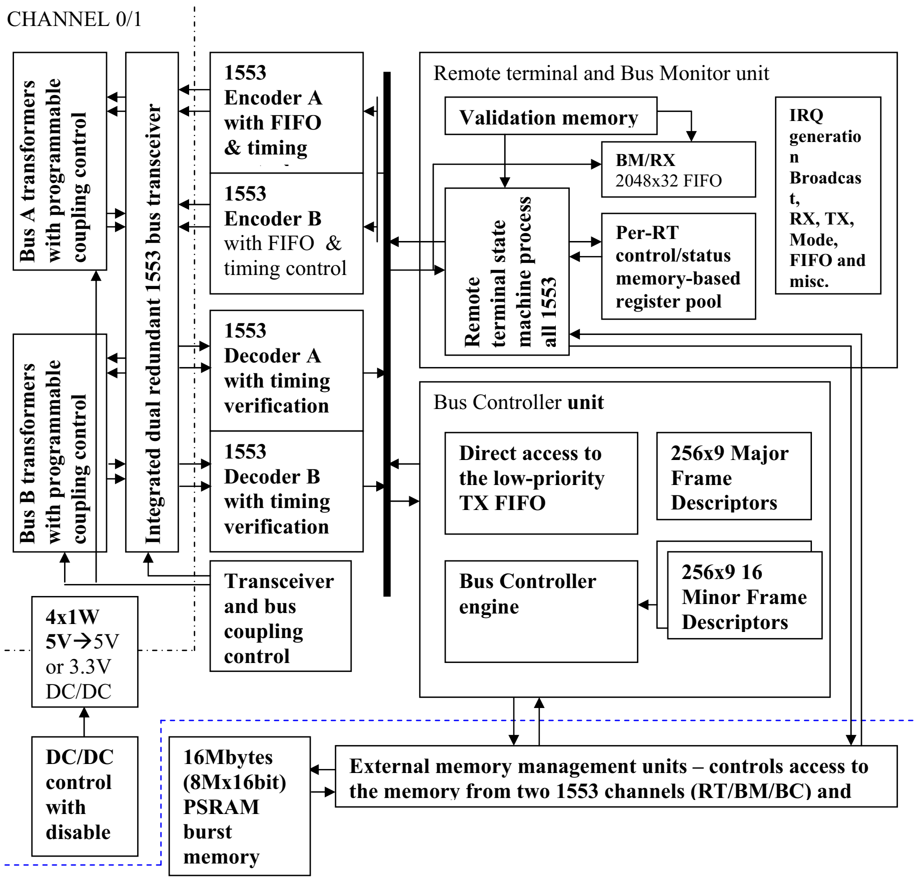

CHANNEL 0/1
Bus A transformers with programmable coupling control
Bus B transformers with programmable coupling control
Integrated dual redundant 1553 bus transceiver
1553 Encoder A with FIFO & timing
1553 Encoder B with FIFO & timing control
1553 Decoder A with timing verification
1553 Decoder B with timing verification
Transceiver and bus coupling control
4x1W 5V→5V or 3.3V DC/DC
DC/DC control with disable
16Mbytes (8Mx16bit) PSRAM burst memory
External memory management units – controls access to the memory from two 1553 channels (RT/BM/BC) and
Remote terminal and Bus Monitor unit
Validation memory
BM/RX 2048x32 FIFO
Remote terminal state machine process all 1553
Per-RT control/status memory-based register pool
IRQ generation Broadcast, RX, TX, Mode, FIFO and misc.
Bus Controller unit
Direct access to the low-priority TX FIFO
256x9 Major Frame Descriptors
Bus Controller engine
256x9 16 Minor Frame Descriptors

Figure 3-1. DNx-1553-553 Logic Block Diagram

As shown in **Figure 3-1**, each channel has dual 1553 decoders that are capable of decoding independent streams of 1553 Manchester words and passing them to upper-level subsystems. Decoders can detect various timing errors on the bus and also keep track of the gap interval between messages as well as data parity errors and type of received data words (command-status or data).

When data is stored into the 1024 32-bit word FIFO (BM mode), it is stored from both A and B buses and each control/status word may be optionally time-stamped. In RT mode, the 1553 protocol is parsed and processed by the RT state machine, which inherently supports up to 32 RTs simulated by the single channel. Each RT may be individually enabled and each sub-address may be also individually configured for RX/TX or both, with maximum and minimum numbers of allowable data words per subsystem. Mode commands may be enabled and disabled as well. Also, each sub-address and each mode command have a flag that controls interrupt generation upon receiving a corresponding RX /TX or mode command. Switching between A and B redundant buses takes place automatically upon receiving a command on the bus, and the host may receive an interrupt once it happens.

The Manchester encoder allows data output on A and/or B buses. However, it always accepts data from the same source; i.e., it is impossible to send different data to A and B buses at the same time. The encoder has a two-256x32 word FIFO that accepts 1553 data in a format that includes bus inactivity gap time delay, word type, and parity information. These FIFOs are called high- and low-priority FIFOs and are used for both RT and BC modes. Currently, a BC only has access to the low priority FIFO. Data is outputted from the FIFOs as a complete message and a low-priority FIFO waits until all messages from the high-priority FIFOs are gone.

A dedicated memory controller interfaces with 16Mbytes of fast burst PSRAM. It keeps up with data requests from both channels for the TX data and accepts RX messages and status information as well. Once a message is received or transmitted for the particular subsystem, special flags are set and once new data is placed in the TX buffer or data is read from the RX buffer, those flags are cleared. DNA may write or read data in blocks as large as 1024 16-bit data words at a time. A read or write is executed as an atomic transaction, i.e., no data change allowed during a read or write to/from the DNA bus.

This document describes the initial function set for BM and RT modes of operation.

For detailed descriptions of the low-level functions you can use with DNx-1553-553 boards, refer to the PowerDNA API Reference Manual, which is available for download at **www.ueidaq.com**

Appendix

A. Accessories

DNA-CBL-COM
1-ft long, round shielded cable with 37-pin male and four 9-pin male D-sub connectors

Index