# Software User's Manual

## BusTools/1553-API

Publication No. 1500-045 Rev. 5.12

Supporting Products:

- QPCX-1553
- QPMC-1553
- R15-EC
- RPCC-D1553
- QVME-1553
- RQVME2-1553
- R15-MPCIE
- QCP-1553
- R15-AMC
- Q104-1553P
- QPCI-1553
- RPCIE-1553
- QPM-1553
- R15-USB-MON
- RXMC-1553
- RXMC2-1553
- R15-LPCIE
- R15-USB
- RAR15-XMC-IT/RAR15XF
- R15-PMC

abaco
SYSTEMS

## Document History

| Revision | Date | Description |
|---|---|---|
| 5.11 | July 2019 | BusTools/1553-API Software Revision: 8.24<br>Hardware Revision: 6.11/6.09/6.08/6.03/5.18/4.6x/4.4x<br>Rebranding to the Abaco Systems format. |
| 5.12 | Dec 2019 | BusTools/1553-API Software Revision: 8.28<br>Updated supported versions of Windows and added Hardware Revision 6.17 |
| | | |

## Waste Electrical and Electronic Equipment (WEEE) Returns

Abaco Systems is registered with an approved Producer Compliance Scheme (PCS) and, subject to suitable contractual arrangements being in place, will ensure WEEE is processed in accordance with the requirements of the WEEE Directive.

Abaco Systems will evaluate requests to take back products purchased by our customers before August 13, 2005 on a case-by-case basis. A WEEE management fee may apply.

# About This Manual

## Conventions

### Notices

This manual may use the following types of notice:

WARNING
Warnings alert you to the risk of severe personal injury.

CAUTION
Cautions alert you to system danger or loss of data.

NOTE
Notes call attention to important features or instructions.

TIP
Tips give guidance on procedures that may be tackled in a number of ways.

LINK
Links take you to other documents or websites.

### Terms

**Windows** – References to “Windows” refer to all supported Microsoft Windows® Operating Systems.

**BusTools/1553 Hardware** – Refers to Abaco Systems MIL-STD-1553 boards supported by BusTools/1553-API.

**BusTools/1553 Software** – Refers to the Graphical User Interface (GUI) program that uses the BusTools/1553-API. Using this GUI, you can program a Bus Controller and Remote Terminals, as well as record 1553 data to disk for later analysis.

**Channel** – Refers to a MIL-STD-1553 interface with dual redundant buses. “Dual Redundant” consists of two 1553 buses, primary and secondary, with one bus active at a time. The primary bus is bus A, and the secondary bus is bus B. Many Abaco Systems products have multiple 1553 interface channels. This document refers to those as Channel 1, Channel 2, Channel 3, and Channel 4.

## Numbers

All numbers are expressed in decimal, except addresses and memory or register data, which are expressed in hexadecimal. Where confusion may occur, decimal numbers have a “D” subscript and binary numbers have a “b” subscript. The prefix “0x” shows a hexadecimal number, following the ‘C’ programming language convention. Thus:

One dozen = $12_D$ = 0x0C = $1100_b$

The multipliers “k”, “M” and “G” have their conventional scientific and engineering meanings of $x10^3$, $x10^6$ and $x10^9$, respectively, and can be used to define a transfer rate. The only exception to this is in the description of the size of memory areas, when “K”, “M” and “G” mean $x2^{10}$, $x2^{20}$ and $x2^{30}$ respectively.

In PowerPC terminology, multiple bit fields are numbered from 0 to n where 0 is the MSB and n is the LSB. PCI terminology follows the more familiar convention that bit 0 is the LSB and n is the MSB.

## Text

Signal names ending with “#” denote active low signals; all other signals are active high. “-” and “+” denote the low and high components of a differential signal respectively.

## Further Information

### Abaco Website

You can find information regarding Abaco products on the following website:

LINK
https://www.abaco.com

### Abaco Documents

This document is distributed via the Abaco website. You may register for access to manuals via the website.

LINK
https://www.abaco.com/products/avionics

- BusTools/1553-API Reference Manual

LINK
https://www.abaco.com/download/bustools1553-api-reference-manual

- UCA32 LPU Reference Manual

LINK
https://www.abaco.com/download/uca32-lpu-reference-manual

- UCA32 Global Register Reference Manual

LINK
https://www.abaco.com/download/uca32-global-reg-ref-manual

- MIL-STD-1553 UCA Reference Manual

LINK
https://www.abaco.com/download/mil-std-1553-uca-reference-manual

## Technical Support Contact Information

You can find technical assistance contact details on the website Embedded Support page.

LINK
https://www.abaco.com/embedded-support

Abaco will log your query in the Technical Support database and allocate it a unique Case number for use in any future correspondence.

Alternatively, you may also contact Abaco's Technical Support via:

LINK
avionics.support@abaco.com

## Returns

If you need to return a product, there is a Return Materials Authorization (RMA) form available via the website Support page.

LINK
https://www.abaco.com/support

Do not return products without first contacting the Abaco Repairs facility.

## Safety Summary

The following general safety precautions must be observed during all phases of the operation, service and repair of this product. Failure to comply with these precautions or with specific warnings elsewhere in this manual violates safety standards of design, manufacture and intended use of this product.

Abaco assumes no liability for the customer’s failure to comply with these requirements.

### Ground the System

To minimize shock hazard, the chassis and system cabinet must be connected to an electrical ground. A three-conductor AC power cable should be used. The power cable must either be plugged into an approved three-contact electrical outlet or used with a three-contact to two-contact adapter with the grounding wire (green) firmly connected to an electrical ground (safety ground) at the power outlet.

### Do Not Operate in an Explosive Atmosphere

Do not operate the system in the presence of flammable gases or fumes. Operation of any electrical system in such an environment constitutes a definite safety hazard.

### Keep Away from Live Circuits

Operating personnel must not remove product covers. Component replacement and internal adjustments must be made by qualified maintenance personnel. Do not replace components with power cable connected. Under certain conditions, dangerous voltages may exist even with the power cable removed. To avoid injuries, always disconnect power and discharge circuits before touching them.

### Do Not Service or Adjust Alone

Do not attempt internal service or adjustment unless another person capable of rendering first aid and resuscitation is present.

### Do Not Substitute Parts or Modify System

Because of the danger of introducing additional hazards, do not install substitute parts or perform any unauthorized modification to the product. Return the product to Abaco for service and repair to ensure that safety features are maintained.

# Contents

About This Manual .... 3
1 • Unpacking and Handling .... 13
1.1 Unpacking Procedures .... 13
1.2 Handling Precautions .... 13
2 • Overview .... 14
2.1 Introduction .... 14
2.2 BusTools/1553-API .... 14
2.2.1 Interface to Other Languages .... 14
2.2.2 API Source Code .... 15
2.3 Supported Hardware .... 15
2.4 Hardware Features .... 16
3 • Operational Modes and Timing .... 17
3.1 Single-function, Dual-function, and Multi-function Modes .... 17
3.2 RT Validation Mode (Single RT) .... 17
3.3 MIL-STD-1553A and MIL-STD-1553B Operation .... 18
3.3.1 MIL-STD-1553A Operation .... 18
3.3.2 MIL-STD-1553B Operation .... 18
3.4 Setting and Using Time-tags .... 18
3.4.1 Time-tag Modes .... 19
3.4.2 Time-tag Initialization .... 20
3.4.3 Formatting and Displaying Time-tags .... 20
3.4.4 Selecting Time-tag Options .... 20
3.4.5 Controlling and Synchronizing Time-tags .... 21
3.4.6 Reading Time-tags .... 22
3.5 Using IRIG-B on Selected Boards .... 22
4 • Bus Monitor .... 24
4.1 BM Hardware Operation .... 24
4.2 BM Software Operation .... 26
4.3 BM Recording .... 28
5 • Remote Terminals .... 29
5.1 RT Hardware Operation .... 29
5.2 RT Software Operation .... 31
5.3 RT Monitor Mode .... 33
5.4 RT Extended Status Mode .... 33
5.5 Dynamic Bus Control .... 34
6 • Bus Controller .... 36
6.1 BC Hardware Operation .... 36
6.1.1 Frame Messaging .... 36
6.1.2 Message Scheduling .... 37
6.2 BC Message Block Types .... 37
6.2.1 1553 BC Message Block .... 37
6.2.2 Conditional Branch BC Message Block .... 38
6.2.3 Stop BC Message Block .... 39
6.2.4 NO-OP BC Message Block .... 39
6.2.5 Timed NO-OP BC Message Block .... 39
6.2.6 Timed NOOP in Different BC Timing Modes .... 39
6.3 Aperiodic 1553 BC Messages .... 40
6.3.1 High Priority Aperiodic Messages .... 40
6.3.2 Low-Priority Aperiodic Messages .... 41

6.3.3 Aperiodic Message Timing ....41
6.4 Dynamic Bus Control ....42
6.5 Message Scheduling ....42
6.6 Minor Frame Definition ....42
6.7 Message Gap Timing ....44
6.8 BC Software Operation ....45
6.8.1 Initializing the Bus Controller ....45
6.8.2 Allocating BC Messages ....47
6.8.3 BC Messages ....47
6.8.4 Defining a Minor Frame ....48
6.8.5 Setup Message Gap Timing ....48
6.8.6 Bus Controller Interrupt Programming ....48
6.9 BC One-Shot Operation ....48
6.10 BC Read/Write/Update/Allocate APIs ....49
7 • Error Injection .... 52
7.1 Error Injection Hardware Operation ....52
7.2 Error Types ....53
7.3 Enhanced Zero-Crossing ....54
7.4 Error Injection Software Operation ....55
8 • Interrupt Queue and Interrupts .... 56
8.1 Interrupt Queue Initialization and Structure ....56
8.2 Selecting Interrupt Events ....57
8.2.1 Selecting Bus Controller Interrupts ....58
8.2.2 Selecting Bus Monitor Interrupts ....59
8.2.3 Selecting Remote Terminal Interrupts ....59
8.2.4 Selecting External Trigger Interrupts ....59
8.3 Interrupt Processing ....59
8.3.1 Interrupt Queue Software Operation ....60
8.3.2 Polling ....61
8.3.3 Interrupts ....61
8.3.4 Setting Up Interrupts with BusTools_RegisterFunction ....63
8.4 Polled or Interrupt Driven? ....64
9 • Board Memory Organization .... 65
9.1 Hardware Operation ....65
9.2 Software Operation ....65
9.2.1 Memory Segmentation ....65
9.3 Memory Organization ....66
10 • C# Support .... 68
10.1 Introduction ....68
10.2 The Reference Solution ....69
10.3 The API, Data, and Constants Classes ....69
10.4 Building and Running the Application ....70
10.5 Adding the Managed Wrapper to an Existing .NET Application ....70
10.6 Important Coding Differences when Using the .NET Wrappers ....70
10.6.1 IntFifo Creation and Updating ....71
10.6.2 Multi-Dimensional Array Data ....71
10.6.3 Class and Struct Versions of Data types ....71
10.7 Application Notes ....72
10.7.1 BustoolsCsApp ....72
10.7.2 BustoolsInterop ....72
11 • LabVIEW Support .... 73
11.1 Information ....73
11.2 System Requirements ....73
11.3 LabVIEW VI Examples ....73

12 • VxWorks Support ..... 74
12.1 Introduction ..... 74
12.2 VxWorks Installation ..... 74
13 • UNIX Support ..... 75
13.1 Introduction ..... 75
13.2 Compiling Applications ..... 76
13.3 Linux Installation ..... 76
13.4 LynxOS Installation ..... 76
14 • Integrity Support ..... 77
14.1 Introduction ..... 77
14.2 Integrity Installation ..... 77
15 • Porting the API to Other Environments ..... 78
15.1 Supported Compilers ..... 78
15.2 API Source Code ..... 78
15.3 Rebuilding the API ..... 79
15.3.1 Customizing the BusTools/1553-API Using Pre-Defined Symbols ..... 80
15.3.2 Other Build Symbols ..... 82
15.4 Windows Calling Conventions ..... 83
15.5 Porting to an Unsupported Operating System ..... 84
15.5.1 Endian Issues ..... 84
15.5.2 Conversion Steps ..... 84
15.6 Rebuilding the BusTools/1553-API Library ..... 91
15.6.1 Changing Project Options ..... 91
15.6.2 Building the DLL ..... 94
15.7 Building a Unix Platform Shared Library ..... 94

# List of Figures

Figure 15-1 Visual Studio Project Propertis ........ 92
Figure 15-2 Visual Studio Project Compiler Include File Path Properties ........ 92
Figure 15-4 Visual Studio Project Compiler Code Generation Properties ........ 93
Figure 15-3 Visual Studio Project Linker Directories File Path ........ 93

# List of Tables

Table 2-1 Abaco Systems MIL-STD-1553 Products .... 15
Table 2-2 1553 Board Feature Guide .... 16
Table 3-1 Time-tag Mode Options .... 19
Table 3-2 Time-tag Initialization Options .... 20
Table 3-3 TTDisplay Options .... 20
Table 3-4 Recommended Settings for Time-tag Modes .... 21
Table 6-1 Allowed BC Timing Modes for Static TNOP and Dynamic TNOP .... 39
Table 6-1 Message Scheduling .... 42
Table 6-2 Single/Double Data Buffer Functionality with F/W V4/5 .... 49
Table 6-3 Single/Double Data Buffer Functionality with F/W V6 .... 49
Table 6-4 Multiple Data Buffer Functionality with F/W V6 and API 8.0+ .... 50
Table 8-1 Interrupt Events .... 57
Table 8-2 Interrupt Events Filters .... 62
Table 13-1 UNIX Support Matrix .... 75
Table 15-1 Rebuilding the API on Various Environments .... 79
Table 15-2 Currently Supported Platforms and Operating Systems .... 79
Table 15-3 Symbols that Customize the BusTools/1553-API .... 80
Table 15-4 Other Symbols .... 82
Table 15-5 Structure Size for Key BusTools/1553-API Structures for Boards Running V5 or Earlier Firmware .... 88
Table 15-6 Structure Size for Key BusTools/1553-API Structures for Boards Running V6 Firmware .... 88

# 1 • Unpacking and Handling

## 1.1 Unpacking Procedures

Any precautions found in the shipping container should be observed. All items should be carefully unpacked and thoroughly inspected for damage that might have occurred during shipment. The board(s) should be checked for broken components, damaged printed circuit board(s), heat damage and other visible contamination. All claims arising from shipping damage should be filed with the carrier and a complete report sent to Abaco Systems Customer Care.

## 1.2 Handling Precautions

Electronic assemblies use devices that are sensitive to static discharge. Observe anti-static procedures when handling these boards. All products should be in an anti-static plastic bag or conductive foam for storage or shipment. Work at an approved anti-static workstation when unpacking boards.

# 2 • Overview

## 2.1 Introduction

This manual is the guide for using Abaco Systems' MIL-STD-1553 product line using a common Application Program Interface (API). Abaco Systems' MIL-STD-1553 interface boards offer a wide range of options, compatible with most system configurations. Most importantly, these products all come with a common software package providing C-Language software control providing the ability to perform rapid application development.

## 2.2 BusTools/1553-API

All Abaco Systems' MIL-STD-1553 products share a common architecture and use the same powerful "C" language Application Program Interface (API) called BusTools/1553-API. This software library is available for 32-bit Windows XP and 32-bit/64-bit Windows 7/2008R2 (SP1 and KB3033929 reqd), 8, 8.1, Window Server 2012 R1/R2 and 10. This manual refers to these Windows versions as "Windows". The API also runs with Linux, VxWorks, Integrity, and LynxOS.

The API supports single, dual, and multi-function boards. This allows applications written for a single-function or dual-function boards to run a multi-function board, without changing the original code.

API Routines control initialization, Bus Monitor (BM), Remote Terminal (RT), and Bus Controller (BC) functions, as well as interrupt handling, timing modes, discretes/triggering and other board features. API routines are easy to use and speed the development of applications. Programmers are not required to know board details or on-board memory management. BusTools/1553-API provides a common interface across PCI, PCI Express, Mini-PCI Express, PMC, XMC, AMC, VME/VXI, PC/104 Plus, and ExpressCard platforms, providing for flexible and reusable software.

This is useful when expanding systems or migrating between platforms.

### 2.2.1 Interface to Other Languages

Abaco Systems supplies interface files and example programs to assist with application use of the API library functions in other languages like C#. See Chapter 10, "C# Support" for more information regarding development with the API in this environment.

Support for National Instruments LabVIEW is provided in a separate product called "LV-1553".

### 2.2.2 API Source Code

API source code is included with the API distribution if you need to adapt the API to other platforms and operating systems.

## 2.3 Supported Hardware

The following table shows Abaco Systems boards supported by BusTools/1553-API. For detailed descriptions of all products, including available options and carriers, refer to the Abaco Avionics site.

LINK
https://www.abaco.com/products/avionics

Table 2-1 Abaco Systems MIL-STD-1553 Products

| Product Name | Bus | Number of Channels | Latest F/W Version |
|---|---|---|---|
| R15-USB | USB | 1 or 2 | 6.11 |
| BT3-USB-MON | USB | 1 | 6.11 |
| QPCI-1553 | PCI | 1, 2 or 4 | 4.68 |
| QPCX-1553 | PCI | 1, 2 or 4 | 6.03 |
| QCP-1553 | CompactPCI | 1, 2 or 4 | 6.03 |
| RPCC-D1553 | PCMCIA | 1 or 2 | 4.40 |
| R15-EC | Express Bus | 1 or 2 | 6.03 |
| RXMC-1553 | XMC | 1 or 2 | 6.09 |
| RXMC2-1553 | XMC | 1, 2 or 4 | 6.17 |
| RAR15-XMC-IT/RAR15XF | XMC | 1, 2 or 4 | 6.17 |
| QPMC-1553 | PMC | 1, 2 or 4 | 4.66 |
| QPM-1553 | PMC | 1, 2 or 4 | 6.17 |
| RPCIE-1553 | PCI Express | 1, 2 or 4 | 6.08 |
| R15-LPCIE | PCI Express | 1 or 2 | 6.03 |
| R15-MPCIE | PCI Express | 1 or 2 | 6.09 |
| R15-AMC | AMC | 1, 2, or 4 | 4.40 |
| R15-PMC | PMC | 1 or 2 | 6.03 |
| RQVME2-1553* | VME | 1, 2, or 4 | 4.40 |
| QVME-1553 | VME | 1, 2 or 4 | 4.40 |
| Q104-1553-P | PC/104-*Plus* | 1, 2 or 4 | 6.03 |

* The RQVME2-1553 is a RoHS redesign of the QVME-1553

## 2.4 Hardware Features

The following table provides more details regarding the features available on Abaco Systems MIL-STD-1553 interface boards.

Table 2-2 1553 Board Feature Guide

| 1553 Board | Avionics Discretes | EIA-485 | IRIG-B | Hard Wired RT Addressing | Test Bus | LRU Bus | DMA | Ext Trig In/Out |
|---|---|---|---|---|---|---|---|---|
| R15-USB | 8 | No | Yes | No | No | No | No | Programmable |
| QPCI-1553 | 10 | Yes | Optional | Ch 1 | Yes | Yes | No | Programmable discretes |
| QPCX-1553 | 10 | Yes | Optional | Ch 1 | Yes | Yes | Yes | Program discretes |
| QCP-1553 | 18 | Yes | Optional | Ch 1 and Ch 2 | No | No | Yes | Program discretes |
| QPMC-1553 | 18 | Yes | Optional | Ch 1 and Ch 2 | No | No | No | Program discretes |
| QPM-1553 | 18 | Yes | Optional | Ch 1 and Ch 2 | No | No | No | Program discretes |
| R15-PCIE | 18 | Yes | Optional | Ch 1 and Ch 2 | No | No | No | Program discretes |
| R15-LPCIE | 14 | Yes | Optional | Programmable | No | No | No | Output - Prog Disc. Input Per channel |
| R15-MPCIE | 2 | Yes | No | Programmable | No | No | No | Program discretes |
| R15-PMC | Optional | Optional | Optional | Yes | No | No | No | Optional |
| R15-AMC | 18 | Yes | Optional | Ch 1 and Ch 2 | No | No | No | Program discretes |
| PCC-D1553 | 2 | No | Optional | No | No | No | No | Program discretes |
| R15-EC | 2 | No | Optional | No | No | No | No | Program Discretes |
| RXMC-1553 | 4 dedicated + 8 Discrete or PIO (optional) | 4 Optional | Optional | Yes | No | No | No | Output - Prog Disc. Input Per channel |
| RXMC2-1553 | 12 | Yes | Optional | Programmable | No | No | No | Output - Prog Disc. Input Per channel |
| RAR15-XMC-IT RAR15XF | 6 | No | IRIG-IN | Programmable | No | No | No | Build option |
| QVME-1553 RQVME2-1553 | 4 | No | Optional | All Channels | Yes | No | No | Per channel |
| Q104-1553P | 10 | Yes | Optional | Ch 1 | No | No | No | Program discretes |

# 3 • Operational Modes and Timing

This chapter focuses on general programming topics such as operational modes, time-tags, and IRIG-B timing. The chapters that follow cover programming the Bus Controller, Bus Monitor, and Remote Terminal along with interrupt programming and memory usage.

## 3.1 Single-function, Dual-function, and Multi-function Modes

Abaco Systems 1553 boards come in single-function, dual-function, and multi-function configurations. Single-function boards allow operation in only one of the three 1553 functions at a time: Bus Controller, Bus Monitor, or Remote Terminal (up to 31 RTs). Although all three modes are available, the board can operate only in one mode at any one time. If your application attempts to run more than one function on a single-function board, a single-function board error message appears.

Dual function boards are available on firmware versions 4.50 or greater. Dual function allows Bus Monitoring with Remote Terminal or Bus Controller operation. This allows application to execute as a Bus Monitor along with either Bus Controller or Remote Terminal functions. Dual function boards report a dual-function board error if the application tries to run the Bus Controller and Remote Terminal together.

Multi-function boards allow any combination of the three modes at the same time. This means a multi-function board can simulate a 1553 bus with a Bus Controller, Bus Monitor, and up to 31 Remote Terminals.

## 3.2 RT Validation Mode (Single RT)

Starting with firmware version 5.00 individual channels can be programmed to support RT Validation mode. In this mode, the channel will pass the RT Validation Test Plan Protocol tests (MIL-HDBK-1553 Appendix A, 24 September 1986). This mode limits the channel to a single operational RT address and is called *Single RT Mode*. When a channel is in RT Validation mode, the Bus Monitor function can be utilized; however, the Bus Controller function is not available.

Single RT Mode is set on a channel-by-channel basis and is persistent through power-cycles. Individual channels can be configured for RT Validation Mode while other channels support normal multi-RT mode. For information on RT Electrical and Noise validation testing, or configuration of channel features including RT Validation mode, Bus Monitor Only Mode, and Hardwired RT address, contact Abaco Systems Avionics technical support.

LINK
avionics.support@abaco.com

## 3.3 MIL-STD-1553A and MIL-STD-1553B Operation

All Abaco Systems 1553 interface channels can be initialized in either MIL-STD-1553A or MIL-STD-1553B modes. Select the operational mode in the initialization call (BusTools_API_OpenChannel or BusTools_API_InitExtended). Refer to the "BusTools 1553-API Reference Manual" for more information on Initialization.

If you initialize the channel to run in MIL-STD-1553B mode, you can select individual Remote Terminal addresses to run the MIL-STD-1553A protocol. Invoke BusTools_Set1553Mode to select the RT addresses on which to support the 1553A protocol. If you initialize the channel for 1553A, then all RT addresses operate in 1553A protocol and cannot be switched to support 1553B.

### 3.3.1 MIL-STD-1553A Operation

When the channel or RT address is operating in MIL-STD-1553A mode, the firmware ignores the T/R bit in the mode code. MIL-STD-1553A defines only mode code 0x0, Dynamic Bus Control. There is no data associated with any mode code. The remaining mode codes are undefined and are system dependent.

Since MIL-STD-1553A does not use broadcast messages or subaddress 31, they are disabled when 1553A operation is selected.

### 3.3.2 MIL-STD-1553B Operation

When the channel or RT address is operating in MIL-STD-553B mode, the firmware defines all 1553B mode codes. Initialization enables subaddress 31 and broadcast messages. Call BusTools_SetSa31 and BusTools_SetBroadcast to disable these settings.

MIL-STD-1553B defines mode codes 0x0 through 0x8 and 0x10 through 0x15. In MIL-STD-1553B, mode codes 0x0 through 0xf do not have an associated data word, and mode codes 0x10 through 0x1f have an associated data word. The MIL-STD-1553 Tutorial provides a complete list of mode codes.

## 3.4 Setting and Using Time-tags

Time-tags are time stamps on each 1553 message. For F/W version v5.x and earlier, time-tags use an internal clock with a 1-microsecond resolution and a 45-bit time-tag register. Starting with F/W version 6.0 and API version 8.00 time-tags use an internal clock with a 1-nanosecond resolution and a 64-bit time-tag register. These registers keep track of the time once a function (BC/BM/RT) starts running. This register defaults to zero, but using BusTools/1553-API functions, you can select different starting values.

BusTools/1553-API can synchronize time-tags to an external source and some of the Abaco Systems 1553 products have IRIG-B capability.

### 3.4.1 Time-tag Modes

All time-tags for a channel come from a single time-tag register. However, the user can implement several timing modes. Timing modes determine how timing operates for the channel. The table below shows the time-tag mode options.

Table 3-1 Time-tag Mode Options

| TTMode | Time-tag operating mode: |
|---|---|
| API_TTM_FREE | Free running time-tag counter (Default). |
| API_TTM_RESET | Time-tag counter is reset to zero on external TTL input discrete active. |
| API_TTM_SYNC | Synchronize the time-tag to the external TTL input. The TTPeriod parameter sets the period of the external TTL input in microseconds. |
| API_TTM_RELOD | Time-tag counter is reset to the value previously loaded into the Time-tag Load register. |
| API_TTM_IRIG | Time-tag counter is reset to the IRIG time from either an external or an internal IRIG source. Board must have IRIG firmware to support this option. |
| API_TTM_AUTO | Time-tag counter is automatically set to the increment of the value store in the time-tag counter load register on an external sync pulse |
| API_TTM_XCLK | Time-tag counter is updated using an external 1MHz clock. Default is to use rising edge. Set lparm1 to TIME_EXT_EDGE to use the falling edge. Requires firmware version 5.00 or higher on RXMC2-1553 only. |

API_TTM_FREE is the default setting for time-tag mode. It allows the time-tag register to run continuously. The other options depend on either an external sync pulse or an external IRIB-B time input. IRIG-B is an option available on some boards. Later sections discuss the IRIG-B function.

API_TTM_RESET, API_TTM_SYNC, API_TTM_RELOD, and API_TTM_AUTO modes depend on an external TTL pulse. API_TTM_SYNC and API_TTM_AUTO are similar functions. The API_TTM_SYNC mode allows the application to sync the time-tag to an external pulse. However, the application must continually set the Time-tag Counter Load Register with the next sync value. For example, if you sync on a 1 second pulse, you increment the initial value by 1,000,000 after each sync pulse. The API_TTM_AUTO automatically increments the Time-tag Counter Load Register on each external pulse. The API_TTM_AUTO is the preferred method to sync the time-tag to an external pulse. API_TTM_SYNC is deprecated and only for legacy code already using that method.

To use the API_TTM_AUTO, program both the Time-tag Counter Load Register with the base start time and the Time-tag Increment Register with the time increment of the sync pulse. You can use a sync pulse increment of 1μs up to 1.193 hours.

API_TTM_RESET and API_TTM_RELOD are similar modes that reset the time-tag to a previous value on an external pulse. API_TTM_RESET resets the time-tag to 0 while API_TTM_RELOD reset the time-tag to an initial setting. The application calls BusTools_TimeTagWrite prior to selecting API_TTM_RELOD to set the re-load value.

API_TTM_XCLCK (RXMC2-1553 only) uses an external clock source instead of the board's clock. The clock input connects through the differential input.

### 3.4.2 Time-tag Initialization

There are several options regarding how an application may choose to initialize channel time for the selected timing mode. The table below defines the timer initialization options available to individual channels on a 1553 board.

Table 3-2 Time-tag Initialization Options

| TTInit | Time-tag Initialization Mode: |
|---|---|
| API_TTI_ZERO | Time set to zero. (Default) |
| API_TTI_DAY | Time of day, relative to midnight, is loaded into the Time-tag counter (Host Clock reference) |
| API_TTI_IRIG | Time of year based on IRIG format (Host clock reference or actual IRIG-B input) |

Time-tag initialization pre-loads the time-tag counter register with a starting value. Both the API_TTI_DAY and API_TTI_IRIG use the host clock to calculate a time in microseconds. An application can override this value by call BusTools_TimeTagWrite after the call to BusTools_TimeTagInit.

### 3.4.3 Formatting and Displaying Time-tags

Each 1553 message has a BT1553_TIME structure that records the time of the message transaction. This time is based on the initial time-tag setting. BT1553_TIME contains the raw time-tag data in µs. An application can display the time-tag as a string by calling BusTools_TimeGetString.

BusTools_TimeGetString displays the time-tag according to the setting of the Time-tag Display parameter in the call to BusTools_TimeTagMode. The table below shows the display options.

Table 3-3 TTDisplay Options

| TTDisplay | Display Type: |
|---|---|
| API_TTD_RELM<br>API_TTD_REML_NS | Relative to midnight format "(ddd)hh:mm:ss.useconds".<br>Only those components necessary are displayed (e.g., if *days* is zero it is not displayed). Default display mode. |
| API_TTD_IRIG<br>API_TTD_IRIG_NS | IRIG Format "(ddd)hh:mm:ss.uuuuuu". Formatting:<br>ddd = days; hh = hours; mm = minutes;<br>ss = seconds; uuuuuu = microseconds.<br>All components displayed; fixed format. |
| API_TTD_DATE<br>API_TTD_DATE_NS | Date Format "(MM/dd)hh:mm:ss.uuuuuu". |

Match the TTInit and TTDisplay values to get a consistent recording and display of the data. Use the _NS version of the display parameter when using V6 firmware to convert time-tags having a one nanosecond resolution.

### 3.4.4 Selecting Time-tag Options

There are six time-tag modes, three time-tag initialization options, and three time-tag display options. Those options provide a wide variety of time-tag settings, all of which will work, but only a few combinations are useful. The following tables show the recommended setting for each time-tag mode ("+" indicates *recommended*; "?" indicates *supported, not recommended*; <blank> indicates *not supported*).

Table 3-4 Recommended Settings for Time-tag Modes

| Time-tag Mode | Init/Display Options | API_TTI_ZERO | API_TTI_IRIG | API_TTI_DAY |
|---|---|---|---|---|
| Free Mode/XCLK | API_TTD_RELM | + | ? | ? |
| | API_TTD_IRIG | ? | + | + |
| | API_TTD_DATE | ? | + | + |
| IRIG Mode | API_TTD_RELM | | | |
| | API_TTD_IRIG | | + | |
| | API_TTD_DATE | | + | |
| Reset/Reload Mode | API_TTD_RELM | + | ? | ? |
| | API_TTD_IRIG | + | ? | ? |
| | API_TTD_DATE | + | ? | ? |
| Sync/Auto Mode | API_TTD_RELM | + | + | + |
| | API_TTD_IRIG | + | + | + |
| | API_TTD_DATE | + | + | + |

### 3.4.5 Controlling and Synchronizing Time-tags

Free run is the default time-tag setting. This means the internal clock increments the time-tag register 1/µs. The internal clock has a 10-25 µs drift per second. This is usually not a problem when running a single 1553 bus. However, this drift can cause problems matching messages from different systems when running several 1553 buses across several platforms. BusTools_TimeTagMode offers several settings for synchronizing time-tags. These include External IRIG time input and synchronizing to an external TTL pulse.

BusTools_TimeTagMode sets up both the Time-tag Counter Load Register and the Time-tag Increment Register. It sets the base time according to the TTInit selection and sets the timer increment to the value of the TTPeriod. The following code examples show the results of various parameter settings in the call to BusTools_TimeTagMode on how the API records and displays time-tags. See the "BusTools/1553-API Reference Manual" for a full description of BusTools_TimeTagMode.

status = BusTools_TimeTagMode(cardnum,API_TTD_DATE,
**API_TTI_ZERO**, API_TTM_FREE, NULL, 0, 0, 0 );

time-tag = 2.801817

status = BusTools_TimeTagMode(cardnum,API_TTD_DATE,
**API_TTI_DAY**, API_TTM_FREE, NULL, 0, 0, 0 );

time-tag = 13:42:56.540472

status = BusTools_TimeTagMode(cardnum,API_TTD_DATE,
**API_TTI_IRIG**, API_TTM_FREE, NULL, 0, 0, 0 );

time-tag = (6/7)12:44:43.562962

status = BusTools_TimeTagMode(cardnum,**API_TTD_IRIG**,
API_TTI_IRIG, API_TTM_FREE, NULL, 0, 0, 0 );

time-tag = (158)12:47:54.719961

status = BusTools_TimeTagMode(cardnum,**API_TTD_RELM**,
**API_TTI_DAY**, API_TTM_FREE, NULL, 0, 0, 0 );

time-tag = 12:49:48.605472

### 3.4.6 Reading Time-tags

BusTools/1553-API allows applications to read the time-tag counter. This is an asynchronous read of the time-tag register, unrelated to a specific message time-tag. BusTools_TimeTagRead returns a BT1553_TIME structure. With BusTools/1553-API version 5.x and earlier, the BT1553_TIME structure is 48-bits. In F/W version 6.x the BT1553_TIME structure is 64-bits. Depending the firmware version, the time-tag resolution is either 1μs or 1ns.

## 3.5 Using IRIG-B on Selected Boards

Some Abaco Systems 1553 boards support IRIG-B timing. See Table 2-2. IRIG-B is an optional feature. If your board has IRIG-B, you can use an external IRIG-B signal for timing or generate an IRIG-B signal on the board. The IRIG-B-capable boards can also generate an IRIG B002 signal externally.

BusTools/1553-API provides several functions to support IRIG-B. Function names all start with a "BusTools_IRIG_" prefix. Call BusTools_IRIG_Config to configure IRIG timing. This function lets you select internal or external IRIG-B signal source. It also sets whether the board outputs the IRIG-B signal. If you use an external or internal IRIG-B signal for timing, you must set the TTMode in the call to BusTools_TimeTagMode to API_TTM_IRIG.

When using an external IRIG-B signal, the application should execute the API calibration function, BusTools_IRIG_Calibration. This function calibrates the IRIG DAC by adjusting the peak detection threshold to 82.5% of the maximum peak using the formula Vmin + .825(Vmax – Vmin), where Vmax is the maximum peak amplitude detection level, and Vmin is the minimum peak detection level. The IRIG DAC adjustment performed by the IRIG calibration applies to all channels on multi-channel boards and is not a channel-specific operation.  This operation is not supported with the RPCC-D1553 or PCCard-1553 boards.

The incoming IRIG-B signal updates the time-tag register 1/s. The onboard clock provides the μs timing. If you lose the external IRIG-B signal, the internal clock continues generating time-tags. Call BusTools_IRIG_Valid periodically to make sure there is a valid IRIG-B signal. If you use the internal IRIG-B, you need to call BusTools_IRIG_SetTime to set the IRIG-B time. You can either pass time or use the system time. See the "BusTools/1553-API Reference Manual", for details on the BusTools_IRIG_xxxx functions.

# 4 • Bus Monitor

The Bus Monitor (BM) function provides the ability to monitor and record all messages, or a defined subset of the messages on the 1553 bus. In addition, the BM provides timing information and detailed error status for each message word (command, data and status).

This chapter describes the BM functionality and the API routines that control the BM functions.

## 4.1 BM Hardware Operation

The BM records information about each message on the 1553 bus. It records the message data and status about the data transmission in a BM Message Buffer in memory. The application program must initialize a list of BM Message Buffers in channel memory and tell the microcode where the list is located (by storing the channel memory address of the list in one of the RAM Registers). In F/W version 5.x and earlier the list of BM Message buffers is a circular linked list (that is, each buffer has a pointer to the location of the next buffer in the list, and the last buffer has a pointer to the first buffer in the list). Starting with F/W version 6.0 the BM message buffer is composed of variable size buffers and the size of the current BM message determines the start of the next buffer. The buffer size is in bytes, not the number of fixed sized messages. A message recorded at the end of the buffer will wrap to the start.

As messages appear on the 1553 bus, the board microcode fills in the BM Message buffer with message data. The microcode then checks the 32-bit Interrupt Status Bits for this message against the set of Interrupt Enable Bits specified in the message buffer. If there are any matching bits (if the logical AND of the two 32-bit values is non-zero), an Interrupt Message is added to the Interrupt Queue.

Finally, the microcode sets an internal register to point to the next message buffer in the list and asserts the hardware interrupt output (if enabled).

The Bus Monitor can run under interrupt control. Chapter 8, “Interrupt Queue and Interrupts” defines the interrupt queue structure. Once an interrupt is received, the address is recorded in the interrupt queue and a hardware interrupt signals the application that an interrupt message containing the channel memory address of the BM Message Buffer generating the interrupt. Any application interfacing to a board running with F/W version 5.x or earlier will convert the memory address to a specific BM Message Buffer using BusTools_BM_GetMessageid. The application reads the BM Message from channel memory using BusTools_BM_MessageRead. The application can then process the message as needed. Any application interfacing to a board running F/W version 6.0 or greater uses the address to a message to read the message data out the BM Buffer. Starting with F/W version 6.03 there is an option to suppress the hardware interrupt, but still allowing the address of the BM message to be recorded in the interrupt queue.

As messages appear on the 1553 bus, the microcode sequences through the BM Message Buffer list recording the data. There is no check to see if there is data in the buffer or if the application program has read the data; therefore, the user's application must allocate enough BM Message Buffers to ensure it can process each buffer before the microcode "wraps-around" and overwrites the data. The user must consider the expected message rate and time required to process the information in each message when deciding on how many buffers to allocate.

The available RAM on the 1553 board determines the maximum number of BM Message Buffers. See Chapter 9, "Board Memory Organization" for more details.

When the API initializes the Bus Monitor function, the default is to monitor and capture all messages. However, the Bus Monitor can filter messages. The BM Filter defines the filtering. The filtering operation is on the message command word (or the first command word in the case of "RT to RT" messages). The filter enables or disables each possible command word for BM recording. In addition, the filtering operation can be set up to record every nth occurrence of each command word type.

Finally, the application can define one or more trigger events used to start the BM recording operation. Before the trigger event occurs, the BM does not record Messages. Once the trigger event occurs, the BM records all messages, subject to the BM Filter. The trigger events are set up using a command, data or status word of incoming messages.

- For each message recorded by the BM, the microcode stores the following items in the BM Message Buffer:
- Each message word; command, data, and status, along with a 16-bit BM Status Word for that word
- The message time-tag (the time counter as of the mid-zero crossing of the parity bit of the command word).
- The 32-bit Interrupt Status word for the entire message; this word is a combination of all the information about the operation of the message. It includes information about the type of message (RT-RT, Mode Code), Completion status (End of Message, Retry Occurred), and Error Conditions (No Response, parity, bit count, etc.).

In addition, the BM measures the RT response time and stores it in the message buffer. The microcode records the information to the message buffer as it sees it on the bus.

The only word of the message that is always in the same place is the command word (the first command word in the case of "RT to RT" messages) and the BM Word Status Bits. After that, the contents of the message buffer depend on the type of message and number of words in the message. Take for example, a "receive" message with 10 data-words. This message has the data immediately following the Command Word's BM Word Status Bits. The data consists of the data value and the BM Word Status Bits for each of the 10 data words (20 words). Following the data is the RT response time (the time the RT required to return its status word). Finally, the BM records the status word and its BM Word Status. For a “transmit” message, the response time and status information is before the data (because the status word appears before the data in the message).

Applications using BusTools/1553-API need not worry about this complexity. The API returns the BM Message Buffer as a fixed format fixed length structure that the application can easily use. See the BM Message Buffer (API_BM_MBUF) structure for details. This is the same structure used to write the Bus Monitor data to the BusTools-1553 Bus Monitor Recording File.

## 4.2 BM Software Operation

This section describes the BusTools/1553-API routines that handle BM operations. The Bus Monitor initialization performs certain hardware set up for the respective channel, such as memory initialization, interrupt queue definition and error-injection-buffer definition. Because of these initialization steps, if you are programming a Bus Monitor, your application must call BusTools_BM_Init prior to executing any functions that configure the Bus Controller or Remote Terminal. You no longer need to call BusTools_BM_Init if you are not programming a Bus Monitor.

After completing hardware initialization (using BusTools_API_InitExtended or BusTools_API_OpenChannel), the next routine called is BusTools_BM_Init. This routine sets up the BM for operation. It allocates and initializes the default BM filter and sets it to enable the recording of all messages. It also sets the default BM trigger to start recording immediately. This routine starting in BusTools/1553-API v 8.12 and with firmware 6.03 can now disable BM hardware interrupts. The messages will still record in the interrupt queue, if the interrupt enable is set in BusTools_BM_MessageAlloc, but they will not create a hardware interrupt strobe.

Next, the application typically sets the desired parameters for the BM Filter. BusTools_BM_Init creates a default BM Filter that enables monitoring of all messages. If this behavior is acceptable for your application, no further initialization of the BM Filter buffer is required.

You modify the BM Filter by using the BusTools_BM_FilterWrite routine. With this routine, you specify the filter settings for a particular RT subunit (a RT subunit is a specific RT address / subaddress / transmit-receive flag combination). You can call this routine for all possible RT subunit combinations. The default BM filter setting lets the BM record all messages. You should not call BusTools_BM_FilterWrite if this is acceptable for your application.

The application should then create the required number of BM Message buffers using the BusTools_BM_MessageAlloc routine. The caller specifies the required number of message buffers. Make this number big enough to keep the expected message traffic on the 1553 bus from overwriting the list before the application can retrieve the data. If there is enough memory on the board, the routine creates the requested number of message buffers and links the buffers. If there is not enough memory for the requested number of buffers, the routine creates as many message buffers as possible and returns the actual number created to the caller. If the application is going to use all the memory on the board, defer this call this until you have created the other board structures. The BusTools_BM_MessageAlloc function will create as many BM buffers as will fit into the remaining memory. You need to allocate only the message buffers before you start the Bus Monitor.

When running with firmware version 6.0 or greater, an invocation of BusTools_BM_MessageAlloc no longer creates a meta-buffer of individual message buffers individually addressable based on a message index. Since the message buffer is a variable sized buffer based on the number of words in the message, the allocate function simply creates a buffer with the size equal to the number of 32-word messages specified. It is important to note the actual number of messages fitting into the allocated memory block will depend on the size of the messages the BM encounters on the bus.

For the R15-USB, you will need to create substantially more message buffers than for other devices. This is due to the serial block-transfer nature of the USB bus. If you are monitoring a heavily loaded bus you should allocate ~3000 message buffers. This will use about half of the RAM available for the channel.

The BusTools_BM_MessageAlloc routine also sets the Interrupt Enable Bits in each message buffer. These bits determine the status conditions that generate an interrupt. Sometimes, you want an interrupt for every message detected on the 1553 bus (by setting the BT1553_INT_END_OF_MESS bit). Other times, you might want to get interrupts only if a parity error is detected (this would be done by setting the BT1553_INT_PARITY bit). Any combination of bits can be set to specify the conditions that generate an interrupt.

The Bus Monitor is susceptible to hardware overflows on heavily loaded buses. An overflow occurs when the on-board Bus Monitor buffers are overwritten with new data before the API can process the current message. Allocating enough message buffer memory will reduce or eliminate overflows. Starting with BusTools/1553-API 8.0 and F/W v6.0 there is an interrupt on Bus Monitor Overflow. You can use this to detect overflows by enabling interrupts on EVENT_BM_OVRFLW. You can also get a count of the number of messages it overflowed by reading the BM Overflow Register.

Change the BM trigger by using the BusTools_BM_TriggerWrite routine. Fill in the trigger definition structure prior to making this call. The default BM trigger is defined such that all messages are recorded, beginning with the first message received. If this setting is acceptable for your application, you do not need to call this routine. There is one BM trigger buffer allocated in hardware memory.

Process interrupts by using BusTools_RegisterFunction. BusTools_RegisterFunction is available for all Windows and UNIX operating systems (supporting POSIX threads). See the “BusTools/1553-API Reference Manual” description of the BusTools_RegisterFunction for more details.

Finally, after the BM is completely initialized, start the BM by calling BusTools_BM_StartStop. The BM begins recording messages detected on the 1553 bus (based on BM filter and BM trigger parameters). If any message meets the criteria specified with the Interrupt Enable Bits, an Interrupt Message is generated.

BM operation continues until the application stops the BM (using the BusTools_BM_StartStop routine) or closes the API (using the BusTools_API_Close function).

## 4.3 BM Recording

This feature is available under Windows and Linux. The API periodically polls the BM buffers and reads the accumulated messages. These messages are stored in a large circular buffer within the API.

To setup BusTools/1553-API to use this feature call BusTools_RegisterFunction using EVENT_RECORDER as the filter type (filterType) in the API_INT_FIFO structure. The callback function passed via the API_INT_FIFO structure must call BusTools_BM_MessageReadBlock to read the most recent list of BM Message buffers. This routine keeps track of which messages are most recent and returns an error to the application if it detects a buffer "wrap-around".

BusTools_BM_MessageReadBlock returns the latest BM Message buffers in an application supplied buffer. The application can then store the information to disk.

# 5 • Remote Terminals

This chapter describes the RT functionality and the BusTools/1553-API routines used to program that functionality. The Remote Terminal (RT) function on the 1553 board provides the ability to control the activities of one or more RTs on the 1553 bus. The definition of each RT includes:

- Legal transmit and receive subaddresses.
- Legal word counts for messages to and from each subaddress.
- One or more data buffers to capture data sent to each subaddress (not just legal ones).
- One or more data buffers for each sub-address (not just legal ones).
- The 1553 Mode Codes supported by the RT.
- The interrupt enable bits for each of the message lists.

## 5.1 RT Hardware Operation

For each possible RT (addresses 0 to 31), there is a four-word RT Address Buffer in channel memory. The 32 Address Buffers form a 128-word structure in RT memory. Each RT Address Buffer contains information that controls the microcode operation for a particular RT. The most important entries in the RT Address Buffer are the enable/disable bits, one for each dual redundant bus (designated as bus A or bus B). If both bits are disabled, the RT simulation ignores any messages to that RT address. If you enable either or both bits, then the microcode responds to commands with that RT address on the enabled channel.

The RT Address Buffer also contains the following:

- A 16-bit status word (initialized by the application and maintained by the microcode). This is the message status word transmitted by the RT in response to a received bus command message. The microcode modifies the "Message Error", "Broadcast Command Received", and "Terminal Flag" bits of this status word as required by the MIL-STD-1553 specification. The other bits are set by the application and are not modified by the microcode.
- A 16-bit last command word (to respond to the "Transmit Last Command" mode code).
- A 16-bit Built-In-Test word (initialized by the application to respond to the "Transmit BIT Word" mode code).
- A 1-bit "inhibit-terminal-flag" (initialized by the application and maintained by the microcode in response to the "Inhibit Terminal" and "Override Inhibit Terminal" Mode Codes).
- A RT Monitor Bit (Set by the user to specify monitor mode. When in monitor mode the RT does not respond to commands)

Once you start the RT simulation, and enable simulation for a specific RT address (by enabling either Bus in the RT Address Buffer), the simulation processes any message to that RT address.

Processing begins by checking that the command word is legal. The command is legal if the RT address; transmit/receive flag, subaddress and word count form a legalized combination. The RT address, transmit/receive flag and subaddress from the command word are used as an 11-bit index into the 2048-word RT Filter Buffer. The resulting one-word entry is extracted and used as the address of a RT Control Buffer.

There can be up to 64 RT Control Buffers created for each enabled RT address (one transmit and one receive for each possible subaddress). The application sets up RT Control Buffers containing a single bit for each possible word count. If the bit is set, then that word count is legal for the associated RT address/subaddress/transmit-receive flag combination. The RT Control Buffer also contains the address of the first or next RT Message Buffer defined for this RT subunit.

If the command is considered legal, the microcode uses the RT Message Buffer address specified in the RT Control Buffer to process the message. If the message is a receive message, the data is stored in the specified RT Message Buffer. If the message is a transmit message, the data is transferred from the specified RT Message Buffer to the 1553 bus.

The final level of complexity is that each RT subunit can have multiple RT Message Buffers defined. The “RT Message Buffer” described above is a circular linked list of RT Message Buffers (that is, each RT Message Buffer contains the address of the next buffer in the list and the last buffer contains the address of the first buffer). Each time the microcode processes a command word that maps to a specific RT Control Buffer, the next RT Message Buffer in the linked list is saved in the RT Control Buffer. BusTools/1553-API is responsible for initializing the message buffers properly.

After processing each message, the microcode checks the 32-bit Interrupt Status Word against the set of Interrupt Enable Bits specified in the RT Message Buffer. If there are any matching bits (if the logical AND of the two 32-bit values is non-zero), an Interrupt Message is added to the Interrupt Queue and if enabled the hardware interrupt line is asserted. See Chapter 8, “Interrupt Queue and Interrupts” for more details about how the interrupt queue works.

Prior to turning on the RT using BusTools_RT_StartStop, the application must initialize all 32 RT Address Buffers (1 for each possible RT), and the entire 2048-word RT Filter Buffer. If you are not using a RT, disable the associated bus enable/disable bit in the RT Address Buffer.

Each entry in the 2048-word RT Filter Buffer must point to a valid RT Control Buffer. For any disabled RTs, the RT Filter Buffer entries can point to the same default “no-op” RT Control Buffer. The call to BusTools_RT_Init creates 32-default RT Control Buffers, one for each possible RT, and sets the addresses in the RT Filter Buffer to point to these buffers. It also creates 32-default RT Message Buffers.

## 5.2 RT Software Operation

This section describes the BusTools/1553-API routines, which control RT operations. You can find detailed syntax and error status information in the "BusTools/1553-API Reference Manual".

Prior to calling any BusTools/1553-API RT routine, initialize the RT-specific global parameters by calling the following functions:

- BusTools_SetBroadcast
- BusTools_SetSa31

Then, initialize RT operations using the BusTools_RT_Init routine. This routine sets up the 32 RT Address Control buffers (disabled), the 2048-word RT Filter Buffer, the 64 or 62 broadcast and 32 non-broadcast RT Control Buffers and the 32 default RT Message Buffers used for all disabled or illegal RT messages. At this point, you can start the RT with the BusTools_RT_StartStop routine (though all RTs are disabled).

The following steps enable RTs for operation:

1. Create the RT Control Buffers for any legal subunit (subaddress and transmit / receive flag) for a RT. In addition, create the linked list of RT Message Buffers for this subunit.

   BusTools_RT_CbufWrite routine handles this setup. The application can call this routine up to 64 times per simulated RT (32 subaddresses, for transmit and receive). This routine is called to establish legal Mode Codes and word counts. Mode codes are simply messages directed to subaddress 0 (or 31 if subaddress 31 mode codes are enabled). If subaddress 31 mode codes are enabled, the RT message buffers allocated to subaddress 0 are used for subaddress 31 messages. Note that a value of 0x00000001 legalizes word count 32 and a value of 0x00000002 legalizes word count 1.
2. Initialize the RT Broadcast Control buffers (if broadcast messages are enabled). Do not attempt to read or write the Broadcast Control buffers if broadcast is not enabled; the function will return an error.

   You must call BusTools_RT_CbufbroadWrite for each subaddress and Transmit/Receive combination enabled for Broadcast Receive or Transmit. Each call to this function establishes a buffer of 31 32-bit control words that are accessed by the firmware each time a broadcast message is detected. (The firmware accesses this entry through the RT_ADDRESS_BUFFER, for RT=31).

   When the RT receives a broadcast message, the firmware sequences through the 31 control words and checks to verify the specified word count is enabled for the associated RT. If it is, the firmware sets the "Broadcast Message Received" bit (bit 4) in the 1553 Status Word for the associated RT.

If you enable subaddress 31 mode codes, subaddress 31 messages use the RT message buffers allocated to subaddress 0.

3. Initialize the RT Message Buffers, as required.

   BusTools_RT_CbufWrite initializes the RT Message Buffers to zero, initializes the error injection pointer, and properly links the buffers together. For all message types (transmit or receive) the RT Message Buffer must contain the desired Interrupt Enable Bits and a valid Error Injection Buffer pointer. In addition, for transmit subunits; the RT Message Buffer(s) should contain the data to be transmitted. The content of each RT Message Buffer is initialized with the BusTools_RT_MessageWrite routine. This routine should be called for every message buffer defined for every subunit of the RT.

   The BusTools/1553-API defines one default message buffer for each RT. While the data values in this default buffer probably do not matter (since many RT Control Buffers usually point to this buffer), the application should set Interrupt Enable Bits.

   For example, the application could set the Interrupt Enable Bits to generate an interrupt any time this buffer is accessed using the "BT1553_INT_END_OF_MESS" bit. This would let the application know any time an un-initialized RT subunit is accessed. The default RT Message Buffer can be set using the BusTools_RT_MessageWriteDef routine.
4. After the RT Control Buffers and RT Message Buffers have been initialized, write the RT Address Control Block, using the BusTools_RT_AbufWrite routine.

   Set the bus-enable bits to enable at least one bus. The application should initialize the status word and BIT word as required. The bit word can either be stored in the address buffer bit word parameter or stored in the RT message buffer, data word 0. If the inhibit terminal flag parameter is OR'ed with 0x10 then bit word data is stored in the message buffer. Otherwise it is stored in the bit word parameter of the address buffer.

   Once one or both bus enable bits have been set (and assuming the RT simulation has been started with the BusTools_RT_StartStop routine), the RT simulation for the specified RT begins.
5. The application software needs to handle interrupts received from the RT. The interrupt queue structure is defined in Chapter 8, "Interrupt Queue and Interrupts".

   The interrupt queue provides the channel memory address of the interrupting RT Message. Once the application receives an interrupt, it converts the memory address to a specific RT Message Buffer using BusTools_RT_GetMessageid. The application then reads the RT Message Buffer from channel memory by calling BusTools_RT_MessageRead. The application can then process the message as needed.

There is another method for processing interrupts, called BusTools_RegisterFunction. This interface takes advantage of the multi-threaded processing features of Windows and UNIX. The BusTools_RegisterFunction interface uses standard WIN32 threads for Windows and should be compatible with other standard WIN32 products that support multiple threads. BusTools_RegisterFunction uses POSIX threads (pthread) for UNIX systems and is compatible with all UNIX systems supporting POSIX threads.

The RT simulation continues for each individual RT until the bus enable bits of the Address Control Block for the RT are cleared.

The entire RT simulation can be turned off using the BusTools_RT_StartStop routine. Starting or stopping either a single RT or the entire RT simulation has no effect on the Bus Controller simulation, or the operation of the Bus Monitor.

If you have a single-function 1553 board, any attempt to run the RT while the BC or the BM is running will fail with an error status of API_SINGLE_FUNCTION_ERR. If you have a dual-function 1553 board you get error status of API_DUAL_FUNCTION_ERR if you attempt to run the RT while the BC is running.

## 5.3 RT Monitor Mode

RT Monitor mode allows the RT to record all messages to a specified RT. However, the RT Monitor does not respond to messages or put any data onto the 1553 bus. All BusTools/1553-API calls are available to read data from the RT Monitor the same as for an RT in standard operating mode. Enable or disable RT Monitor Mode for an RT address by setting or resetting bit 0 of the enable members in the API_RT_ABUF structure. BusTools_RT_MonitorEnable allows the caller to enable or disable this mode during standard operating mode. The application can dynamically change from RT standard operating mode to RT Monitor Mode or from RT Monitor Mode to RT standard operating mode using this command. See the "BusTools/1553-API Reference Manual" for more information on BusTools_RT_MonitorEnable and the structure API_RT_ABUF.

## 5.4 RT Extended Status Mode

The status word returned by the RT is normally set for all subaddresses.

For example, if you set the Busy Bit, it is set for all commands to that RT until cleared. BusTools_RT_AbufWrite and BusTools_RT_MessageWriteStatusWord are used to set bits in the status word. However, if you enable extended status by ORing in RT_ABUF_EXT_STATUS to the inhibit terminal flag in the API_RT_ABUF structure, a unique status can be set for each RT message buffer. Call BusTools_RT_MessageWriteStatusWord to set the status word.

## 5.5 Dynamic Bus Control

Dynamic Bus Control allows the BC to transfer Bus Controller duties to an RT. You can program a single RT within an RT simulation for Dynamic Bus Control Acceptance. The Bus Controller sends the Dynamic Bus Control (DBC) Mode Code (0000) to initiate this operation. You must program the RT to accept the DBC mode code and to act as a Bus Controller. Multi-function boards also allow you to keep running the RT function as well as Bus Controller functions. MIL-STD-1553 paragraph 4.3.3.5.1.7.1, Dynamic Bus Control describes the Dynamic Bus Control function.

You must specifically enable this mode, since it requires passing control of the data bus to another device. Ensure that the device assumes the Bus Controller functions, and that there is only a single Bus Controller running at any time.

Control passes once the RT returns its status word. The RT shows acceptance of the Bus Controller function by setting the Dynamic Bus Control Acceptance bit in the status word. The RT sets this bit only when responding to a Dynamic Bus Control mode code (MIL-STD-1553 paragraph 4.3.3.5.3.10). At that time, the selected RT must start running as a Bus Controller, and the old Bus Controller must stop operations. If the RT does not set the Dynamic Bus Control Acceptance bit, the Bus Controller must maintain functioning as a Bus Controller, and the RT must not start Bus Controller operations.

To implement Dynamic Bus Control (DBC), you must set up both the RT and the Bus Controller (BC). Only setup the RT or RTs that you want to have the DBC Acceptance enabled. The following paragraphs describe the steps needed to setup an RT for DBC Acceptance.

Legalize mode code 0 transmit by calling BusTools_RT_CbufWrite. This call must specify at least one data buffer. You must initialize that RT message buffer with a call to the function BusTools_RT_MessageWrite, to enable interrupts on this message (at least enable “BT1553_INT_END_OF_MESS” interrupts in the “enable” control word). You must also enable the RT DBC mode by calling BusTools_RT_AbufWrite. Enable the RT and enable DBC by setting the “RT_ABUF_DBC_ENA” bit in the inhibit_term_flag word.

On multi-function boards, BusTools_RT_AbufWrite gives you the option of shutting down the RT when accepting the DBC, or leaving the RT running. Setting the “RT_ABUF_DBC_RT_OFF” bit causes the accepting RT to shut off. Resetting this bit causes the API to leave the RT running. If you keep the RT running, clear the DBC acceptance bit in the RT status word before the BC sends the RT another message. You can do this by calling BusTools_RT_AbufWrite and passing a new status word (“status”) that has the DBC acceptance bit clear. You do not need to do this on single-function or dual-function boards since the API turns off the RT simulation on a single function or dual-function board before it turns on the BC.

Before starting the RT simulation, you need to setup the BC, so it is ready to become the Bus Controller when the RT receives the dynamic-bus-control mode code. Create all BC messages and initialize them completely, but do not start the BC (via BusTools_BC_StartStop. The API does this after accepting the DBC mode code.

Normally, the API polls the interrupt queue at a 10-ms rate. This can delay starting the BC for up to 10ms. If this delay is too long, you can enable hardware interrupts by setting the flag (mode) parameter in the call to BusTools_API_OpenChannel, or BusTools_API_InitExtended to API_HW_ONLY_INT. Using hardware interrupts reduces this latency to the interrupt response of the host processor.

The BusTools/1553-API software distribution contains a programming example of Dynamic Bus Control.

# 6 • Bus Controller

The Bus Controller (BC) function on the 1553 board performs the activities of the BC on the 1553 bus. This chapter describes the BC functionality and the BC BusTools/1553-API routines supporting Bus Controller operations.

## 6.1 BC Hardware Operation

The Bus Controller (BC) controls all activity on the 1553 bus. The BC determines which messages appear on the bus and when they appear. The BC executes a pre-defined list of messages (called a Bus List or a Major Frame) organized into sets of messages called Minor Frames. Each message in a Minor Frame executes sequentially, with an optional delay between messages. The execution of each Minor Frame starts at a specified time interval. There are two options for creating a bus list: frame messaging and message scheduling.

### 6.1.1 Frame Messaging

The first option for creating a bus list is the traditional frame messaging. The user writes each message into the frames in which they are to transact. For example, an application could define three Minor Frames, one with ten messages, one with 25 messages and the last with one message. If the user needs a message to repeat in each frame, the application must put that message into each frame.

If the Minor Frame rate is set to 100ms, a Minor Frame is started every 100ms. The major frame re-starts every 300ms with the start of Minor Frame #1.

1. When the BC operation starts, it executes Minor Frame #1 immediately (the BC sends the ten messages in that Minor Frame in sequence).
2. The BC will remain idle until the beginning of the next 100-ms time interval, and then it executes Minor Frame #2 (sending the 25 messages in that Minor Frame).
3. Once again, the BC operation will remain idle until the beginning of the next 100-ms time interval, and then it executes Minor Frame #3 (sending the one message in that Minor Frame).
4. If you configure the BC to execute the Minor Frames only once (“1shot” mode), then the BC operation halts.
5. If you set the BC to “loop” on the Minor Frames, then the BC operation will remain idle until the beginning of the next 100-ms time interval. Then it restarts with the first Minor Frame or the Minor Frame defined as the start of the loop.

The application must ensure that the contents of each Minor Frame can execute within the programmed Minor Frame time. If a Minor Frame exceeds the Frame Rate, the BC simulation sets the “Minor Frame Overflow” bit and skips to the next minor frame.

### 6.1.2 Message Scheduling

The second option for creating a bus list is to use Message Scheduling. This is a new option starting with BusTools/1553-API version 6.20 (along with F/W version 4.40). Message Scheduling allows the user to schedule a message into different frames by specifying a start frame and repeat rate for each BC message. You can get the same frames as in the example above by scheduling messages into the first, second and third frames.

With fixed frame messaging, the application must allocate and write each message into all the minor frame buffers in which the message transacts. Each frame must contain all the messages specified for that frame. Message scheduling eliminates this redundancy by scheduling a message to transact at a scheduled rate.

## 6.2 BC Message Block Types

A BC Message Block defines each message within a Minor Frame. Each BC Message Block contains a control word defining the purpose of the BC Message Block. Valid options are 1553 message, Conditional Branch, Stop BC, Timed-No-op, or No-op.

Message blocks refer to the entire BC message data. This is comprised of the control block and the data block. There are two different formats for the control and data structure. Starting with F/W version 6.0 and BusTools/1553-API v8.00 the structure for the control block and data block changed to accommodate multiple BC buffers. The "MIL-STD-1553 UCA Reference Manual" defines the legacy structures. The Enhanced Universal Core Architecture Local Processing Unit (LPU) manual "UCA32 LPU Reference Manual" describes the new structures.

### 6.2.1 1553 BC Message Block

A 1553 message block contains the 1553 command word; message status words returned from the addressed RT, a 32-bit interrupt status word, pointers to the data buffers, time-tag, Error Injection Buffer, message number of the next BC Message Block, and the gap time.

If this is the first message in the minor frame, the Firmware does not execute the gap time delay before starting the message. The message starts on the minor frame time mark.

When the 1553 message executes, the contents of the selected data buffer are transmitted to the RT via BC->RT message, and filled with receive data on RT->BC messages. Status words returned from the addressed RTs are stored in the BC Message block.

After each message executes, the 32-bit Interrupt Status word is stored in the BC Message Block. The microcode then checks these Interrupt Status Bits against the BC Interrupt Enable Bits. If there are any matching bits (if the logical AND of the two 32-bit values is non-zero), an interrupt block is added to the Interrupt Queue and an Interrupt Message is sent to the thread assigned to handle BC interrupts. It is up to the application to ensure that the appropriate information is retrieved from the BC Message Block before that specific message is re-executed by the microcode (if the BC has been set up to “loop”).

In F/W version 5.x and earlier each BC Message Block can specify either one or two data buffers for the message. If you select only one buffer, that buffer is used each time this message executes. Use this setting to increase the number of BC messages that fit within the memory available on the 1553 board.

If you select two buffers, the “Buffer A” bit in the message specifies the buffer used. If the “Buffer A” bit is set, the messages use the first buffer; if it is clear, the second data buffer “Buffer B” is used. This allows the software to update the current unused data buffer, and then switch the BC to use that buffer. When you select the message-scheduling mode using F/W v5.0 or earlier only one buffer can be used.

Starting with F/W version 6.0 you can specify multiple BC data buffers. Multiple buffers are specified by adding the value MULTIPLE_BC_BUFFERS into the BC options parameter when you initialize the BC. There is a different method for allocating and programming multiple buffers, as described in the BC Software Operation section that follows in this document.

### 6.2.2 Conditional Branch BC Message Block

Conditional Branch messages allow for altering the bus list based on the value of data located within board memory, including data from a previous command. Conditional branching uses a data word address, an expected value, and a test mask to determine if the BC should branch. The BC compares the data to the expected value using the mask. If the comparison is true (equal), execution of the bus list transfers to the branch message. If the comparison is false (unequal), execution transfers to the address of the next message.

The Conditional Branch BC Message puts no traffic on the MIL-STD-1553 bus. Conditional Branch BC Message Blocks are executed by the microcode as quickly as possible. In many cases, the execution time is overlapped with the execution of the previous Message Block, resulting in an effective execution time of zero. The worst-case conditional branch delay is less than 25µs.

Using the parameters specified in the Conditional Branch Message Block, the microcode tests a data word and changes the order of message execution based on the results of the test. Select any location in the respective buffer memory for the data word. This means you can select the command, data, or status of a recently transacted 1553 message or a user-defined location in memory. This feature allows an application to reconfigure the bus list very quickly by changing only one word in buffer memory.

Starting with BusTools/1553-API v8.12 and firmware version 6.03 the firmware records the address of the message executed after the branch in the BC Message buffer.

### 6.2.3 Stop BC Message Block

The Stop BC Message Block clears the BC-run and BC-busy bits in the 1553 control register, terminating the execution of the Bus Controller. This block also contains a time-tag corresponding to the BC stop time.

### 6.2.4 NO-OP BC Message Block

The NO-OP BC Message Block skips to the next BC Message Block. It contains the address of the next BC Message Block to execute. A 1553 BC Message Block converts into a NO-OP message by clearing one bit in the control word and then converts back into a 1553 message by setting the bit.

Like the Conditional Branch BC Message Block, execution of the NO-OP Message Block is overlapped, as much as possible, with the previous message. This typically results in an effective execution time of zero.

### 6.2.5 Timed NO-OP BC Message Block

The Timed NO-OP BC Message Block skips to the next BC Message Block while inserting a timing delay between the two messages. It contains the address of the next BC Message Block to execute and a gap time. The timed NO-OP uses the gap-time so the user can insert delay between messages to extend the gap time beyond the maximum 16-bit delay. A 1553 BC Message Block converts into a Timed NO-OP message by clearing one bit and setting another in the control word and then converts back into a 1553 message by setting/clearing those two bits, respectively. A Timed NO-OP Message Block can also trigger a BC interrupt and related processing.

### 6.2.6 Timed NOOP in Different BC Timing Modes

For all V6.x firmware, there are two types of a Timed NOOP (TNOP): static Timed NOOP and dynamic Timed NOOP, and they have different usage rules; care must be taken when coding a TNOP. A static TNOP is configured by programming BC_CONTROL_TIMED_NOP for a message write operation; a dynamic TNOP is configured via BusTools_BC_MessageNoop invocation by setting the NoopFlag to TIMED_NOOP.

This table shows the BC timing modes where you can use the static TNOP and dynamic TNOP.

Table 6-1 Allowed BC Timing Modes for Static TNOP and Dynamic TNOP

| | Relative Time (default) | Fixed Gap Timing | Frame Start Timing |
|---|---|---|---|
| Static TNOP | Yes | Yes | No |
| Dynamic TNOP | No | Yes | No |

In the relative timing mode, the programmed gap time is measured from the end of the message to the start of the next message.  Dynamically disabling the message with TIMED_NOOP set would produce a much shorter time delay than was intended.

In frame start timing mode, the time relative to the start of frame is already programmed into the gap time word, so there is no need for a TNOP; and, if programmed incorrectly, could alter the bus list.

A TNOP should probably not be used with the message scheduling option (available with all three timing modes); but if it is, the user must perform a thorough analysis for each unique frame in order to obtain the desired results.

## 6.3 Aperiodic 1553 BC Messages

Normally, the Bus Controller executes a pre-defined list of messages (called a Bus List or a Major Frame) organized into sets of messages called Minor Frames. Each Minor Frame is executed sequentially, beginning on a time mark called the Minor Frame Rate.

Aperiodic messages are 1553 messages that the application designates to be inserted into a running bus list on a one-shot basis. An application can create a list of BC messages, consisting of one or more messages, limited only by available memory. These messages transact as soon as the Bus Controller can send them. You can send any type of 1553 message as an aperiodic. However, you cannot use the retry feature with aperiodic messages. The Application inserts the aperiodic messages by calling BusTools_BC_AperiodicRun.

Abaco Systems 1553 boards support two types of aperiodic message queues:

- High Priority
- Low Priority

You can have messages in both aperiodic message queue types at the same time.

### 6.3.1 High Priority Aperiodic Messages

An application creates a list of BC messages (consisting of one or more messages, limited only by available memory) and passes the address of the first message to the High Priority Aperiodic Message Queue.

Messages in the High Priority Aperiodic Message Queue are transmitted in sequence following completion of the currently transmitting periodic message. On completion of transmitting the aperiodic messages, the BC returns to processing the remaining Minor Frame periodic message list. If the transmission of messages in the High Priority Aperiodic Message Queue causes the minor frame to overflow, the BC function will wait for the beginning of the next minor frame to transmit the message(s).

CAUTION
Inserting high priority messages into a bus list may cause a minor frame overflow. If there is not enough time to transmit the remaining periodic messages in the bus list, a minor frame error will occur, transmission of the current minor frame will terminate, and transmission of the next minor frame will begin. The application must ensure that the time required to process the aperiodic messages does not cause the minor frame period to be exceeded.

### 6.3.2 Low-Priority Aperiodic Messages

An application creates a list of BC messages (consisting of one or more messages) and passes the address of the first message to the Low Priority Aperiodic Message Queue.

Messages in the Low Priority Aperiodic Message Queue are transmitted in the time period between transmission of the last message in a minor frame and the start of the next minor frame. Before transmitting a low priority aperiodic message, the BC determines if there is time to transmit the message before the start of the next minor frame. If enough time is available, the BC will transmit the aperiodic message; otherwise, the BC suspends processing low priority aperiodic messages until the end of the last message in the next minor frame. It repeats this process until transmission of all the messages in a low priority message list is complete.

NOTE
Low-priority messages are only transmitted if enough time is available at the end of the minor frame. It is possible to build a bus list that does not allow time for any low priority message to be transmitted. The application is required to ensure minor frame timing provides enough time to transmit messages in the Low Priority Aperiodic Message Queue.

The firmware calculates the time an aperiodic message takes to execute by counting the number of words and multiplying by 25µs and compares that with minor frame timer.

### 6.3.3 Aperiodic Message Timing

The 1553 command word and the rt31_bcst and sa31_mc bits, (see the functions BusTools_SetBroadcast and BusTools_SetSa31) determine an approximate duration (in µs) for transmitting a message. See below.

- **Normal messages**
  The time computed is 125µs + 25µs/data word.
- **Mode codes**
  The time computed is either 125µs or 150µs (without/with data word).
- **Broadcast messages**
  The time is computed as 300µs + 25µs/data word.

Error injection is ignored when computing the duration to transmit a message. It is possible to overflow the minor frame when using high word errors or programmable response times in either the aperiodic or periodic messages. The following message attributes are not added to the computed duration to transmit:

- 20µs for high word Error Injection
- 20µs for maximum programmable response time of 32µs
- RT-RT adds 20µs for programmable response time #2

Take care when setting up both the bus list and the aperiodic message list. Ensure that enough time is available in the programmed minor frames time to support the addition of aperiodic messages. Also, ensure the bus list and aperiodic messages account for any additional error injection timing.

## 6.4 Dynamic Bus Control

BusTools/1553-API supports Dynamic Bus Control (DBC) to allow the Bus Controller to transfer Bus Controller functions to a Remote Terminal. To perform DBC the Bus Controller issues the Dynamic Bus Control mode code (0000) and checks the DBC Acceptance bit in the returned status word. If that bit is set, the BC must immediately stop.

If the bit is reset, the Bus Control must continue running. Set up the Bus Controller for this operation using the standard API calls. See Section 5.6, "Dynamic Bus Control" for more details.

## 6.5 Message Scheduling

Message Scheduling allows the application to specify a start-frame and a repeat-rate for each message.

Scheduling is set up by setting MEG_SCHD when initializing the Bus Controller and programming a start-frame and a repeat rate. The table below shows the scheduling options.

Table 6-2 Message Scheduling

| Start Frame | Repeat Rate | Message Schedule |
|---|---|---|
| 0 | 0 | Never |
| 0 | n | Never |
| n | 0 | Once in the nth frame |
| 1 | 1 | Every frame |
| n | 1 | Every frame starting in the nth frame |
| n | n | Every nth frame starting in the nth frame |

With message scheduling, there is only a single instance of a message that is scheduled to transact in the defined minor frames.

## 6.6 Minor Frame Definition

A minor frame is a group of 1553 messages sent at a multiple of the frame rate you program for the Bus Controller.

Defining the beginning and ending of the minor frame are the key elements in configuring a minor frame. Adding the value BC_CONTROL_MFRAME_BEG to the control word of the first message in the minor frame in conjunction with adding the value BC_CONTROL_MFRAME_END to the control word of the last message in the minor frame will define the minor frame. You can have any number and type of messages in a minor frame. If the minor frame consists of a single message, then the values BC_CONTROL_MFRAME_BEG and BC_CONTROL_MFRAME_END should both be added to the control word of that message.

You can have any number of minor frames. Start the message number for the first message in the first minor frame at 0. Increment the message count by one for each additional message, no matter how many minor frames you specify.

The BC transmits messages on the 1553 bus according to the message list in the minor frames you program. The message list you create tells the BC how to transmit the messages.

Next, determine if the BC will transmit the messages once or continually. If you want to send the bus list only once, the control word of the last message in the list is BC_CONTROL_LAST. This stops the Bus Controller. If you want to send the messages continually, set the "next-message pointer" of the last message in the bus list to 0. This creates a circular linked list of messages that continually run according to the bus list parameters you program. Start and stop the BC by calling BusTools_BC_StartStop.

The next thing you must decide is the number of minor frames. A minor frame is a group of 1553 messages sent at a multiple of the frame rate you program for the Bus Controller. Program the frame rate in the call the BusTools_BC_Init.

If you have one minor frame, the BC starts the messages in that frame at the frame rate. For example, a 1-s frame rate causes the BC to send the frame every second. If you have more than one minor frame, each frame starts on the multiple of the frame rate. If there are n minor frames, each minor frame runs at *n* times the frame rate (n x frame rate).

A minor frame can have from one message to the number of messages that will fit within the frame rate. If the time to transact the messages in a minor frame exceeds the frame rate, you get a minor-frame-overflow error. Remember the transaction time for the frame includes gap time and response times. To calculate the time used for a message, count 20μs for each word (command, status, and data) plus 7μs for average response time, plus the programmed gap time. For example, a one-word BC-RT message with a 15-μs gap will take approximately 60 + 7 + 15 = 82μs to transact.

There are two options for defining a minor frame. When initializing the Bus Controller, you can select to use frame messaging or message scheduling. Frame messaging is more tedious and requires multiple instances of the same messages that repeat in different frames, but it does allow a more complex bus list. Message Scheduling is new starting with BusTools/1553-API version 6.20. This method schedules messages to transmit in various frames by setting the start frame and repeat rate parameters.

The BC transmits messages on the 1553 bus according to the bus list you program. The bus list you create tells the BC how to transmit the messages. Control the bus (A or B), message type, gap time, and other message data by setting parameters in the API_BC_MBUF structure you pass to BusTools_BC_MessageWrite.

Next, determine if the BC will transmit the messages once or continually. If you want to send the bus list only once, the control word of the last message in the list is BC_CONTROL_LAST. This stops the Bus Controller. If you want to send the messages continually, set the "next-message pointer" of the last message in the bus list to 0. This creates a circular linked list of messages that continually run according to the bus list parameters you program. Start and stop the BC by calling BusTools_BC_StartStop.

The next thing you must decide is the number of minor frames. A minor frame is a group of 1553 messages sent at a multiple of the frame rate you program for the Bus Controller. Program the frame rate in the call the BusTools_BC_Init.

If you have one minor frame, the BC starts the messages in that frame at the frame rate. For example, a 1-s frame rate causes the BC to send the frame every second. If you have more than one minor frame, each frame starts on the multiple of the frame rate. If there are n minor frames, each minor frame runs at n times the frame rate (n x frame rate).

The number of messages allowed in a minor frame ranges from one message to the number of messages whose duration to transmit will fit within the defined minor frame rate. If the time to transact the messages in a minor frame exceeds the frame rate, a minor-frame-overflow error will be encountered. Remember the transaction time for the frame includes gap time and response times. To calculate the time used for a message, add 20µs for each word (command, status, and data) plus 7µs for average response time, plus the programmed gap time. For example, a one-word BC-RT message with a 15-µs gap will take approximately 60 + 7 + 15 = 82µs to transact.

Define BC minor frames using the BC_CONTROL_MFRAME_BEG and BC_CONTROL_MFRAME_END bits in the “control” word of the BC message structure. The first Message Block of each minor frame must be marked with the BC_CONTROL_MFRAME_BEG bit. The last Message Block of each minor frame must be marked with the BC_CONTROL_MFRAME_END bit.

As you define each message, specify the next message in the list to execute (typically, this is the next message in the list). The hardware expects to see a channel memory address for this entry. However, since the BusTools/1553-API handles all memory addressing requirements, the application need only refer to messages by number. Message #0 is the first message in the BC Message block list. Once the Message Blocks are filled in, start the BC simulation using the BusTools_BC_StartStop routine.

## 6.7 Message Gap Timing

Message gap time is the time interval between MIL-STD-1553 messages in a minor-frame. The application defines this value during the Bus Controller message setup. The valid range for message gap time value is 4µs to 16777216 µs, (upper limit is 65535 µs for firmware versions prior to Ver. 4.40).

When initializing the Bus Controller, three options are available for specifying message gap time. They are relative gap timing (default), fixed gap timing, and frame start timing.

Relative Gap timing applies the programmed gap-time to the completion of a message to determine when the next message will be processed. Completion of the transaction includes the data transmission, response time, and possible no-response from a Remote Terminal. The start of the next message is always relative to the end of previous message and response (or no response), and the gap duration inserted between two sequential messages in a frame will always be the value of the gap time programmed.

Fixed Gap timing applies the gap-time relative to the start of the current message. Transmission of the next message is always a fixed time from the start of the current message, regardless of any instance of no response or retry. The gap time must consider the duration of a complete message transaction, including all possible retries. If a message transaction is still active when the gap time expires, frame timing will be corrupted.

Frame Start timing applies the gap-time relative to the start of the current frame. With this option all messages will have a defined start time within the frame, including an optional delayed transmission of the first message in the frame. When using frame start timing the application must account for the duration of all message transactions defined within in the respective minor frame.

Any one of these three gap timing options can be used with either Frame Messaging or Message Scheduling.

## 6.8 BC Software Operation

This section describes the BusTools/1553-API routines that control BC operations. The "BusTools/1553-API Reference Manual" has detailed syntax, structure layout, and error status information for these routines.

### 6.8.1 Initializing the Bus Controller

Prior to calling any BusTools/1553-API BC routine, you must initialize the Bus Controller operation. An invocation of BusTools_BC_Init Init defines the attributes for the BC function on this channel, including the Interrupt Enable Bits for all BC messages, the frame rate (Minor Frame period), the "no-response" and "late-response" time-out periods, the hardware retry conditions, the BC gap timing mode (Relative Gap, Fixed Gap, or Frame Start Gap), scheduling method, and single/multiple BC buffer selection. This routine should only be invoked after initializing the channel via invocation of BusTools_API_InitExtended or BusTools_API_OpenChannel.

The default BC operation uses relative gap timing without message scheduling or multiple BC Buffers. See the "BusTools 1553-API Reference Manual" for details.

Define the BC Interrupts by ORing the desired interrupt option bits from Table 8-1 Interrupt Events". The BC interrupt enable register is a 32-bit register with each individual bit programming a different interrupt condition. Review the example code in the Example directory for Bus Controller coding examples. The ability to select which Bus Controller message generates interrupts is also available. Interrupts can be programmed on a message-by-message basis by ORing the value BC_CONTROL_INTERRUPT into the message control word.

Calling the BC initialization allows you to set retry conditions. You can have the BC retry a 1553 command if the firmware detects any of the conditions below.

| | |
|---|---|
| • BC_RETRY_NRSP | No Response |
| • BC_RETRY_ME | Message Error |
| • BC_RETRY_BUSY | Busy Bit Set |
| • BC_RETRY_TF | Terminal Flag |
| • BC_RETRY_SSF | Subsystem Flag |
| • BC_RETRY_INSTR | Instrumentation |
| • BC_RETRY_SRQ | Service Request |
| • BC_RETRY_INV_WRD | Invalid Word[1] |
| • BC_RETRY_INV_SYNC | Sync error[2] |
| • BC_RETRY_MID_BIT | Mid-bit error[2] |
| • BC_RETRY_TWO_BUS | Two bus error[2] |
| • BC_RETRY_PARITY | Parity Error[2] |
| • BC_RETRY_CONT_DATA | Non-contiguous data[2] |
| • BC_RETRY_EARLY_RSP | Early response |
| • BC_RETRY_LATE_RSP | Late response |
| • BC_RETRY_BAD_ADDR | Bad RT address |
| • BC_RETRY_WRONG_BUS | Wrong Bus[3] |
| • BC_RETRY_NO_GAP | No inter-message gap |

[1] for F/W version 6.10 use BC_RETRY_INV_WRD in place of sync, parity, two bus, mid bit and bit count errors
[2] Retries on these conditions are not supported in F/W v6.10
[3] Not supported in F/W 5.0 and greater

Program retries on a message-by-message basis when defining the 1553 message by ORing the value BC_CONTROL_RETRY into the control word. Invoke BusTools_BC_RetryInit to define how a retry executes.

The call to BusTools_BC_Init also allows the application to define time-out values for both late response and no-response. The late response period is the duration between completion of a BC message transmission and a future point in time in which the corresponding RT response is considered late. If an RT response is received in the period between the late response time-out and the no-response time-out, it will be designated as late and the corresponding Late Response bit will be set in the Interrupt Status Word. If the duration of an RT response exceeds the no-response time-out, the corresponding No Response bit in the Interrupt Status Word is set. When programming these two time-out values, an application should consider the response capabilities of the RTs in the target system.

The Frame Rate value is the BC initialization of the minor frame time. For example, selecting 100,000 as the frame rate sets a 10 Hz rate. This means a minor frame transacts every 100,000µs. If there are 3 minor frames in the bus list, then it will take 300,000µs to transact all three frames.

For BC function code demonstrations, review the coding examples in the BusTools/1553-API software distribution Examples directory.

### 6.8.2 Allocating BC Messages

The next step in setting up the BC is to allocate and initialize the list of BC Message Blocks. BusTools_BC_MessageAlloc provides the method to allocation BC Message Blocks. When invoking BusTools_BC_MessageAlloc the application should define the number of BC messages required in the minor frames for all transaction scenarios, including additional aperiodic messages. Allocating extra message blocks that are not used is recommended if spare channel memory is available. As an example, if the application scenario requires 3 minor-frames with 10 messages in each, 50 messages may be allocated without affecting BC operations. However, if memory for only 30 messages is allocated and an additional aperiodic message is added, an API_BC_ILLEGAL_MBLOCK error will be encountered when the application attempts to write to that message block.

Applications using legacy products with older firmware and API versions are limited in message buffer allocation to one buffer, or two buffers switched under application control. Starting with firmware version 6.0 and BusTools/1553-API version 8.00 the Bus Controller can have multiple BC Data Buffers allocated to individual messages, specified in the invocation of BusTools_BC_Init. This feature allows an application to allocate any number of buffers for each message, limited only by memory available on the channel. To program multiple BC buffer usage in the application, invoke the function BusTools_BC_MessageBlockAlloc for each BC message buffer created.

BusTools/1553-API 8.0 and greater allows either BC Message allocation method. Existing applications using the legacy method will run the same.

### 6.8.3 BC Messages

An application should define all required BC Message Blocks via invocation of BusTools_BC_MessageWrite; however, unused allocated message blocks do not require initial values. Each message is referenced via message index or message number, zero referenced and tracked by the application based on the number of messages provided in the invocation of BusTools_BC_MessageAlloc. If an application defines 3 minor frames with 10 messages each, then 30 messages referenced via message number 0 through 29 should be initialized.

If you are using Multiple BC Buffers, then use BusTools_BC_MessageWrite to write the Message definition, and data for the message and first data buffer. Then use BusTools_BC_DataBufferWrite to fill in the additional data buffers specified in BusTools_BC_MessageBufferAlloc.

### 6.8.4 Defining a Minor Frame

Minor frames can be configured for transmission via one of two schemes, frame messaging or message scheduling. The Message Scheduling method schedules messages to run in various frames by setting the start frame and repeat rate parameters, while Frame Messaging requires multiple instances of the same messages repeated in different frames. Refer to Section 6.1 "BC Hardware Operation" for details on scheduling message activity in Minor Frames.

### 6.8.5 Setup Message Gap Timing

There are three options for specifying gap time. They are relative gap timing (default), fixed gap timing, and frame start timing. The three gap-time setting is used with either Frame Messaging or Message Scheduling. Refer to Section 6.7, "Message Gap Timing" for details.

### 6.8.6 Bus Controller Interrupt Programming

The firmware checks the status of each BC message to decide if it should generate an interrupt. If the firmware detects the conditions specified for an interrupt, the firmware writes an entry to the interrupt queue and asserts the hardware interrupt (IRQ) line (if enabled).

The BusTools/1553-API provides a high-level method for processing interrupts, BusTools_RegisterFunction. This function uses multi-threaded processing. BusTools_RegisterFunction is available for Windows, Linux, LynxOS, VxWorks and Integrity. Windows uses Windows threads while all other systems use POSIX threads.

## 6.9 BC One-Shot Operation

The Bus Controller can execute a bus list continuously or as a single pass. The single pass execution is referred to as "One Shot Mode". Previous versions of the API supported "One Shot" functions to facilitate programming this type of operation. These functions are no longer available. Instead, the user must set up the Bus Controller using BC_CONTROL_LAST to stop the list after a single pass.

## 6.10 BC Read/Write/Update/Allocate APIs

BusTools/1553-API provides several API functions for its basic Bus Controller read, write, update and message allocation operations. This section lists all applicable API functions for the respective firmware designs.

Table 6-3 lists the API functions for firmware version 4/5 single/double data buffer functionality.

Table 6-4 lists the API functions for firmware version 6 single/double data buffer functionality.

Table 6-5 provides the API functions for the multiple buffer feature introduced with the combination of firmware version 6 and API version 8.0.

Table 6-3 Single/Double Data Buffer Functionality with F/W V4/5

| BusTools/1553-API Function | Descriptions |
|---|---|
| BusTools_BC_MessageReadDataBuffer ( wCardnum, wMessageid, wBufferid, pBuffer ); | This routine reads the specified data buffer from the specified BC message from the 1553 board. If BusTools_BC_Init is called with num_buffers equal to 2, this function reads the data from the selected buffer. If num_buffers is equal to 1, this routine reads a single BC data buffer. |
| BusTools_BC_MessageUpdate( wCardnum, wMblock_id, pBuffer ) | If BusTools_BC_Init is called with num_buffers equal to 2, this function writes the new data to the inactive buffer. If num_buffers is equal to 1, this routine updates the single BC data buffer. The new buffer becomes active after the message buffer control-word has been written. The first of the two buffers is active after initialization. |
| BusTools_BC_MessageUpdateBuffer( wCardnum, wMblock_id, wBufferid, pBuffer ) | If BusTools_BC_Init was called with num_buffers equal to 2, this function writes the new data to the buffer selected by wBufferid. If num_buffers is equal to 1, this routine updates the single BC data buffer regardless of the value of wBufferid. |
| BusTools_BC_MessageAlloc( wCardnum, wCount ) | This function allocates the message buffers from 0 to wCount-1 and clears the allocated buffers. |
| BusTools_BC_MessageRead ( wCardnum, wMessageid, pMessage ) | This routine reads the specified BC message from the Abaco Systems 1553 device |
| BusTools_BC_MessageWrite( wCardnum, wMessageid, pMessage ); | This routine writes a Bus Controller message or control structure to a specified Bus Controller message buffer on the 1553 board. This function is used to fill in the message parameters and data in the first buffer when the double data buffers feature is selected. |

Table 6-4 Single/Double Data Buffer Functionality with F/W V6

| BusTools/1553-API Function | Descriptions |
|---|---|
| BusTools_BC_MessageBufferRead ( wCardnum, wAddr, pMessage ) | This function is used in the user callback function specified in the API_INT_FIFO structure passed in a call to BusTools_RegisterFunction. Use this function in a user callback function setup to read BC messages. The address of that buffer is stored in the API_INT_FIFO entry fifo[index].buff_off. Use the fifo[index].buffer_off for wAddr. |
| BusTools_BC_ReadDataBuffer ( wCardnum, wBufAddr, pDbuffer); | This function uses the buffer address stored in the interrupt queue to read the data recorded in that buffer. This function only reads the data portion of the Bus Controller data buffer. |
| BusTools_BC_DataBufferUpdate ( wCardnum, wBufaddr, wDcount, pwBuffer ) | This function is used in the user callback function specified in the API_INT_FIFO structure passed in a call to BusTools_RegisterFunction. The interrupt queue contains the address of the data buffer that generated the interrupt. The address of that buffer is stored in the API_INT_FIFO entry fifo[index].buff_off. Use this address as the wBufaddr. |

| BusTools/1553-API Function | Descriptions |
|---|---|
| BusTools_BC_MessageAlloc( wCardnum, wCount ) | This function allocates the message buffers from 0 to wCount-1 and clears the allocated buffers. |
| BusTools_BC_MessageRead ( wCardnum, wMessageid, pMessage ) | This routine reads the specified BC message from the Abaco Systems 1553 device. |
| BusTools_BC_MessageReadData ( wCardnum, wMessageid, pBuffer ); | This routine reads the data buffer of the specified BC message from the 1553 board. The value of the A/B buffer bit in the BC Message Buffer Control Word determines which data buffer to update. If BusTools_BC_Init is called with num_buffers equal to 2, this function reads the data from the active buffer. If num_buffers is equal to 1, this routine reads a single BC data buffer. |
| BusTools_BC_MessageReadDataBuffer ( wCardnum, wMessageid, wBufferid, pBuffer ); | This routine reads the specified data buffer from the specified BC message from the 1553 board. If BusTools_BC_Init is called with num_buffers equal to 2, this function reads the data from the selected buffer. |
| BusTools_BC_MessageUpdate( wCardnum, wMblock_id, pBuffer ) | If BusTools_BC_Init is called with num_buffers equal to 2, this function writes the new data to the inactive buffer. If num_buffers is equal to 1, this routine updates the single BC data buffer. The new buffer becomes active after the message buffer control-word has been written. The first of the two buffers is active after initialization. |
| BusTools_BC_MessageUpdateBuffer( wCardnum, wMblock_id, wBufferid, pBuffer ) | If BusTools_BC_Init is called with num_buffers equal to 2, this function writes the new data to the buffer selected by wBufferid. If num_buffers is equal to 1, this routine updates the single BC data buffer regardless of the value of wBufferid. |
| BusTools_BC_MessageWrite( wCardnum, wMessageid, pMessage ); | This routine writes a Bus Controller message or control structure to a specified Bus Controller message buffer on the 1553 board. This function is used to fill in the message parameters and data in the first buffer when the double data buffers feature is selected. |

Table 6-5 Multiple Data Buffer Functionality with F/W V6 and API 8.0+

| BusTools/1553-API Function | Descriptions |
|---|---|
| BusTools_BC_MessageBufferRead ( wCardnum, wAddr, pMessage ) | This function is used in the user callback function specified in the API_INT_FIFO structure passed in a call to BusTools_RegisterFunction. Use this function in a user callback function setup to read BC messages. The address of that buffer is stored in the API_INT_FIFO entry fifo[index].buff_off. Use the fifo[index].buffer_off for wAddr. |
| BusTools_BC_ReadDataBuffer ( wCardnum, wBufAddr, pDbuffer); | This function uses the buffer address stored in the interrupt queue to read the data recorded in that buffer. This function only reads the data portion of the Bus Controller data buffer. |
| BusTools_BC_DataBufferUpdate ( wCardnum, wBufaddr, wDcount, pwBuffer ) | This function is used in the user callback function specified in the API_INT_FIFO structure passed in a call to BusTools_RegisterFunction. The address of that buffer is stored in the API_INT_FIFO entry fifo[index].buff_off. Use this address as the wBufaddr. |
| BusTools_BC_DataBufferWrite ( wCardnum, wMsgId, wBufId, pwBuffer ) | This function is used with Multiple BC Data Buffers to populate data into the respective data buffer accessed by message ID and Buffer ID. |
| BusTools_BC_MessageBlockAlloc( wCardnum, wBufid, wCount ) | This function creates a BC message control block and the number of associated data buffers as indicated in the count parameter. At least 1 data buffer must be allocated, with a maximum data buffer count limited only be the amount of available memory. You must call this routine for each BC message buffer. You must complete allocating all BC message blocks either before or after creating the Bus Monitor buffers. You cannot allocate additional buffers after allocating the Bus Monitor buffer. If you are using aperiodic messages, make sure you allocate enough messages for the periodic bus list and the aperiodic bus list, if used, since both reside in channel memory. |
| BusTools_BC_MessageRead ( wCardnum, wMessageid, pMessage ) | This function reads the specified BC message from the 1553 device |

| BusTools/1553-API Function | Descriptions |
|---|---|
| BusTools_BC_MessageWrite( wCardnum, wMessageid,  pMessage ) | This routine writes a Bus Controller message or control structure to a specified Bus Controller message buffer on the 1553 board. This function is used to fill in the message parameters and data in the first buffer when multiple data buffers are allocated. |
| BusTools_BC_SelectBufferRead( wCardnum, wMessno, wBufno, pDbuffer) | This function uses the message number and the buffer number to read the data recorded in that buffer. This function only reads the data portion of the Bus Controller data buffer. |
| wStatus = BusTools_BC_SelectBufferUpdate( wCardnum, wMessno, wBufno, wCount, pDbuffer) | This function uses the message number and the buffer number to update the data in the buffer. |

# 7 • Error Injection

The Error Injection (EI) function on the Abaco Systems MIL-STD-1553 boards creates specific errors in the 1553 data stream, allowing you to test how other equipment responds to these errors. This chapter describes the EI functions.

## 7.1 Error Injection Hardware Operation

You can inject several different errors into the 1553 data stream. Some errors affect individual words within the message, while others affect the entire message. Some errors result in erroneous behavior by the RT. In all cases, errors are specified using the Error Injection Buffer associated with a specific operation. The API defines a default Error Injection Buffer with no errors. The API uses this buffer for all BC and RT message buffers without errors (this is API error injection buffer “0”).

Each word of an Error Injection Buffer controls the error(s) in a specific 1553 transmit-word. For each 1553 message, there are up to three Error Injection Buffers. If the board is simulating the BC, there is an Error Injection Buffer associated with the BC Message Buffer. If the board is simulating a specific RT, there is an Error Injection Buffer associated with the RT Message Buffer. Finally, for situations involving RT-to-RT messages between two simulated RTs on the board, each has its own Error Injection Buffer. BC and RT message buffers are described in the “BusTools 1553-API Reference Manual”.

The maximum size of an Error Injection Buffer is 34 words. The actual size depends on the Error Injection Buffer type. There are five types of Error Injection Buffers:

- **BC Receive Error**: Error Injection Buffer for errors in a BC Receive Message. This buffer contains 34 error specification words. There is one error specification word for the message command word followed by 33 error specification words for message data words. The API uses 33 words for a High Word Count Error.
- **BC Transmit Error**: Error Injection Buffer for BC Transmit Messages. This buffer contains only one error specification word for the message command word.
- **BC RT-to-RT Error**: Error Injection Buffer for BC RT-to-RT Messages. This buffer contains two error specification words, one for each message command word.
- **RT Receive Error**: Error Injection Buffer for RT Receive Messages. This buffer contains one error specification word that applies to the message status word.
- **RT Transmit Error**: Error Injection Buffer for RT Transmit Messages. This buffer contains up to 34 error specification words. There is one error specification word for the message status word and up to 33 error specification words for message data words. The API uses 33 words for a High Word Count Error.

## 7.2 Error Types

The following section describes the possible errors you can inject into the 1553 data stream. The board checks the current position in the Error Injection Buffer to see if you specified any error conditions for that word. If there are, the firmware adds those errors to the word during transmission. The firmware injects errors on a word-by-word basis in a command word, status word, or data word, as it is transmitted on the bus.

The following errors are set using either **BusTools_EI_EnhEbufWrite** or **BusTools_EI_EbufWrite**.

- **Bit Count Error**: The hardware transmits a sync pulse, 16 data bits and a parity bit for each 1553 word. Setting this error allows you to alter the number of data bits in a word. The RT Validation Test Plan recommends using values in the range 14 to 19. Values outside of this range may cause other parts of the hardware to function improperly. Set this error for any entry in any EI buffer type. In Bit Count errors, the parity bit counts as a bit. Thus, injecting 17 bits may produce a parity error, not a bit count error.
- **Word Count Error**: Normally, the command word specifies the number of words in a message. This error causes the actual number of words in the message to differ from that in the command word. You can set any word count from one to 33. You must specify this error condition on the command word of a BC message, for BC Receive buffer types, or the status word of a RT message, for RT Transmit buffer types.
- **Sync Error**: Use this error to transmit a word with an invalid sync code. The data pattern specified in the error injection buffer indicates the shape of the sync code. A data pattern of zero (0) defaults to "inverted sync". This error can be specified for any entry in any EI buffer type.
- **Mid-Bit Mid-Parity and Mid-Sync**: These three error injection codes delay the zero-crossing point by 300ns from where it is expected. For mid-bit zero-crossing errors, you can select the bit (0 through 15). Bit 0 is the LSB of the 16-bit word, however it transmits last in the serial stream. Similarly, bit 15, the MSB, is transmitted first.
- **Programmable Response**: This RT selection allows the application to enter a programmable response time of up to 31½µs, programmed with the LSB equal to 500ns. Do not use an entry less than 4-µs as this is not supported and undefined operation may result. An entry greater than 14µs is a late response according to the MIL-STD-1553B Specification. This error applies only to the RT status word in either of the RT buffers.
- **Parity Error**: This error transmits a word with an inverted parity bit. You can specify this for any entry in any EI buffer type.
- **Data gap**: Add a gap time between data words from 0.5 to 2.5µs in .5-µs increments.

- **Respond with Wrong Address**: Normally, the RT status word contains the address of the RT. This error forces the status word to contain a different address. You can set the address to any number from 0 to 31. You can select this error only in RT Receive and RT Transmit Error Injection Buffer types.
- **Bi-Phase Error**: A bi-phase bit error is an error where there is no zero crossing for the entire bit time. It is not predictable how a 1553 decoder will interpret a word when this error is on one of the first two bits. This depends on timing and the states of the first two bits. When the bit does not cross zero, it may stay in one state for 1½µs and the decoder might try to reestablish sync. Therefore, the API does not support injection in either of the first two bits, but you should be aware of the possible behavior should this case occur.

The following errors are set only using BusTools_EI_EnhEbufWrite (Available in BusTools/1553-API v6.42 or later running with firmware version 5.00 or later.)

- **Bi-Phase Low Error**: The selected bit is at a logic LOW during the entire bit time.
- **Bi-Phase High Error**: The selected bit is at a logic HIGH during the entire bit time.
- **Bi-Phase Parity Low Error**: The error is injected into the parity bit. This bit will be at a logic LOW the entire bit time.
- **Bi-Phase Parity High Error**: The error is injected into the parity bit. This bit will be at a logic HIGH during the entire bit time.
- **Enhanced Zero-Crossing Error**: The error is inserted into a specified zero crossing transition in the specified word. The zero crossing may be programmed to occur before the expected transition time by setting the ZC_Early bit, or after the expected transition time by clearing the ZC_Early bit. ZC_Early is part of the Enhanced Zero-Crossing Error Inject word that (see below). The offset value has a resolution of 6.25ns, which allows for a 400ns offset using a 6-bit field.
- **T-Enhanced Zero-Crossing Error**: in some cases, the software must compensate the transition offset. For late offsets, the firmware must handle the compensation. This error code provides for firmware compensation.

## 7.3 Enhanced Zero-Crossing

Enhanced-zero crossing error shifts the zero-crossing transition up to +/- 400ns from the nominal crossing time. If you want to inject Enhanced Zero-Crossing Errors or T-Enhanced Zero-Crossing Errors, you must reference the “error_injection_word” paragraph in the Error Injection Buffer description of the UCA or UCA32 reference manual applicable to the firmware revision programmed on your board. This section explains how to setup the 16-bit enhData used in BusTools_EI_EnhEbufWrite.

## 7.4 Error Injection Software Operation

This section describes the BusTools/1553-API routines that handle Error Injection Buffers. Details of the syntax and error status information are in the "BusTools 1553-API Reference Manual".

When Bus Monitor operations are initialized using BusTools_BM_Init, the API initializes board memory with 64 Error Injection Buffers. The API identifies buffers by a number between 0 and 63. Initialization fills all buffers with "zeros", so the firmware does not inject errors for any Error Injection Buffer type.

The first buffer (buffer id #0) should be left in its initialized state, reserved as the "No Error" Error Injection Buffer. All BC Message Buffers and RT Message Buffers not affected by error injection should reference it.

The remaining buffers should be defined with the appropriate error injection information as required using BusTools_EI_EbufWrite. The caller must specify which Error Injection Buffer is to be accessed and a structure that indicates the buffer type and contents. The Error Injection structure API_EIBUF defines the elements of the error injection structure. It is up to the caller to ensure that each element of the structure matches the requirements of the Error Injection Buffer type.

# 8 • Interrupt Queue and Interrupts

The Interrupt Queue is the core of the flexible interrupt processing on all Abaco Systems' MIL-STD-1553 products. The Interrupt Queue is an on-board data structure maintained by the firmware. The firmware updates the interrupt queue with information about an event when it occurs. The interrupt queue buffers events, so the host system need not respond to them immediately.

BusTools/1553-API allows you to define which events go into the interrupt queue. It also allows you to enable the firmware to send a hardware interrupt to the host. The firmware records these events in the interrupt queue when they occur. You must select one of the hardware interrupt options for the firmware to propagate the interrupt to the host. If you do not use hardware interrupts, your application must poll the interrupt queue to process interrupt events. You can do this automatically with a "software interrupt" that checks the interrupt queue at a specified interval.

Control of interrupts for a channel is specified at initialization when you select the mode parameter (BusTools_API_OpenChannel) or flag parameter (BusTools_API_InitExtended). There are four general interrupt mode options:

| | |
|---|---|
| • API_SW_INTERRUPT (1) | – Software Interrupt mode |
| • API_HW_INTERRUPT (2) | – Software + Hardware interrupts |
| • API_HW_ONLY_INT (3) | – Hardware interrupt mode |
| • API_MANUAL_INT (0x20) | – No interrupt processing |

Use hardware or software interrupts in conjunction with BusTools_RegisterFunction to process the interrupt events. BusTools_RegisterFunction calls a user-supplied function when a specified event happens. The following sections describe setting up and using both polling and hardware interrupts.

## 8.1 Interrupt Queue Initialization and Structure

The interrupt queue consists of a linked list of 296, three-word entries. The first of these words is the interrupting mode (BC/BM/RT), the second word is the message buffer address, and the third word is a pointer to the next queue entry. The last entry in the list points back to the first entry, creating a circular linked list. The API initializes the "interrupt_queue_pointer" (a hardware register) to point to the first word of the queue.

Once an application has started bus traffic running (with calls to BusTools_BC_StartStop, BusTools_BM_StartStop, or BusTools_RT_StartStop), the firmware updates the interrupt_queue_pointer data recorded in the queue. Since the interrupt queue is a circular buffer, after the firmware fills the queue, it wraps to the start of the queue. Read data from the interrupt queue at a rate fast enough to avoid having data in the queue overwritten. This rate is application dependent. For example, if your application has 300 interrupt events per second, you must read the data from the interrupt queue faster than a 1 Hz rate.

BusTools_RegisterFunction reads the interrupt queue every 10ms in software interrupt mode and for every interrupt in hardware interrupt mode. You can call BusTools_SetPolling to modify the software interrupt polling rate.

## 8.2 Selecting Interrupt Events

BusTools/1553-API allows you to select the events that generate interrupts through API function calls. The Bus Controller, Bus Monitor, and Remote Terminal each have different methods for setting interrupt events, but they all use the "Interrupt Enable/ Message Status Bits structure" to select interrupt events.

The "Interrupt Enable / Message Status Bits structure" is a 32-bit, unsigned integer, with each bit representing an interrupt event. The section "Interrupt Enable/Message Status Bits" in the Data Structures section of the "BusTools/1553-API Reference Manual" describes this structure in detail. Table 8-1 shows the selectable interrupt events. Select the interrupt events by setting bits in the "Interrupt Enable word" as described in the following sections.

The firmware generates a 32-bit "Interrupt Status word" for each 1553 message and stores it in the "Interrupt status word" data element in the BC/BM/RT message buffer (BC – int_status; RT – status; BM – int_status). By testing the bits in this data element, you can see which interrupt events occurred. In addition to the interrupts in the table below, you can get an interrupt on external trigger. This interrupt is not linked to any message and does not have a bit in the interrupt status table.

Table 8-1 Interrupt Events

| Interrupt Name | Interrupt Value | Interrupt Description |
|---|---|---|
| BT1553_INT_HIGH_WORD | 0x00000001 | high word error |
| BT1553_INT_BIT_COUNT_DATA | 0x00000001 | bit count err, data word ** |
| BT1553_INT_INVALID_WORD | 0x00000002 | Invalid word error ** |
| BT1553_INT_LOW_WORD | 0x00000004 | low word error |
| BT1553_INT_INVERTED_SYNC | 0x00000008 | Inverted sync** |
| BT1553_INT_MID_BIT | 0x00000010 | Mid Bit Error** |
| BT1553_INT_TWO_BUS | 0x00000020 | data on both buses error |
| BT1553_INT_PARITY | 0x00000040 | parity error** |
| BT1553_INT_NON_CONT_DATA | 0x00000080 | non-contiguous data** |
| BT1553_INT_EARLY_RESP | 0x00000100 | early response |
| BT1553_INT_LATE_RESP | 0x00000200 | late response |
| BT1553_INT_BAD_RTADDR | 0x00000400 | incorrect rt address |
| BT1553_INT_CHANNEL | 0x00000800 | Bus (0=A, 1=B) |
| BT1553_INT_WRONG_BUS | 0x00002000 | Response on wrong bus |
| BT1553_INT_BIT_COUNT | 0x00004000 | bit count error** (Not used by F/W 5.00 or later) |
| BT1553_INT_NO_IMSG_GAP | 0x00008000 | No/Short inter-message gap |

| Interrupt Name | Interrupt Value | Interrupt Description |
|---|---|---|
| BT1553_INT_END_OF_MESS | 0x00010000 | End of message |
| BT1553_INT_BROADCAST | 0x00020000 | broadcast message |
| BT1553_INT_RT_RT_FORMAT | 0x00040000 | rt-to-rt message format |
| BT1553_INT_RESET_RT | 0x00080000 | Reset rt |
| BT1553_INT_SELF_TEST | 0x00100000 | Self-test |
| BT1553_INT_MODE_CODE | 0x00200000 | Message is a Mode Code |
| BT1553_INT_NOCMD | 0x00400000 | Command unseen by decoder |
| BT1553_INV_RTRT_TX | 0x00800000 | Invalid RTRT TX CMD2 |
| BT1553_INT_RTRT_RCV_NRSP | 0x01000000 | RT-RT No response on Rcv |
| BT1553_INT_RETRY | 0x02000000 | Retry |
| BT1553_INT_NO_RESP | 0x04000000 | no response (RT-RT, set if EITHER is no resp.) |
| BT1553_INT_ME_BIT | 0x08000000 | 1553 status word message error bit |
| BT1553_INT_TRIG_BEGIN | 0x10000000 | message with trigger begin |
| BT1553_INT_TRIG_END | 0x20000000 | message with trigger end |
| BT1553_INT_BM_OVERFLOW | 0x40000000 | message at buffer overflow |
| BT1553_INT_ALT_BUS | 0x80000000 | retry on alternate bus |

** While all events shown in this table apply to individual command words, status words or entire messages, only these interrupt events will be encountered for a fault in an individual message data word.

### 8.2.1 Selecting Bus Controller Interrupts

The API allows you to set the interrupt conditions for all Bus Controller messages. You can also select which Bus Controller messages generate an interrupt. Set the interrupt conditions by setting the appropriate bits in the "Interrupt Enable Word" passed in the call to BusTools_BC_Init. This is illustrated in the coding example for Bus Controller. The firmware generates a Bus Controller interrupt if the Interrupt Enable Word passed to BusTools_BC_Init is non-zero.

The coding example enables interrupts on the "end of message" (BT1553_INT_END_OF_MESS). The "end of message" event is always set on at the end of each valid 1553 message. If the "Interrupt Status Word" has this bit set, then the firmware completed processing the message.

To select the specific BC messages you want to interrupt, set the BC_CONTROL_INTERRUPT (0x0080) bit in the control word of the API_BC_MBUF structure passed to BusTools_BC_MessageWrite. There is also the option to have the message go into the interrupt queue but not generate a hardware interrupt. If you want the message recorded in the interrupt queue without getting hardware interrupts, then also set the BC_CONTROL_INTQ_ONLY. Use this, for example, to record all the messages in a minor frame while only getting an interrupt on the last message in the frame.

The coding example shows how to enable Bus Controller interrupts for a specific BC message with the option of only recording the message in the interrupt queue.

### 8.2.2 Selecting Bus Monitor Interrupts

Bus Monitor interrupt events are set via the Interrupt Enable word bit fields supplied in the call to BusTools_BM_MessageAlloc. Review the Bus Monitor examples in the BusTools/1553-API software distribution Examples directory regarding Bus Monitor Interrupts. The firmware records Bus Monitor messages in the interrupt queue only if the Interrupt Enable word is non-zero. If you want the firmware to record all Bus Monitor messages in the interrupt queue, use the BT1553_INT_END_OF_MESS bit assignment in the Interrupt Enable word.

For Bus Monitor applications only, you may use the “nth occurrence feature” to interrupt on ‘one out of every N’ messages and read N messages once the nth message sets the interrupt. Use BusTools_BM_FilterWrite to set up this feature.

Starting with BusTools/1553-API and firmware version 6.03, applications can suppress hardware interrupts from the board. This option is available when initializing the Bus Monitor with BusTools_BM_Init by disabling interrupts in the *bm_ctrl* parameter. That allows BM message events to be recorded in the interrupt queue but not generate a hardware interrupt. This is useful in limiting the number interrupt processed by the ISR. Applications process BM messages using BusTools_BM_ReadLastMessageBlock.

### 8.2.3 Selecting Remote Terminal Interrupts

The BusTools/1553-API allows Remote Terminals to select interrupt events for each RT Address, Subaddress, Receive/Transmit, and buffer number combination. This means you can specify different interrupt conditions for each of these settings. You can select BT1553_INT_END_OF_MESS on a transmit message sent to RT 4, Subaddress 4, while selecting BT1553_INT_ME_BIT on a receive message sent to RT 4, Subaddress 4. Set the Interrupt Enable word in the call to BusTools_RT_MessageWrite.

The Remote Terminal coding examples in the BusTools/1553-API software distribution Examples directory demonstrate several aspects for programming a Remote Terminal. The firmware records only RT messages in the interrupt queue that have a non-zero Interrupt Enable word. If you want the firmware to record every RT message, then you must at least set the Interrupt Enable bit BT1553_INT_END_OF_MESS in the call to BusTools_RT_MessageWrite

### 8.2.4 Selecting External Trigger Interrupts

Interrupts from an external trigger are supported via invocation the API function BusTools_ExtTrigIntEnable. This function enables the external trigger interrupt, resulting in an external trigger interrupt event recorded in the interrupt queue. There is no message data or time-tag data associated with this interrupt so the message pointer for a trigger interrupt event is NULL.

## 8.3 Interrupt Processing

When the firmware processes a 1553 message, it does a bitwise “AND” of the 32-bit “Interrupt Enable word” with the 32-bit “Interrupt Status word”. If the result is non-zero, the firmware writes the message data into the interrupt queue. When this happens, the firmware clears the “interrupt acknowledge” bit, sets the “interrupt mode” bit, writes the message address, and updates the “interrupt_queue_pointer” register with the address of the next interrupt queue entry. If you enabled hardware interrupts, the hardware then triggers the hardware interrupt line to the host.

Hardware interrupts from MIL-STD-1553 bus traffic can overwhelm even a fast host processor. For example, Mode Codes can occur every 50µs. A PC running Windows can take over 50µs to process an interrupt, resulting in a PC that appears to be “locked up”. Use care in selecting which interrupts to process.

### 8.3.1 Interrupt Queue Software Operation

If you use the BusTools_RegisterFunction routine, the API hides the structure of the interrupt queue from your application. The API looks at the queue at a given time interval. During this interval, it processes all the messages transacted during this time. To reduce latency and improve overall processing efficiency, you may want to write your own polling or interrupt processing functions. If you write your own polling or interrupt processing code, you need to ensure that you process all messages added to the queue during this time interval.

When servicing the Interrupt Queue, the host application must keep track of both the current queue pointer stored in “interrupt_queue_pointer” register and the last queue pointer position. The host application must then check each entry between the last queue pointer and the current “interrupt_queue_pointer” for events of interest.

When polling, look at the queue at a given time interval. During that interval, the firmware could have written several messages into the queue. Read all the messages written into the queue since the last look or you could miss data. Even if you are using hardware interrupts, you must check to make sure you read all the data transacted.

The interrupt mode word shows the source of the event. The mode word identifies BC, BM, RT, BM trigger, and tag timer overflow interrupts. Use the message address to read the message data. Then, step through all the messages between the last queue pointer and the current queue pointer, using the pointer to the next queue entry.

### 8.3.2 Polling

User applications can directly poll the interrupt queue, or they can use API functions to poll and read data from the interrupt queue. Review the polling example applications in the BusTools/1553-API software distribution Examples directory regarding methods to poll the interrupt queue.

The API has nine polling functions that hide the interrupt queue structure, while allowing the caller to access queue data. These functions are:

- BusTools_BC_ReadNextMessage
- BusTools_BC_ReadLastMessage
- BusTools_BC_ReadlastMessageBlock
- BusTools_RT_ReadNextMessage
- BusTools_RT_ReadLastMessage
- BusTools_RT_ReadlastMessageBlock
- BusTools_BM_ReadNextMessage
- BusTools_BM_ReadLastMessage
- BusTools_BM_ReadlastMessageBlock

Use these functions in place of BusTools_RegisterFunction if you want to reduce the latency of high-level interrupt processing. Make sure that the interval between calling these functions is less than the time it takes to overflow the interrupt queue.

### 8.3.3 Interrupts

A hardware interrupt occurs when a specified interrupt event happens on the bus. When there is an interrupt, the Interrupt Service Routine (ISR) determines which channel(s) on the device generated the interrupt. The ISR clears the interrupt and invokes a deferred process that stores the interrupt data including the card number for the channel and passes it on to a user supplied interrupt callback function. The hardware interrupt deferred processing function searches the interrupt queue for all interrupt events, but there is always at least one matching interrupt event.

BusTools_RegisterFunction is the mechanism the API provides to connect a user function to an interrupt event. BusTools_RegisterFunction is available for Windows, Linux, LynxOS, VxWorks and Integrity. VxWorks also has low-level interrupt processing that clears the interrupt and invokes the user callback function.

BusTools_RegisterFunction allows you to filter interrupts (Table 8-2). If you are only interested in BM interrupts, select EVENT_BM_MESSAGE when invoking BusTools_RegisterFunction. You can OR together events from Table 8-2. The coding example in the in the BusTools/1553-API software distribution Examples directory illustrates this technique.

Table 8-2 Interrupt Events Filters

| Event Pneumonic | Event Description |
|---|---|
| EVENT_IMMEDIATE | Immediately calls the users function without processing the interrupt queue |
| EVENT_EXT_TRIG | External Trigger Interrupt |
| EVENT_TIMER_WRAP | Tag Timer overflow or discrete input |
| EVENT_RT_MESSAGE | RT message transacted |
| EVENT_BM_MESSAGE | BM message transacted |
| EVENT_BC_MESSAGE | BC message transacted |
| EVENT_BC_CONTROL | BC control transacted (Last, conditional branch, timed no-op) |
| EVENT_BM_TRIG | BM trigger event (start/stop) |
| EVENT_BM_START | BM started (BusTools_BM_StartStop) |
| EVENT_BM_STOP | BM stopped (BusTools_BM_StartStop) |
| EVENT_BM_OVRFLW | BM detect a overflow (head PTR = tail PTR) |
| EVENT_BC_START | BC started (BusTools_BC_StartStop) |
| EVENT_BC_STOP | BC stopped (BusTools_BC_StartStop) |
| EVENT_RT_START | RT started (BusTools_RT_StartStop) |
| EVENT_RT_STOP | RT stopped (BusTools_RT_StartStop) |
| EVENT_RECORDER | BM recorder buffer has 64K or timeout |
| EVENT_MF_OVERFLO | Minor frame timing overflow |
| EVENT_LP_MF_OVFL | Low Priority Aperiodic message list extend beyond 1 frame |
| EVENT_HP_MF_OVFL | High Priority Aperiodic message list extend beyond 1 frame |
| EVENT_BC_BSY_OVFL | Overflow on BC Busy |
| EVENT_API_OVERFLO | BM API Recorder buffer overflowed |
| EVENT_HW_OVERFLO | BM HW Recorder buffer overflowed |

EVENT_IMMEDIATE allows the application to specify the API immediately invoke the user callback function when an interrupt occurs on the specified channel. You can use no other interrupt events on a channel if you use EVENT_IMMEDIATE. The API_INT_FIFO structure contains no message information. The callback function must do all low-level processing to get the information. The option can reduce latency under some conditions

The example code searches for a specific interrupt event in the line
if(sIntFIFO → FilterType == ***YOUR_EVENT_TYPE***).

You can omit this line if you have only a single interrupt event for this interrupt function.

### 8.3.4 Setting Up Interrupts with BusTools_RegisterFunction

BusTools_RegisterFunction handles both software (timer interval) and hardware interrupts. The underlying interrupt function searches the interrupt queue for matching records, records them in the API_INT_FIFO and then calls the user-supplied function. Use the following steps to setup interrupts.

**Bus Controller Interrupts:**

1. Set the BC Interrupt events in the call to BusTools_BC_Init.
2. Set the BC messages that interrupt in BusTools_BC_MessageWrite.
3. Set EVENT_BC_MESSAGE or EVENT_BC_CONTROL in the API_INT_FIFO structure filterType.
4. You can further refine the interrupt selection by setting FilterMask and EventMask. FilterMask allows you to select an interrupt by word Count. EventMask allows you to select an interrupt by the message status bits.

**Bus Monitor Interrupts:**

1. Set the BM Interrupt events in the call to BusTools_BM_MessageAlloc.
2. Set EVENT_BM_MESSAGE in the API_INT_FIFO structure filterType.
3. Further refine the interrupt selection by setting FilterMask and EventMask. FilterMask allows you to select interrupts by word Count. EventMask allows you to select interrupts by the message status bits.

**Remote Terminal Interrupts:**

1. Set the RT interrupt events in the call to BusTools_RT_MessageWrite. You can set interrupt events for each RT address, RT Subaddress, Transmit/Receive, and buffer combination.
2. Set EVENT_RT_MESSAGE in the API_INT_FIFO structure filterType.
3. Further refine the interrupt selection by setting FilterMask and EventMask. FilterMask allows you to select interrupts by word Count. EventMask allows you to select interrupts by the message status bits.

**External Trigger Interrupts:**

1. Call BusTools_ExtTrigIntEnable and enable the external trigger interrupt.
2. Set EVENT_EXT_TRIG in the API_INT_FIFO structure filterType.
3. No other filtering is available.

## 8.4 Polled or Interrupt Driven?

The API can use either hardware interrupts or software polling. Determining the most efficient mechanism (polled or interrupt-driven) requires an in-depth knowledge of the characteristics of the overall system. In both cases, the interrupt queue tracks the events on the 1553 bus.

Consider the following when choosing between polled and interrupt driven mode:

- Operating System (OS) Interrupt Latencies. The minimum latency is a function of the OS and the speed of the processor. The maximum latency is a function of the OS and the other processes, such as network interfaces and disk accesses.
- Allowable latency between a message occurring on the 1553 bus and the software processing that message.
- The normal and maximum number of messages occurring per second and the number of messages per second causing interrupts.
- The number of software processing cycles required per second.
- The speed of the host processor.
- The amount of processing time available for 1553 processing.
- The availability of hardware interrupts on the host processor.

Interrupt driven processing is normally most efficient when the interrupts occur irregularly or at long intervals. It is also desirable when you must process the data immediately after the event. The processing load is directly proportional to the interrupt rate.

Polled operation is normally most efficient when the interrupts are closely spaced in time. The processing load is not a strong function of the interrupt rate.

# 9 • Board Memory Organization

Abaco Systems MIL-STD-1553 products operate using data structures defined in the board's host interface, stored in onboard memory. Knowledge of these details is not necessary when programming the board with the API. However, this information is useful if you need to modify the API code for special needs.

This chapter provides an overview of the memory structures as defined in the "UCA32 LPU Reference Manual" and "UCA32 Global Reg Ref Manual". It also describes how the API routines create and align the various memory structures. The size of these structures is limited by the amount of memory on the board, and by various placement restrictions required by both the hardware and the API.

Additional information about the memory organization for older firmware revisions is available in the "MIL-STD-1553 UCA Reference Manual".

## 9.1 Hardware Operation

There is one Megabyte of memory on all boards addressed as 512k 16-bit words. The API partitions this memory into five categories of structures:

- **BM Operation**: Memory structures used to control the operation of the Bus Monitor.
- **RT Operation**: Memory structures used to control the operation of one or more Remote Terminals.
- **BC Operation**: Memory structures used to control the Bus Controller.
- **Error Injection Buffers**: Buffers used to control the injection of error conditions into the 1553 data stream. If the application defines a BC or RT, there must be at least one error injection buffer.
- **Interrupt Queue**: The Interrupt Queue stores significant events. The host processor uses the Interrupt Queue to process these events as they occur.

## 9.2 Software Operation

BusTools/1553-API routines provide a framework that makes the actual memory organization transparent. However, some knowledge of the framework and memory organization used by the API routines may be helpful. This section describes the assumptions made by the API routines.

### 9.2.1 Memory Segmentation

All boards have one megabyte of on-board RAM memory organized as 512K 16-bit words. Most of the address-offsets on the board are 16-bits; however, 19 bits are required to address the full memory range. BusTools/1553-API uses only the upper 16 bits of the address. This restricts the start address of these structures to even eightword boundaries. A few structures require 16-bit addressing. This limits these structures to the first 64K words of memory, referred to as Segment 1.

Starting with F/W version 6.0 all address offsets on the board are 32-bits. The only restriction is that structure addresses start on an even double word boundary.

## 9.3 Memory Organization

The API assigns memory at the following word offsets for Abaco Systems 1553 boards with firmware version 5.x or earlier:

- **(0x0000 - 0x007F) Hardware Control Registers**: They include the operational control bits and the addresses of the various control structures.
- **(0x0080 - 0x0094) BM Trigger Buffer**: Even if no BM triggers have been defined, this block must be allocated and set to the default values of all zeros, which is automatically done when the channel is initialized.
- **(0x0094 - 0x009C) Default BM Control Buffer**: When the BM is initialized, all addresses in the BM Filter buffer point to this location. The default Control Buffer enables recording of all messages for the specified RT address/ subaddress. The second buffer in this block disables all word counts. The API uses it whenever the caller of BusTools_BM_FilterWrite disables all word counts for a specified RT address/subaddress.
- **(0x00D9 - 0x0450) Interrupt Queue**: This memory block contains the Interrupt Queue. BusTools/1553-API operates with 296 entries.
- **(0x0451 - 0x080D) Error Injection Buffers**: BusTools/1553-API operates with 30 Error Injection Buffers.
- **(0x1000 - 0x17FF) BM Filter Buffer**: There is one word in this block for each possible RT address, Transmit/Receive, and RT subaddress combination. This block is always present. By default, all addresses point to the default BM Control buffer. The block must begin on an even 2K-word boundary.
- **(0x1000 - 0x7FFFF) Available**: This area is available for BM, RT, and BC control structures and data buffers, RT message buffers and RT broadcast control buffers (508K words). If Broadcast is enabled, 3906 or 4032 words are reserved for the Broadcast RT Control Buffers, depending on the state of the Subaddress 31 Mode Code Enable switch. The API allocates BM and BC message and data buffers, and RT data buffers, at word addresses beginning with 0x10000. RT control buffers are allocated below 0x10000.

The API assigns memory at the following word offsets for Abaco Systems 1553 boards with firmware version 6.x and later:

- **(0x0000 - 0x03FF) Hardware Control Registers**: They include the operational control bits, and the addresses of the various control structures, (registers are in separate memory segment and do not overlap RAM).
- **(0x0000 - 0x005B) BM Trigger Buffer**: Even if no BM triggers have been defined, this block must be allocated and set to the default values of all zeros, which is automatically done when the BusTools_BM_Init routine is called.
- **(0x005c- 0x006B) Default BM Control Buffer**: When the BM is initialized, all addresses in the BM Filter Buffer point to this location. The default Control Buffer enables recording of all messages for the specified RT address/ subaddress. The second buffer in this block disables all word counts. The API uses it whenever the caller of BusTools_BM_FilterWrite disables all word counts for a specified RT address/subaddress.
- **(0x0100- 0x10FF) Interrupt Queue**: This memory block contains the Interrupt Queue. BusTools/1553-API operates with 512 queue entries.
- **(0x1100 - 0x217F) Error Injection Buffers**: BusTools/1553-API operates with 30 Error Injection Buffers.
- **(0x3000 - 0x4FFF) BM Filter Buffer**: There is one 32-bit element in this block for each possible RT address, Transmit/Receive, and RT subaddress combination. This block is always present. By default, all addresses point to the default BM Control buffer.
- **(0x5000 - 0x7FFFF) Available**: This area is available for BM, RT, and BC control structures and data buffers, RT message buffers and RT broadcast control buffers (508k words). If Broadcast is enabled, 3906 or 4032 words are reserved for the Broadcast RT Control Buffers, depending on the state of the Subaddress 31 Mode Code Enable switch. The API allocates BM and BC message and data buffers, and RT data buffers, at word addresses beginning with 0x10000. RT control buffers are allocated below 0x10000.

# 10 • C# Support

## 10.1 Introduction

The .NET Interop Reference Solution can give you a quick start to harnessing the power of the .NET framework for your 1553 application.

The distributed Bustools/1553-API library is an "unmanaged" DLL and does not use the .NET framework. By using managed wrapper classes, you can use the distributed Bustools/1553-API library in your managed application.

The reference solution consists of:

- A managed wrapper class written in C# that encapsulates the Bustools/1553-API functions and data types.
- A sample managed GUI written in C# that uses the wrapper class to operate an Abaco Systems 1553 board.
- A sample C# project and C++ unmanaged DLL project that can be used as a workbench to explore aspects of .NET interop.

The managed wrapper class can be used with C#, VB.NET, or any of the managed languages .NET supports.

This documentation assumes familiarity with Visual Studio 2008 or later and creating and running .NET applications.

## 10.2 The Reference Solution

The reference solution is a Microsoft Visual Studio 2008 solution in the root folder, named **BustoolsInterop.sln**. When you open this solution, you will see four projects in the solution explorer window.

- BustoolsCsApp: A C# GUI that opens and operates a 1553 board.
- BustoolsCsWrapper: A C# class library that wraps the unmanaged Bustools/1553-API library functions, constants, and data types.
- BustoolsInterop: A C# project that works in conjunction with the C DLL project to demonstrate .NET interop concepts.
- C DLL: An unmanaged C++ DLL that represents an unmanaged API. Used by the BustoolsInterop project.

Familiarity with .NET and C# concepts including interop is required to understand the Reference Solution.

NOTE
The Reference Solution demonstrates one way to use the Bustools/1553-API distribution with a .NET application. Other ways and other user-defined wrapper definitions are also possible.

## 10.3 The API, Data, and Constants Classes

The Bustools1553 namespace encapsulates all the API functions, Data Types, and Constant definitions. It is recommended you do not change this namespace name, as it identifies the wrapper and provides name separation when loaded into other projects.

The API static class contains managed entry points for each API call in the unmanaged Bustools/1553-API C library. .NET interop requires that managed entry points be contained in a static class. This class is found in file API.cs.

The DataTypes namepsace contains managed equivalents of the structures required by the unmanaged Bustools/1553-API C library. The managed equivalents are implemented using C# structures, classes, and unions. This namespace is found in file DataTypes.cs.

The Constants static class contains managed definitions of the constants required by the unmanaged Bustools/1553-API C library. This class is found in file Constants.cs.

## 10.4 Building and Running the Application

You can build and run the Reference Application "out of the box", as long as you meet the following assumptions:

- You have Microsoft Visual Studio 2005 or later.
- Abaco MIL-STD-1553 avionics hardware and Bustools/1553-API software has been properly installed on your host.

In Visual Studio, set BustoolsCsApp as the startup project, and then select "Rebuild All". You can then run BustoolsCsApp.

## 10.5 Adding the Managed Wrapper to an Existing .NET Application

First, it is suggested (but not required) that you add the C# project BustoolsCsWrapper to your existing .NET solution.

Then, in the Solution Explorer, right-click your project and select "Add Reference". If you added BustoolsCsWrapper to your solution, click the Projects tab and select BustoolsCsWrapper. Otherwise, select the Browse tab and locate BustoolsCsWrapper.dll on your disk.

At the top of each of your code pages, add the following lines:

```
using Bustools1553;
using Bustools1553.DataTypes;
using System.Runtime.InteropServices;
```

You may need to edit the file API.cs in the BustoolsCsWrapper project. At the top of this file is a statement that defines exactly where Busapi32.dll should be found.

You can now use the managed wrapper classes in your project.

## 10.6 Important Coding Differences when Using the .NET Wrappers

Managed .NET Applications can't directly access memory via pointers, so there is no possibility of sharing memory between the Managed Application and an Unmanaged DLL. This is where "Interop Marshaling" with "Platform Invoke" a.k.a. "P/Invoke" comes in.

When a Managed Application calls functions in an Unmanaged DLL, the parameters to be passed are "Marshaled" across the managed / unmanaged boundary by P/Invoke. In general, marshaling means copying data across the boundary in one or both directions.

When a structure or class is Marshaled to the Unmanaged DLL, it is copied to the DLL's address space where the DLL can access it. When the function returns, the structure may be Marshaled back to the Managed Application if specified. Because of this, certain Bustools/1553-API operations that require shared pointers to **IntFifo structures** need to be modified.

Additionally, marshaling presents some challenges with multi-dimensional arrays which are embedded in structures. Because of this, it is required to flatten the array and use accessor functions to get at it with multiple indices.

Lastly, most Bustools/1553-API functions require pointers to structures, so a .NET **class** (a reference type) is required. However, some functions require arrays of structures. These must be Marshaled as a .NET struct (a value type). This means that the .NET wrapper contains both struct and class representations of certain data types.

### 10.6.1 IntFifo Creation and Updating

When using the BusTools/1553-API in C, the application creates the IntFifo and passes the API a pointer to it. When using .NET, request that the API create the IntFifo because it must stay in unmanaged memory.

Additionally, when the managed application has completed its IntFifo processing and modified the tail pointer, it must make a special call to have the API update its version of the tail pointer.

For an example of this, see the function **SetupBcIntFifo** in RtTests.cs, in the **BustoolsCsApp** project.

### 10.6.2 Multi-Dimensional Array Data

The **API_INT_FIFO** Data Type contains the embedded multi-dimensional arrays **eventMask** and **filterMask**. The wrapper flattens these and defines these members as Private. To access them, use the accessor functions **GetEventMask**, **SetEventMask**, **GetFilterMask**, and **SetFilterMask**.

The **API_BC_MBUF** and **API_BC_MBUF_STRUCT** Data Type contains the embedded multi-dimensional array **data**. The wrapper flattens it and defines this member as Private. To access it, use the accessor functions **GetData** and **SetData**.

### 10.6.3 Class and Struct Versions of Data types

The Bustools/1553-API functions **BC_ReadLastMessageBlock**, **BM_ReadLastMessageBlock**, and **RT_ReadLastMessageBlock** require arrays of **API_BC_MBUF**, **API_BM_MBUF**, and **API_RT_MBUF_READ**, respectively. Therefore, the wrapper provides both a class and struct version of each. The struct versions have "_STRUCT" at the end of their names.

If you do not use the struct MBUF version when calling the above functions, you get an Execution Exception.

For an example of this, see the function BcUserTimerCallback in **RtTests.cs**, in the **BustoolsCsApp** project.

## 10.7 Application Notes

### 10.7.1 BustoolsCsApp

The primary code for this project can be found in the Form1 and RtTests classes. Form1 implements a basic GUI, and RtTests contains source for a simple demonstration application that configures and interacts with an Abaco Systems 1553 board.

### 10.7.2 BustoolsInterop

The primary code for this project can be found in the Form1 class. Although this is a Windows Forms project, the interop tests all take place in the Form1 constructor, and do not actually implement any GUI operations. Thus, the result is a blank form.

Use the Microsoft Visual Studio debugger to step through the constructor code to see how interop works.

# 11 • LabVIEW Support

## 11.1 Information

LabVIEW support for BusTools/1553-API is available as a separate product from Abaco Systems called LV-1553. LV-1553 combines components built on the fundamental Abaco Systems BusTools/1553-API into a suite of LabVIEW Virtual Instruments (VIs), complete with integrated examples that are ready to use. Its aim is to provide the tools you need to operate Abaco Systems 1553 products with LabVIEW. Further information can be found in the "LV-1553 User's Manual".

## 11.2 System Requirements

Make sure that your PC (or compatible) and its software conforms to the following requirements:

- Microsoft Windows 32-bit XP, 32-bit/64-bit Windows 7/2008R2 (SP1 and KB3033929 required), 8, 8.1, Windows Server 2012 R1/R2, or 10 is required.
- National Instruments LabVIEW v8.6 (or higher) is required.

## 11.3 LabVIEW VI Examples

The LV-1553 distribution also includes example VIs that show how to implement 1553 applications using the LV-1553 VI library. These examples may be, in some cases, suitable for simple applications; however, the example VIs are only tutorials and not designed as complete applications.

# 12 • VxWorks Support

## 12.1 Introduction

VxWorks is an embedded real-time operating system (RTOS). BusTools/1553-API supports VxWorks on PowerPC and Intel x86 processors. You can also port BusTools/1553-API support to other BSPs. See Chapter 15, “Porting the API to Other Environments”.

Two VxWorks operating systems are supported by BusTools/1553-API, VxWorks 6.x and VxWorks 7. The current support for the V6.x kernel covers versions 6.2 to 6.9. The driver and API provide limited support for PCI/PMC devices under VxWorks 5.5.1.

BusTools/1553-API supports both kernel modules with legacy device drivers (for PCI/PMC and VME devices) and VxBus device drivers (for PCI/PMC and PCI Express/XMC devices).

## 12.2 VxWorks Installation

Follow the instructions provided in the Installation Options section of the *BusTools1553-API VxWorks User Manual* located in the VxWorks_Install folder on the BusTools/1553-API CD and accessible via the Abaco Systems website link below.

LINK
https://www.abaco.com/download/bustools1553-api-vxworks-user-manual

# 13 • UNIX Support

## 13.1 Introduction

The BusTools/1553-API distribution contains support for Linux and LynxOS. The table below shows the boards supported by each operating system.

Table 13-1 UNIX Support Matrix

| Board Type | Linux | LynxOS |
|---|---|---|
| QPCX-1553 | Yes | Yes |
| QPM-1553 | Yes | Yes |
| QPC-1553 | Yes | Yes |
| Q104-1553P (PCI) | Yes | Yes |
| RPCIE-1553 | Yes | - |
| R15-LPCIE | Yes | - |
| R15-MPCIE | Yes | - |
| R15-AMC | Yes | - |
| R15-USB | Yes | - |
| R15-EC | Yes | - |
| RPCC-D1553 | Yes | - |
| RXMC-1553 | Yes | - |
| RXMC2-1553 | Yes | - |
| QVXI2-1553X | - | Yes |
| RQVME2-1553 | - | Yes |
| RAR15-XMC-IT/RAR15XF | Yes | - |

Abaco distributes the Linux and LynxOS API versions with the BusTools/1553-API distribution media. The installation process installs both the API as a shared library and the driver as a module. Application programs link with the shared library. These UNIX API versions support all core API functions, as well as interrupts (for POSIX compliant systems).

## 13.2 Compiling Applications

Use the following command line to build and link your UNIX application program:

*cc* app-name.c –D*nnnnn* –I/*Condor-Default_Directory*/Include –lbusapi –o app-name

Where *cc* is your C compiler such as cc or gcc and –D*nnnnnnn* defines your target operating system and processor. Options include:

- _LINUX_X86_
- LYNXOS_VME_PPC
- LYNXOS_PMC_PPC
- LYNXOS_X86

Some compilers may not automatically recognize C++ (//) style comments. You may need to provide a compiler option to allow C++ style comments. For example, SUNS native compiler requires the compiler option –xCC.

## 13.3 Linux Installation

Follow the instructions provided in the Installation section of the **Linux_install_v<LSP VERSION>.txt** file located in the *Linux_install* folder on the BusTools/1553-API CD, in the Linux distribution archive file **linux_bt1553_v<*API VERSION*>.tgz**, and on supported product web pages accessible via the Abaco MIL-STD-1553 product webpage.

## 13.4 LynxOS Installation

Follow the instructions provided in the Install section of the text file appropriate for your LynxOS host:

***lynxos_4_ppc_install_bt1553_vNNN.txt*** for a LynxOS 4 PowerPC host
***lynxos_5_ppc_install_bt1553_vNNN.txt*** for a LynxOS 5 PowerPC host
***lynxos_4_x86_install_bt1553_vNNN.txt*** for a LynxOS 4 x86 host

where *NNN* is the BusTools/1553-API revision. These files are located in the *LynxOS_install* folder on the BusTools/1553-API CD, and on supported product web pages accessible via the Abaco MIL-STD-1553 product webpage.

LINK
https://www.abaco.com/download/bustools1553-api-lynxos-4-powerpc-support

LINK
https://www.abaco.com/download/bustools1553-api-lynxos-4-x86-support

# 14 • Integrity Support

## 14.1 Introduction

Green Hills Integrity is a secure high reliability real-time operating system (RTOS) intended for use in mission critical embedded systems. The BusTools/1553-API distribution provides support for Integrity on systems with PowerPC and x86 processors for the following boards: RAR15-XMC-IT/RAR15XF, R15-MPCIE, R15-LPCIE, RPCIE-1553, RXMC2-1553, RXMC-1553, QPM-1553, QPCX-1553, QCP-1553, QVXI2-1553X, and RQVME2-1553.

Integrity is flexible in how it builds the kernel and application software. You can build a monolith containing the kernel, BSP, and application software, or you can build a separate kernel/BSP and the application as a Dynamic Download. The BusTools/1553-API Integrity distribution supports either method.

The BusTools/1553-API distribution provides the Integrity PCI driver, API source code, and example program source. The source code allows you to modify the static library if needed.

## 14.2 Integrity Installation

Follow the instructions provided in the Installation section of the **Integrity_install.pdf** file located in the *Integrity_install* folder on the BusTools/1553-API CD and on supported product web pages accessible via the Abaco MIL-STD-1553 product webpage.

LINK
https://www.abaco.com/download/bustools1553-api-integrity-installation-guide

# 15 • Porting the API to Other Environments

This chapter describes the steps to customize the API and to port to a non-supported operating system. The BusTools/1553-API Installation includes device drivers, low-level interface routines, and libraries for all supported platforms. The Installation includes pre-compiled libraries for all operating systems. There is no need to build the API unless you intend to customize the code or port to a non-supported platform.

These API routines provide consistent, documented interfaces to all hardware features while hiding the initialization, addressing details and other differences that exist among the various board models.

BusTools/1553-API also supports easy customization and porting to non-supported operating systems. All source and include files needed to build the BusTools/1553-API library come on the installation CD-ROM. Installation copies these files to the Include and Source directories on your host system.

## 15.1 Supported Compilers

The latest BusTools/1553-API version for the Windows/Intel platforms is built using Microsoft C/C++ version 6.0 and Microsoft Visual Studio 2008. Linux and LynxOS libraries use the GNU C/C++ compiler. The Integrity API builds with the native Multi C compiler.

## 15.2 API Source Code

The BusTools/1553-API installation includes the source code for the API, providing the ability to customize or port the API. The distributed BusTools/1553-API libraries control all Abaco Systems 1553 boards, and in most cases this installation should suffice; however, in some cases you may want to customize the API to improve speed or size, or port to an unsupported platform

The API is written in C. Although it is intended to be standard C, there might be certain constructs in the code that need to be changed when migrating to a different compiler.

There are two ways to customize the API. The first is to use the pre-defined macros to control how the API compiles. These macros allow you to include or exclude API functions. The other is to modify the source and re-compile the BusTools/1553-API. Modify the API code with caution as unintended effects could result.

## 15.3 Rebuilding the API

The following table itemizes combinations of compiler and target environments that might be used when re-building the API for the Windows/Intel environments:

Table 15-1 Rebuilding the API on Various Environments

| Target Environment | Target Tool Set | Compile Source with | Implementation Comments |
|---|---|---|---|
| Windows Application | Microsoft – compatible tools including C/C++ and Visual Basic | Microsoft C/C++ V6 or above. | Distribution API is compiled using Microsoft Visual Studio 6 and 2008. While the DLL is compatible with all standard tools, the .lib file is specific to the Microsoft tool version. |
| VxWorks | Workbench 3.2 or greater | C Compiler supplied with Workbench | Build according to the instructions in Chapter 12, "VxWorks Support". |
| Integrity | Multi v5.10 or greater | Multi C compiler | Build according to the instructions in Chapter 14, "Integrity Support". |
| Linux Versions 2.4x, 2.6x, 3.x, 4.x, and 5.x | GNU Tools GCC Version 2.0 | GNU Tools GCC Version 2.0 | Link with the libceill.so low-level library file which resides in the default path for libraries (/usr/lib or /usr/lib64). |

The installation provides a project or Make file for rebuilding the API. The Windows BusTools/1553-API projects for Microsoft Development Studio are located under the respective toolset directory in the installed directory tree based on the host operating system:

for Windows 7 and later versions:

C:\Users\Public\Documents\Condor Engineering\BusTools-1553-API\Windows API Projects

for Windows XP:

C:\Program Files\Condor Engineering\BusTools-1553-API\Windows API Projects

The Linux and LynxOS installations provide Make files for building the device driver, example program, and API library. These are located within the respective driver, examples, and source directories beneath the bt1553 directory. VxWorks does not have a make file or project, but Chapter 12, "VxWorks Support" provides information regarding where to find the project build instructions.

When you compile the API, you need to include busapi.h, which in turn includes target_defines.h. These files are in the BusTools/1553-API Include directory, created during the software installation process. The target_defines.h file determines how the API compiles for each supported target. There are define-blocks for each target. Table 15-2 shows which platforms are currently supported by the BusTools/1553-API.

Table 15-2 Currently Supported Platforms and Operating Systems

| Operating System/Platform | Compiler Directive | Comment |
|---|---|---|
| Windows | _WIN32 or WIN32 | The Microsoft Visual C++ compiler defines WIN32 |
| Linux (x86 platform) | _LINUX_X86_ | Defined in the Makefile when building the API |

| Operating System/Platform | Compiler Directive | Comment |
|---|---|---|
| VxWorks x86 with PCI/PCIe/PMC/XMC | VXW_PCI_X86 | All VxWorks x86 BSP's with PCIbus-based boards. |
| VxWorks x86 with VME | VXW_VME_X86 | All VxWorks x86 BSP's with VMEbus-based boards. |
| VxWorks PPC with VME | VXW_VME_PPC | All VxWorks PowerPC BSP's with VMEbus-based boards. |
| VxWorks PPC with PCI/PCIe/PMC/XMC | VXW_PCI_PPC | All VxWorks PowerPC BSP's with PCIbus-based boards. |
| Integrity PPC with PCI/PCIe/PMC/XMC | INTEGRITY_PCI_PPC | Green Hills Integrity PowerPC with PCIbus-based boards. |
| Integrity PPC with VME | INTEGRITY_VME_PPC | Green Hills Integrity PowerPC with VMEbus-based boards. |
| LynxOS PPC with VME | LYNXOS_VME_PPC | LynxOS PowerPC host with VMEbus-based boards. |
| LynxOS PPC with PCI/PCIe/PMC/XMC | LYNXOS_PMC_PPC | LynxOS PowerPC host with PCIbus-based boards. |
| LynxOS x86 with PCI/PCIe/PMC/XMC | LYNXOS_X86 | LynxOS x86 host with PCIbus-based boards. |

### 15.3.1 Customizing the BusTools/1553-API Using Pre-Defined Symbols

You can customize how the API compiles by changing the symbol definitions within the target blocks. If you port to an unsupported platform, you need to use an existing target definition block or build a new target definition block.

If you choose to create a new block, you should start with an existing block like your target platform. For example, if you port the API to an SGI processor running IRIX, you may want to create a new block.

Within each block these symbol-defines determine how the API builds. Table 15-3 shows the symbols you can use to customize how the API compiles.

Table 15-3 Symbols that Customize the BusTools/1553-API

| Symbol Name | Purpose | O/S Support | Comment |
|---|---|---|---|
| MAX_BTA | Define the number of 1553 interface channels that the API can use | All | This is set to a default of 64 for all API versions. You can increase or decrease as needed. |
| INCLUDE_VMIC | Include the optional VMIC support. | Windows | This allows the Win32 API to run on VMIC Intel processors and control VME boards. Defined by default for WIN32 and undefined for all other O/S |
| _WIN16_INTERRUPTS_ | Includes support for legacy 16-Bit Windows (Windows 3.1) interrupts | Windows | Some applications use legacy 16-Bit Windows interrupt handling. Defined by default for WIN32, undefined for other O/S's |

| Symbol Name | Purpose | O/S Support | Comment |
|---|---|---|---|
| FILE_SYSTEM | This enables logging information to file. | All | The API has dump and error logging functions that write data to files on a disk drive. If your O/S supports a file system, you can use these features by defining this symbol. Defined for systems supporting file systems like Windows and Linux. Un-defined for embedded systems like VxWorks and Integrity |
| DEMO_CODE | Allows the API to run in demo mode without hardware. | WIN32 | By default only defined for WIN32 systems. |
| ADD_TRACE | Allows a trace of all API calls. | All | Use for diagnostic purposes. By default undefined for all O/S's. Requires a file system (FILE_SYSTEM) |
| DO_BUS_LOADING | Allows the API to calculate bus loading statistic | All | This optional feature allows the API to calculate Bus Loading statistics. The Default Windows API defines this symbol. All others undefine the symbol. |
| _INIT_EXTERNAL_ | Uses external DLL for API initialization | Windows | When running in a Windows environment some systems like NI-VXI can use an external DLL to initialize the board. You supply the DLL and functionality according to documentation for BusTools_API_InitExternal. By default this is defined for Windows and undefined for all other O/S's. |
| _LABVIEW_ | Includes the LabVIEW routines for use with National Instrument LabVIEW programs | Windows | Allows the API to run with LabVIEW. Defined for Windows and undefined for all other O/S's. |
| _USER_DLL_ | Allows a Windows system to include a user-defined DLL to alter how the API executes. This is normally used with BusTools, Graphical Bus Analyzer program, but can be used for other applications. | Windows | Defined under Windows, undefined for all other systems. Un-defining removes the feature from the API. |
| _PLAYBACK_ | Include the Playback function into the API. | Windows and UNIX | Playback allows the API to take a BusTools-1553 Bus Monitor file (.bmd) and playback the 1553 messages recorded over a 1553 bus. Requires defining the REGISTERFUNCTION symbol, BusTools_RegisterFunction, to run. |
| INCLUDE_VME_1553 | Include the VME board capability into the API | All | The VME-1553 uses a large firmware file during initialization. If you aren't using the VME-1553 board, you can reduce the size of the API by undefining this symbol. Defined for Windows and all systems supporting the RQVME2-1553 and QVME-1553 board. |

| Symbol Name | Purpose | O/S Support | Comment |
|---|---|---|---|
| NO_ASSEMBLY | Excludes assembly language code | All | The API uses some assembly code in the time.c file to optimize the time functions. This code is for Windows systems. Undefine this symbol for all non-Windows environments. |
| _GCC_ | Compile with gcc | UNIX VxWorksLynxOS | This define specifies if you are compiling with GNU gcc. It allows the API to take advantage of gcc structure packing. |
| _VMEBOARDSETUP<br>_LINUXBOARDSETUP<br>_LINUXVMEBOARDSETUP<br>_X86BOARDSETUP<br>_PCIBOARDSETUP<br>_LYNXOSBOARDSETUP<br>INTEGRITY_PCI_PPC | Board setup options | UNIX VxWorks Integrity | Select the board setup option you compile with the API. The file btdrv.c includes the selected file according to the setup option you define in your target_define block. If you omit this define, you get a compiler error. |

## 15.3.2 Other Build Symbols

BusTools/1553-API uses several other symbols to allow compiling across different platforms and operating systems (see Table 15-4). When porting to an unsupported operating system, make sure you define these symbols correctly. If you port to a Windows-based system, these symbols are defined under the __WIN32__ definition block or by default. When porting a UNIX based system, defining the _UNIX_ in your target block defines these symbols for a UNIX system.

Table 15-4 Other Symbols

| Symbol | Definition | Operating System | Comment |
|---|---|---|---|
| CCONV | Calling Convention | Windows | Define as _stdcall for Window and blank for all other operating systems. |
| NOMANGLE | C++ Compilers | Windows | Define as blank for C compilers. |
| PACKED | Provides proper structure alignment and packing for GNU compilers. | UNIX | Certain structures need 2-byte alignment. When compiling with a GNU compiler define PACKED as: __attribute__ ((aligned(2),packed)) Otherwise, define PACKED blank. |
| PRAGMA_PACK | Local structure alignment for MS Visual C++ | Windows | Use for Microsoft Visual C++ compiler to pack structure with 2-byte alignment. |
| MSDELAY | Provides a 1 millisecond Sleep function | All | Define a MSDELAY(p1) function that sleeps for p1 milliseconds. For example, Linux uses the following definition:<br>#define MSDELAY(p1) usleep(p1*1000) |
| __int64<br>__unt64 | 64-bit integer and unsigned 64-bit integer | All | The time.c file requires a 64-bit integer to calculate the tag time. Declaring a 64-bit signed or unsigned integer varies between compilers. You must define __int64 as a 64-Bit integer and __uint64 as a 64-Bit unsigned integer. For example, VxWorks uses the following:<br>#define __int64 long long<br>#define __uint64 unsigned long long |
| CEI_MALLOC<br>CEI_FREE | Used for Bus Monitor setup | All | For Windows define CEI_MALLOC as GlobalAlloc and CEI_FREE as GlobalFree. All other system can use malloc and free. |

| Symbol | Definition | Operating System | Comment |
|---|---|---|---|
| CEI_MUTEX | Mutual Exclusion device | All | CEI_MUTEX is defined in target_defines.h. Base this on the O/S you are using. Defined for windows as CRITICAL_SECTION, POSIX as pthread_mutex_t, VxWorks as SEM_ID |
| CEI_EVENT | Event variable | All | CEI_EVENT is defined in target_defines.h. Base this on the event variable for our systems. Defined for Windows as HANDLE, POSIX as phtread_cond_t; Vxworks as MSG_Q_ID |
| CEI_THREAD | Thread definition | All | CEI_THREAD is defined in target_defines.h. Base this on the thread type for the system you are using. CEI_THREAD is defined in Windows as int; POSIX as pthread_t, VxWorks as int |
| CEI_THREAD_EXIT | Thread exit macro | All | This macro is defined in target_defines.h. base this on thread type you are using. Defined for Windows as ExitThread(0); POSIX as pthread_exit(0); Vxworks as taskDelete(a) where a is the tasked. |
| CEI_MUTEX_LOCK | Mutex Lock macro | All | The Mutex lock macro is defined in target_defines.h. Base this macro on the mutex you are using. Defined in Windows as EnterCriticalSection, POSIX as pthread_mutex_lock; and VxWorks as semTake |
| CEI_MUTEX_UNLOCK | Mutex Un-Lock macro | All | The Mutex unlock macro is defined in target_defines.h. Base this macro on the mutex you are using. Defined in Windows as LeaveCriticalSection, POSIX as pthread_mutex_unlock; and VxWorks as semGive |
| CALLBACK | Used with a timer callback function when in Windows | Windows | For other system define as blank |
| WORD<br>DWORD<br>UINT<br>HWND<br>LPSTR<br>LPINT<br>LPWORD<br>LPLONG<br>LPDWORD<br>LPVOID | These are defined in Windows. | UNIX, VxWorks, Integrity | typedef unsigned short WORD;<br>typedef unsigned long DWORD;<br>typedef unsigned int UINT;<br>typedef int HWND;<br>typedef char * LPSTR;<br>typedef int * LPINT;<br>typedef WORD * LPWORD;<br>typedef long * LPLONG;<br>typedef DWORD * LPDWORD;<br>typedef void * LPVOID; |
| LARGE_INTEGER | A Microsoft structure used for a 64 bit unsigned integer. | UNIX and VxWorks | LARGE_INTEGER is used with the ADD_TRACE capability. If you want to include this debugging tool into your API, you need to define LARGE_INTEGER as a signed 64-bit integer. |
| VOID | Void | ALL | Define VOID as void |
| timeGetTime | Windows and provides a timeGetTime function that returns the current time in milliseconds | UNIX, VxWorks, Integrity | You need to provide a timeGetTime function that returns time in milliseconds. The wrapper versions are in the time.c file. |

## 15.4 Windows Calling Conventions

When compiled for Windows, all API functions use the **stdcall** or **WINAPI** calling conventions. Consequently, you can call the busapi32.dll from any Windows application that supports those calling conventions, such as LabVIEW, and Visual Basic. These applications do not need to re-compile the API to use it.

## 15.5 Porting to an Unsupported Operating System

BusTools/1553-API library source is written in C, not C++. Although it is standard C, there might be certain constructs in the code you need to change when migrating to a unsupported compiler. You must also assure all include files are valid for your system.

### 15.5.1 Endian Issues

The API supports operation on Little Endian and Big Endian systems. By default, the API is Little Endian. When compiling for a Big Endian system assure the API does all bit- and byte-swapping correctly. Due to variations in operating systems and compilers this may be inconsistent across Big Endian systems. You can define several macros that will convert a Big Endian value to Little Endian.

- NON_INTEL_WORD_ORDER – flips 32-bit words to and from Big Endian
- NON_INTEL_BIT_FIELDS – Reorders certain bit fields in structures.
- WORD_SWAP – Flip every 16-bit access to the board.

The BusTools/1553-API defines NON_INTEL_WORD_ORDER and NON_INTEL_BIT_FIELDS for RQVME2-1553, and QVME-1553 on the PowerPC processor, and defines WORD_SWAP for the PMC-1553 on a PowerPC system.

When porting the API to a new platform, make sure you understand all the Endian issues and correctly set the Endian conversion symbols. You must consider the *Endianess* for both the host processor and the target Bus. For example, the VME bus is Big Endian and the PCI bus is Little Endian.

### 15.5.2 Conversion Steps

The sections that follow detail the following conversion issues:

- Choosing an initial environment.
- Target definition
- In-line Intel assembly language.
- Syntactical issues.
- Bit fields word ordering
- Structure alignment and byte/word accesses.
- Mapping the board into host memory.
- Interrupt support

### Choosing an Initial Environment

BusTools/1553-API supports the following target environments:

- Windows
- Linux
- VxWorks
- LynxOS

- Integrity

The first decision to make is which environment most closely resembles your target environment. The target_defines.h included by Busapi.h, defines the labels that control the compilation targets. Consider the following items for porting to a non-supported platform.

- When compiling for Windows, define _WIN32 or WIN32.
- When compiling for non-Intel environments, or compilers that do not support in-line x86 assembly, define "NO_ASSEMBLY".
- When compiling for UNIX environments, define _UNIX_. This generates a build targeted to an UNIX-like system. It will substitute UNIX system calls for Windows system calls. This includes all systems using GNU compilers including VxWorks.
- When running VxWorks, define VXWORKS.
- When running Integrity, define _INTEGRITY_.
- When compiling for an environment which defines bit fields reversed from the Intel ordering, define the label NON_INTEL_BIT_FIELDS.
- When running PCI bus on a PowerPC, define WORD_SWAP. This macro maintains proper word ordering.
- When compiling for processors that order words reversed from the Intel ordering, define the label NON_INTEL_WORD_ORDER. This enables the "flip" macro that swaps the two words of a DWORD when they cross a 32-bit boundary.
- Always ensure that the compiler does not pad structures (e.g., specify two-byte structure alignment). You need to ensure that your system has the correct structure size for all structures. Table 15-5 shows the required structure sizes. You need to ensure that the size of these structures in your code matches the sizes in this table.
- When compiling for an environment that supports a file system, you can define the label "FILE_SYSTEM" to include the diagnostic and memory dump output capabilities of the API (BusTools_DumpMemory, et al.).
- When compiling for systems that support hardware interrupts, define HW_INTERRUPTS.

## Target Definition

Target_defines.h has define-blocks for the supported processors and operating systems. The following is an example of a define block for a PCI boards running on a Linux x86 system.

```
/*******************************************************
 * Target defines for Linux running for the PCI Boards.
 *******************************************************/
#if defined(_LINUX_X86_)  /*                          */
  #define _UNIX_          /*                          */
  #define _GCC_           /* Using GCC compiler       */
  #undef _Windows         /*                            */
  #undef __WIN32__        /*                            */
  #define _LINUXBOARDSETUP/*                            */
  #undef INCLUDE_VMIC     /* supports the VMIC VME      */
  #undef _PLAYBACK_       /*                            */
  #define FILE_SYSTEM     /* Add file systems functions*/
  #undef DEMO_CODE        /* API DEMO version           */
  #undef  NON_INTEL_BIT_FIELDS  /* Intel Bit Ordering  */
  #undef  NON_INTEL_WORD_ORDER  /* Intel Word Ordering */
  #undef  ADD_TRACE       /* function call trace code  */
  #undef DO_BUS_LOADING   /* bus loading code           */
  #undef _USER_INIT_      /*                            */
  #undef _LABVIEW_        /*                            */
  #undef _USER_DLL_       /*                            */
  #define _PLAYBACK_      /*                            */
  #undef INCLUDE_VME_VXI_1553 /*                        */
  #define INCLUDE_PCCD    /*                            */
  #include <unistd.h>     /*                            */
  #include <string.h>     /*                            */
  #define MSDELAY(p1) usleep(p1*1000) /*Sleep marcro   */
  /* the following are the O/S specific definitions for*/
  /* malloc, free threads, mutexes and events          */
  #include <pthread.h>    /* pthread for Interrupts    */
  #define CEI_MALLOC(a) malloc(a) /* memory alloc      */
  #define CEI_FREE(a)   free(a)   /*memory free        */
  #define CEI_MUTEX pthread_mutex_t /* mutex type      */
  #define CEI_THREAD pthread_t      /* thread type     */
  #define CEI_EVENT pthread_cond_t  /* event variable  */
  #define CEI_MUTEX_LOCK(a) pthread_mutex_lock(a)/*    */
  #define CEI_MUTEX_UNLOCK(a) pthread_mutex_unlock(a)/**/
  #define CEI_THREAD_EXIT(a) pthread_exit(0) /*        */
#endif /* end _LINUX_X86_                               */
/*******************************************************/
```

## Inline Intel Assembly Language

Inline Intel assembly language has an alternate implementation in portable “C” code. The “C” implementation is selected when the “NO_ASSEMBLY” label is defined.

## Syntax Issues

While the API source is written in “C”, it uses a mix of “C” and “C++”style comments (“//”) throughout the code. If you use the –ansi directive with GNU C, the compiler does not recognize // comments.

Another syntax issue is the use of system-include files. The code uses both the PC platform set of .h files and the UNIX (gnu C) set of include files. Select the set of include files the API uses by setting the macros in the target_defines.h file. These two include file options aren’t necessarily compatible with other compilers. If you are using a non-supported compiler, you may need to modify the include files to work with your compiler.

## Bit Fields and Word Ordering

Many non-Intel-targeted compilers number the bits in the bit fields opposite to that required by the Windows (Little Endian) platform. Defining the compile-time symbol “NON_INTEL_BIT_FIELDS”, causes the bit numbering to be reversed to agree with these targets.

Many non-Intel platforms reverse the ordering of words within a quantity. The “flip” macro is activated when the compile-time symbol “NON_INTEL_WORD_ORDER” is defined.

Define “NON_INTEL_WORD_ORDER” if you are running on a Big Endian processor. Further, you may need to define the WORD_SWAP macro if you are running a PowerPC or other Big Endian processor with a PCI bus.

## Structure Alignment and Byte/Word Accesses

The BusTools/1553-API uses two types of structures, host and on-board. The host structures use “API” in their name (API_BC_MBUF for example). These host structures pass information between the API and the application software. The alignment of these structures is not critical to execution, so long as you compile all modules with the same alignment. This is not true of the board structures.

Board structures pass information between the API and the 1553 interface board. The API needs the compiler to align and pack these structures on a two-byte boundary. If this alignment is off, the API doesn’t execute correctly. Make sure that the board structures match the size in Table 15-5.

BusTools_BIT_StructureAlignmentCheck is a BIT function that checks for the structure alignment in your application. You can use this function in the early application development to check for alignment problems and remove later when alignment is correct.

Table 15-5 Structure Size for Key BusTools/1553-API Structures for Boards Running V5 or Earlier Firmware

| Structure Name | Size in Bytes |
|---|---|
| BC_MESSAGE | 48 |
| BC_CBUF | 48 |
| BC_DBLOCK | 68 |
| BM_CBUF | 8 |
| BM_MBUF | 168 |
| BM_FBUF | 4096 |
| BM_TBUF_ENH | 106 |
| EI_MESSAGE | 66 |
| RT_ABUF_ENTRY | 8 |
| RT_ABUF | 256 |
| RT_CBUF | 6 |
| RT_CBUFBROAD | 126 |
| RT_FBUF | 4096 |
| RT_MBUF_HW | 88 |
| RT_MBUF_API | 8 |
| RT_MBUF | 96 |
| BT1553_TIME | 8 for API v8.0x and 6 for earlier API revisions. |
| IQ_MBLOCK | 6 |

Table 15-6 Structure Size for Key BusTools/1553-API Structures for Boards Running V6 Firmware

| Structure Name | Size in Bytes |
|---|---|
| BC_V6MESSAGE | 36 |
| BC_V6CBUF | 36 |
| BC_V6DBLOCK | 68 |
| BM_V6CBUF | 8 |
| BM_V6MBUF | 172 |
| BM_V6FBUF | 8192 |
| V6MBUF_HW | 96 |
| BM_V6TBUF_ENH | 92 |
| EI_V6MESSAGE | 66 |
| RT_V6ABUF_ENTRY | 8 |
| RT_V6ABUF | 256 |
| T_V6CBUF | 8 |
| RT_V6CBUFBROAD | 132 |
| RT_V6FBUF | 8192 |
| RT_V6MBUF_HW | 96 |
| RT_MBUF_API | 8 |
| RT_V6MBUF | 104 |
| BT1553_TIME | 8 |
| IQ_V6MBLOCK | 8 |

## Mapping the Board into Host Memory

BusTools/1553-API controls the board through direct pointer access. This means that the API requires only a pointer to the board's base address. Some boards require just a pointer to memory while other boards require a pointer to memory and a configuration space. Check either the "MIL-STD-1553 UCA Reference Manual" or the "UCA32 Global Register Reference Manual" for the mapping requirements for your 1553 interface, depending on the firmware version programmed on the board.

When porting to non-supported systems, you need to supply the API with a device driver or a similar function that can provide the pointer to the board. You also need a low-level interface. The low-level interface provides a consistent interface between the API and the mapping function.

You need to have the following two functions for all board types.

- vbtMapBoardAddresses
- vbtFreeBoardAddresses

You also need to provide:

- vbtGetPCIConfigRegister (for Native PCI boards or a stub)

lowlevel.h has the function prototypes for these routines.

The process of mapping the board into memory depends on the board you are using and the operating environment. Abaco Systems uses a third-party driver for Windows operating systems and supplies a device driver for Linux and LynxOS systems. The API also has the following low-level interface files:

- **lowlevel.c**: Windows. This is a proprietary file and only the lowlevel.obj file is on the distribution.
- **mem.c**: Used for all board types under Linux and LynxOS.
- **mem_vxWorks.c**: Used for all boards and supported Board Support Packages.
- **mem_integrity.c**: Used for all board and supported Board Support Packages.

You can use one of these files as the basis for any new development. They all include lowlevel.h.

## Interrupt Support

BusTools/1553-API supports interrupt processing via invocation of the function BusTools_RegisterFunction. This function allows the user to provide an interrupt callback function that runs when the specified interrupt event occurs. You can implement this feature on unsupported systems or modify how interrupts run on supported systems by providing code for the CEI interrupt templates. These templates are functions, MACROS, or data types. Fill in the code to provide the functionality needed. The following table lists the templates.

| Name | Type | Description |
|---|---|---|
| CEI_THREAD_DESTROY | function | Terminates thread and events |
| CEI_THREAD_CREATE | function | Create thread and events |
| CEI_WAIT_FOR_EVENT | function | Wait (blocks) for specified event |
| CEI_EVENT_SIGNAL | function | Signal waiting thread of event |
| CEI_MUTEX_LOCK | MACRO | Lock mutex |
| CEI_MUTEX_UNLOCK | MACRO | Unlock mutex |
| CEI_THEAD_EXIT | MACRO | Thread exit macro |
| CEI_THREAD | type | Type definition for thread |
| CEI_EVENT | type | Type definition for event |
| CEI_MUTEX | type | Type definition for mutex |

Currently the API supports hardware interrupts for Windows, POSIX and VxWorks. These are implemented in 3 files, CEI_WIN_INTERRUPT_FUNCTIONS.c, CEI_POSIX_INTERRUPT_FUCNTIONS.c, and CEI_VXW_INTERRUPT_FUNCTIONS.c. There is a CEI_INTERRUPT.h file containing the function prototypes for the interrupt functions. The macros and data type are defined in target block in target_defines.h. You can use these predefined interrupt functions or use the file CEI_template_INTERRUPT_FUNCTION.c provided in the source directory and fill in missing code with the interrupt processing required.

## 15.6 Rebuilding the BusTools/1553-API Library

Abaco Systems provides the source code to the BusTools/1553-API library as part of the standard distribution. This allows you to understand how the API works and allows customization the API if desired. The Windows BusTools/1553-API projects for Microsoft Development Studio are located under the respective toolset directory in the installed directory tree based on the host operating system:

for Windows 7 and later versions:

C:\Users\Public\Documents\Condor Engineering\BusTools-1553-API\Windows API Projects

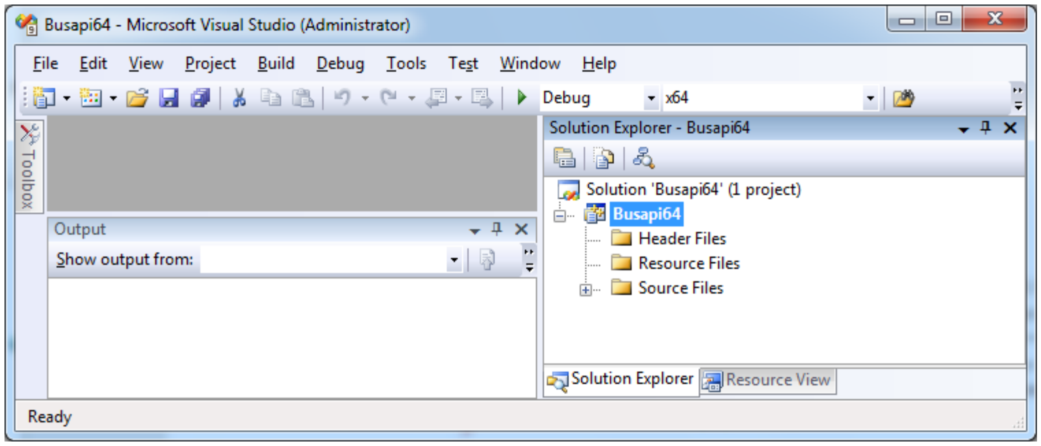
for Windows XP:

C:\Program Files\Condor Engineering\BusTools-1553-API\Windows API Projects

For Microsoft Visual C++ 6, open the project for Busapi32.dll via double-click on the Busapi32.dsw (workspace) file in the MSVS 6.0\Busapi32 directory.

For Microsoft Visual Studio 2008, open the 64-bit API library project via double-click on the solution file Busapi64.sln located in the MSVS 2008\Busapi64 directory.  The following example build is for Microsoft Visual Studio 2008:

You can build the Busapi32.dll by clicking on the Build icon or pressing the F7 key.

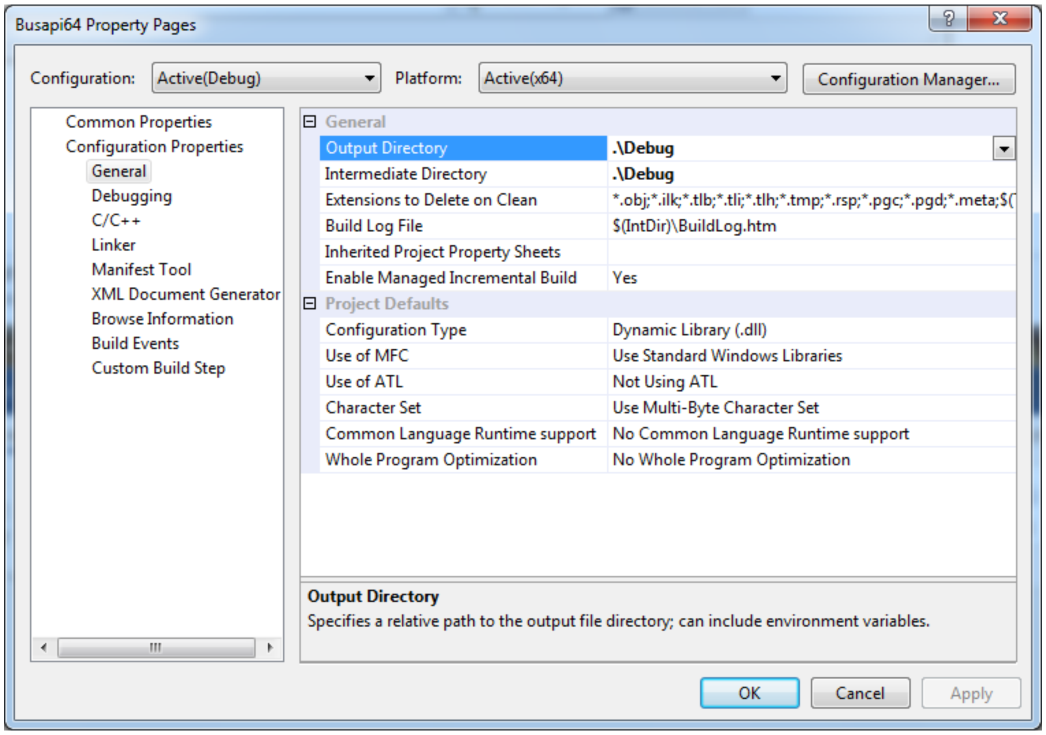

Busapi64 - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Tools Test Window Help
Debug
x64
Toolbox
Output
Show output from:
Solution Explorer - Busapi64
Solution 'Busapi64' (1 project)
Busapi64
Header Files
Resource Files
Source Files
Solution Explorer
Resource View
Ready

You can customize this build by changing the Project Option settings.

### 15.6.1 Changing Project Options

1. Right-click on the project ***Busapi64***, select ***Properties***… to access the following dialog.

Figure 15-1 Visual Studio Project Propertis



Busapi64 Property Pages

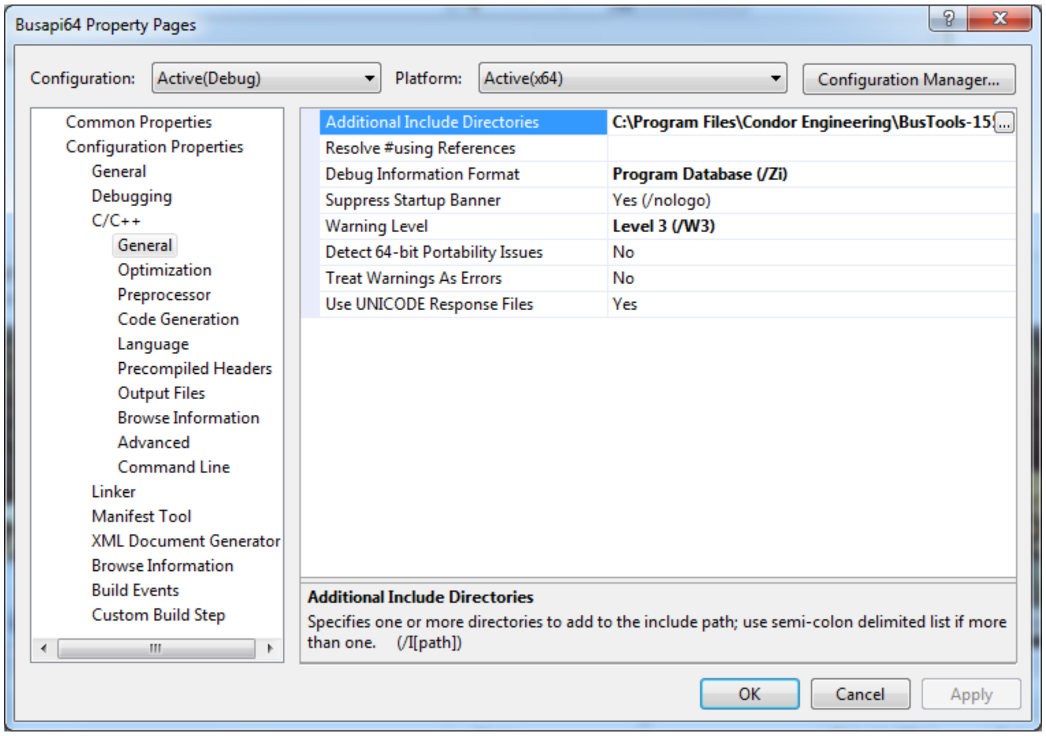
Configuration: Active(Debug) Platform: Active(x64) Configuration Manager...

| | |
|---|---|
| **General** | |
| Output Directory | .\Debug |
| Intermediate Directory | .\Debug |
| Extensions to Delete on Clean | *.obj;*.ilk;*.tlb;*.tli;*.tlh;*.tmp;*.rsp;*.pgc;*.pgd;*.meta;$( |
| Build Log File | $(IntDir)\BuildLog.htm |
| Inherited Project Property Sheets | |
| Enable Managed Incremental Build | Yes |
| **Project Defaults** | |
| Configuration Type | Dynamic Library (.dll) |
| Use of MFC | Use Standard Windows Libraries |
| Use of ATL | Not Using ATL |
| Character Set | Use Multi-Byte Character Set |
| Common Language Runtime support | No Common Language Runtime support |
| Whole Program Optimization | No Whole Program Optimization |

**Output Directory**
Specifies a relative path to the output file directory; can include environment variables.

2. To change the path to the API header files: Under the ***Configuration Properties -> C/C++*** category, select the ***General*** subcategory and enter the appropriate path in the ***Additional Include Directories*** property.

Figure 15-2 Visual Studio Project Compiler Include File Path Properties



Busapi64 Property Pages

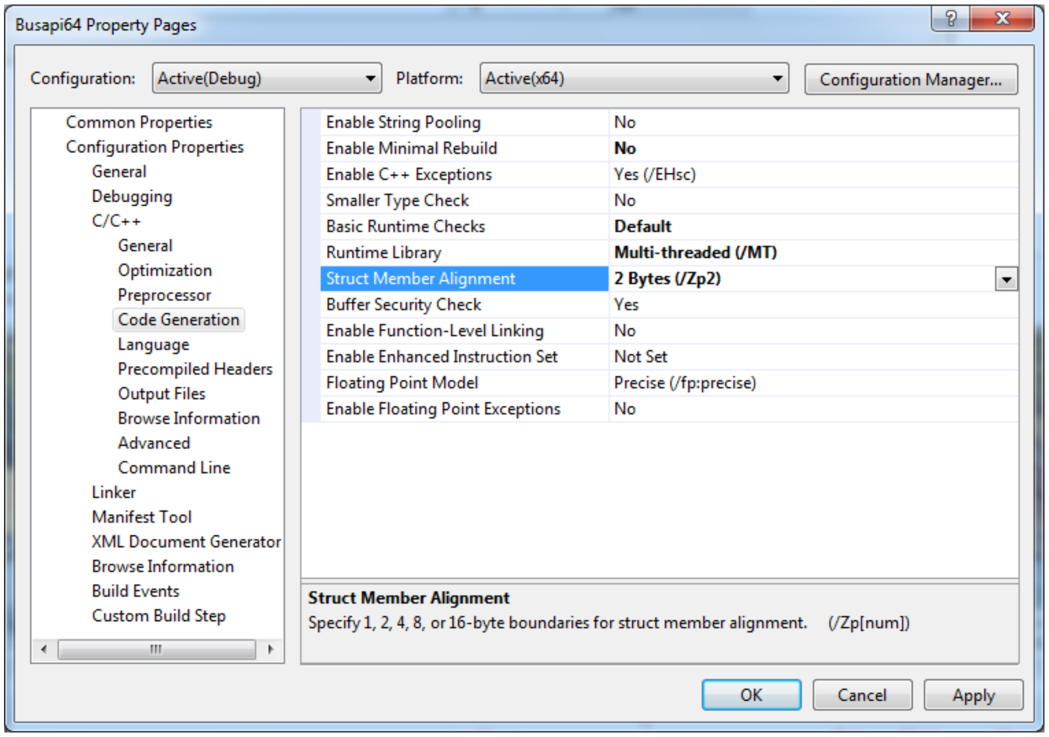
Configuration: Active(Debug) Platform: Active(x64) Configuration Manager...

| | |
|---|---|
| Additional Include Directories | C:\Program Files\Condor Engineering\BusTools-15 |
| Resolve #using References | |
| Debug Information Format | Program Database (/Zi) |
| Suppress Startup Banner | Yes (/nologo) |
| Warning Level | Level 3 (/W3) |
| Detect 64-bit Portability Issues | No |
| Treat Warnings As Errors | No |
| Use UNICODE Response Files | Yes |

**Additional Include Directories**
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

3. Build the API with 2-byte alignment: Under the ***Configuration Properties -> C/C++*** category, select the ***Code Generation*** subcategory and modify the ***Struct Member Alignment*** property.

Figure 15-3 Visual Studio Project Compiler Code Generation Properties


Busapi64 Property Pages

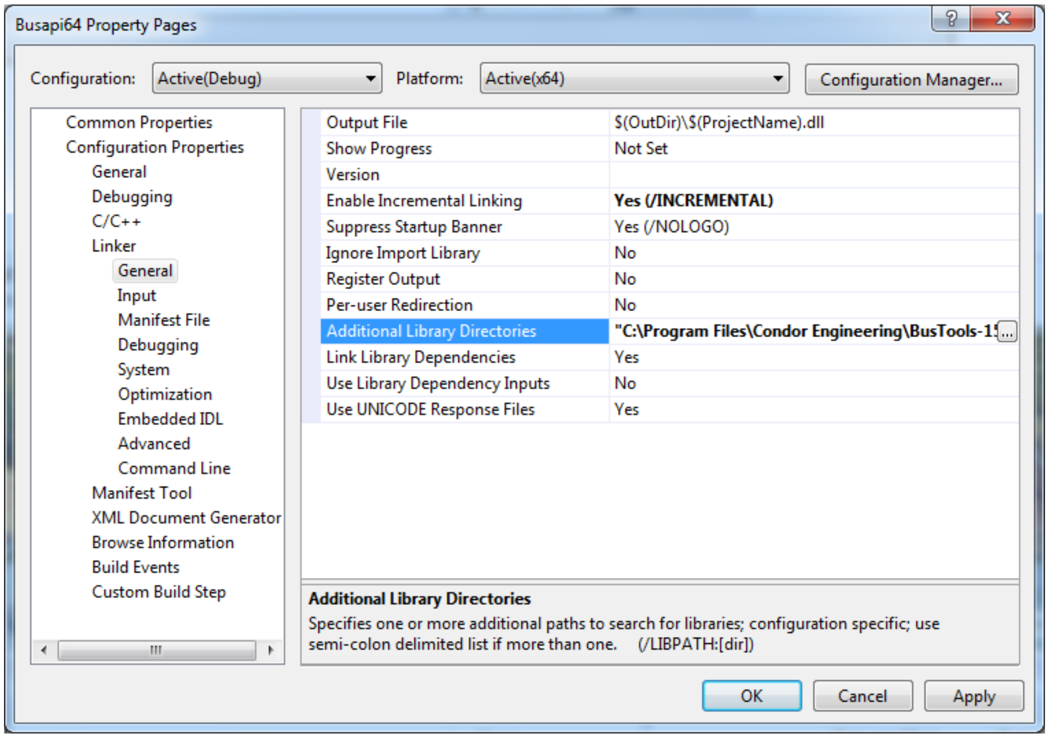
Configuration: Active(Debug) Platform: Active(x64) Configuration Manager...

| | |
|---|---|
| Enable String Pooling | No |
| Enable Minimal Rebuild | **No** |
| Enable C++ Exceptions | Yes (/EHsc) |
| Smaller Type Check | No |
| Basic Runtime Checks | **Default** |
| Runtime Library | **Multi-threaded (/MT)** |
| Struct Member Alignment | **2 Bytes (/Zp2)** |
| Buffer Security Check | Yes |
| Enable Function-Level Linking | No |
| Enable Enhanced Instruction Set | Not Set |
| Floating Point Model | Precise (/fp:precise) |
| Enable Floating Point Exceptions | No |

**Struct Member Alignment**
Specify 1, 2, 4, 8, or 16-byte boundaries for struct member alignment. (/Zp[num])

4. The precompiled Lowlevel.obj and usb.obj files are included in the build from ***Configuration Properties -> Linker*** category, select the ***General*** subcategory and enter the appropriate path in the ***Additional Library Directories*** property.

Figure 15-4 Visual Studio Project Linker Directories File Path


Busapi64 Property Pages

Configuration: Active(Debug) Platform: Active(x64) Configuration Manager...

| | |
|---|---|
| Output File | $(OutDir)\$(ProjectName).dll |
| Show Progress | Not Set |
| Version | |
| Enable Incremental Linking | **Yes (/INCREMENTAL)** |
| Suppress Startup Banner | Yes (/NOLOGO) |
| Ignore Import Library | No |
| Register Output | No |
| Per-user Redirection | No |
| Additional Library Directories | **"C:\Program Files\Condor Engineering\BusTools-1** |
| Link Library Dependencies | Yes |
| Use Library Dependency Inputs | No |
| Use UNICODE Response Files | Yes |

**Additional Library Directories**
Specifies one or more additional paths to search for libraries; configuration specific; use semi-colon delimited list if more than one. (/LIBPATH:[dir])

### 15.6.2 Building the DLL

Use the build icon, press the ‘F7’ key or under ***Build*** menu, select ***Build Busapi64*** or ***Rebuild Solution***. This compiles the files and builds the 64-bit DLL. The Busapi64.dll builds without any errors or warnings.

## 15.7 Building a Unix Platform Shared Library

The Linux make will compile the API as a shared library (libbusapi.so). Under LynxOS it is built as a static library (libbusapi.a). Re-build the library and install it into the proper directory by navigating to the bt1553 directory in the install directory tree and typing ‘make’.

# Glossary

| | |
|---|---|
| **1553** | A component or message in accordance with MIL-STD-1553. |
| **1773** | A component in accordance with MIL-STD-1773, which is an optically coupled version of MIL-STD-1553. |
| **Windows** | An operating system developed by Microsoft. |
| **API** | Application Programmer's Interface. A defined and documented software interface, which permits software written by one person or organization to interact with the software written by another person or organization without requiring either party to know the details of the implementation of the other's software. |
| **BC** | Bus Controller. One of three possible devices that may be connected to a MIL-STD-1553 bus. Determines the message traffic on a 1553 bus. |
| **BC-RT** | A 1553 message that transfers data from the BC to a RT. Also called a RT Receive message. |
| **BIOS** | Basic Input/Output System. The resident software that initializes the computer hardware and provides low-level access to some of the computer components. |
| **BM** | Bus Monitor. One of three possible devices that may be connected to a MIL-STD-1553 bus. A passive monitor, which cannot create or request traffic on a 1553 bus. |
| **Broadcast** | A class of 1553 messages characterized by multiple receivers and one sender. Broadcast messages are directed to the broadcast Remote Terminal number (31) but are actually received and processed by all Remote Terminals on the bus. |
| **Broadcast BC-RT** | A specific 1553 message directed to RT address 31, where all RTs receive the data sent by the BC, and that they do not respond with a status word. |
| **Broadcast Mode Code** | A class of 1553 messages, where a mode code is directed to RT address 31. This causes all RTs on the bus to process the message, and not respond with a status word. |
| **Broadcast RT-BC** | A message type that is not defined or permitted on a MIL-STD-1553 bus system. |
| **Broadcast RT-RT** | A specific 1553 message, where two command words are transmitted by the BC, and that the first command word tells all RTs to listen. The second command word instructs a specific RT to ignore the listen command and to transmit data. |
| **BSP** | Board Support Package. Software used by embedded systems to setup the hardware on a specific processor board. Roughly equivalent to the BIOS on a PC or PC/AT. |
| **cPCI** | Compact version of the PCI interface. See PCI below. |
| **DLL** | Dynamic Link Library. A stand-alone library of software functions that may be used by an application. |
| **FPGA** | Field Programmable Gate Array |
| **include file** | A file with an extension of ".h", used by "C" programmers to contain function and data structure definitions that are shared among various program modules. |
| **Linux** | UNIX based operating system for PCs |
| **Microcode** | The instructions for the programmable element of the 1553 interface contained in the WCS. |
| **Microsecond** | 1/1000000 (millionth) of a second. Abbreviated as µs. |
| **Millisecond** | 1/1000 (thousandth) of a second. Abbreviated as ms. |
| **MIL-STD-1553** | A military communication standard that specifies the interconnection of one Bus Controller, multiple Remote Terminals, and optionally, one or more Bus Monitors, into an integrated communication system. |
| **MIL-STD-1773** | An optically coupled version of MIL-STD-1553. |
| **Mode Code** | A class of 1553 messages, using Subaddress 0 or 31, and with the word count interpreted as the mode code number. Mode codes have zero or one data word, depending on the mode code number. While all word counts are potentially valid, only a subset of the possible mode codes is valid, as specified by the standard. |
| **Operating System** | The software that operates the computer, such as Windows or Linux. |
| **OS** | Operating System. The software that operates the computer. |
| **PC** | Personal Computer. A specific type of host computer based on the Intel architecture. |
| **PCI** | Peripheral Component Interconnect. A board-level communication bus used in Personal Computers (and other computer systems) based on the PCI Specification from the PCI Special Interest Group. |

| | |
|---|---|
| **PCIe** | Peripheral Component Interconnect Express. A board-level communication bus used in Personal Computers. |
| **Playback** | The ability to regenerate MIL-STD-1553 message traffic on the physical bus using data that was previously recorded. |
| **PMC** | PCI Mezzanine Card. A slim modular mezzanine card based on the PCI specification. |
| **RT** | Remote Terminal. One of three possible devices that may be connected to a MIL-STD-1553 bus. Responds to a Bus Controller. |
| **RT Number** | The address of a specific RT. A value between 0 and 30, with 31 being reserved for the Broadcast function. |
| **RT Receive** | A 1553 message that transfers data from the Bus Controller to a Remote Terminal. Also called a BC-RT message or just a Receive message. |
| **RT Transmit** | A 1553 message that transfers data from a Remote Terminal to the Bus Controller. Also called a RT-BC message or just a Transmit message. |
| **RT-BC** | A 1553 message that transfers data from a RT to the BC. Also called a RT Transmit message. |
| **RT-RT** | A class of 1553 messages, where there are two command words transmitted by the BC. The first command tells a specific RT to listen for data, the second command word instructs another RT to transmit data. The BC is neither the source nor destination for the data. |
| **Subaddress** | The address within a RT that acts as the source or destination of a specific message. Subaddresses 1 through 30 are used for messages, SA 0 and 31 are reserved for mode codes. |
| **USB** | Universal Serial Bus, a type of computer port which can be used to connect external equipment to a computer. |
| **WCS** | Writeable Control Store |
| **Window** | A functional component of an application. A display. |
| **Windows** | Any one of several Windows operating systems supplied by Microsoft corporation. |
| **XMC** | Express Mezzanine Card.  A slim modular mezzanine card based on the PCI Express specification. |

LINK

See the Abaco Glossary Reference Manual for more terms and acronyms.

## Abaco Systems Information Centers

Americas:
1-866-652-2226 (866-OK-ABACO)
or 1-256-880-0444 (International)

Europe, Middle East and Africa:
+44 (0)1327 359444

## Additional Resources

For more information, please visit the Abaco Systems web site at:

www.abaco.com

Publication No. 1500-045 Rev. 5.11