Software Reference Manual

BusTools/1553-API

Publication No. 1500-038 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 Hardware Revision: 6.11/6.09/6.08/6.03/5.18/4.6x/4.4x Rebranding to the Abaco Systems format.
5.12	Dec 2019	BusTools/1553-API Software Revision: 8.28 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 uses the following types of notice:

CAUTION

Cautions alert you to system danger or loss of data.

NOTE

Notes call attention to important features or instructions.

TIP

Tips give helpful hints on procedures that may be tackled in a number of ways.

LINK

Links take you to other documents or websites. The blue link color may also be used within a body of text or paragraph to indicate a link (or hyperlink) to a different part of the same document.

Terms

Windows – References to "Windows" refer to all supported Microsoft Windows® Operating Systems.

BusTools/1553 Hardware – Refers to any Abaco Systems 1553 board 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 post-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 channels. This document refers to those as Channel 1, Channel 2, Channel 3, and Channel 4.

Channel Initialization – In the functions provided by the BusTools-1553/API, the statement "Prior to calling this function, the 1553 channel must be initialized by calling one of the initialization functions." This statement refers to the functions BusTools_API_OpenChannel and BusTools_API_InitExtended. For PCI, PCI Express, and PCMCIA devices, BusTools_API_OpenChannel is recommended. For VME and VXI devices, BusTools_API_InitExtended is required.

Device – Represents the Abaco Systems 1553 board installed in the system, which may be configured with one or more channels. Channels on the board are independently initialized and programmed except for interrupts, IRIG, discretes and

Differential I/O, which are shared between the channels on the board. The user application is responsible for coordinating the use of IRIG, discretes, and differential I/O among the channels on a board. There is a single hardware interrupt for each Abaco Systems 1553 board, no matter how many channels are actively in use. For high-level applications, the API determines which channel issued an interrupt and invokes the appropriate user function(s).

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.

Additional C Sample Programs

There are additional sample programs included that demonstrate how to use the API to program and control Abaco Systems 1553 interface boards. These programs are found in the following folder after the API has been installed:

For Windows 7 and later

C:\Users\Public\Documents\Condor Engineering\BusTools-1553-API\Examples\C\C Example Code\examples

For Windows XP

C:\Program Files\Condor Engineering\BusTools-1553-API\Examples\C\C Example Code\examples

Example application source for non-Windows operating systems include tstall.c for UNIX and Integrity, and VxW_Demo for VxWorks.

Not all example code compiles and runs directly on every operating system. Refer to Appendix A, "Sample Programs" for the descriptions of each sample program and BusTools/1553-API functions used in the sample program.

QuikView1553 – The API distribution provides various installation test example applications that will execute with installed 1553 board(s). QuikView1553 is an application executable designed specifically for Windows systems. Execute QuikView1553 from the desktop on Windows platforms, invoked via the QuikView1553 icon. QuikView1553 provides the ability to test all installed Abaco Systems 1553 boards.

Further Information

Abaco Systems Manuals

This document is distributed via the internet. You may register for access to manuals via the website whose link is given below.


LINKS
https://www.abaco.com

Abaco Website

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


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

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 User's Manual


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

You can find technical assistance contact details on the web site Support page.

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

Abaco will log your query on 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 web site 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

List of Tables

List of Figures

1 • Introduction

This manual is a reference guide for BusTools/1553-API, the software distribution supporting the Abaco Systems MIL-STD-1553 Application Programming Interface (API). This software distribution provides high-level software control of the MIL-STD-1553 products, allowing rapid development of custom applications.

1.1 Operating Systems Supported

This BusTools/1553-API software distribution supports the following products with a common interface on a range of form factors and operating systems, with installation instructions provided in the *BusTools/1553-API Software User's Manual*:

- **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** – all products
- **64-bit/32-bit Linux (kernels 5, 4, 3, 2.6, 2.4)** – all products with support for PCMCIA boards limited to kernels before 2.6.15
- **64-bit/32-bit VxWorks 7, 6.x** – all products except PCMCIA boards.
- **64-bit/32-bit Integrity 11, 5.10** – all products except PCMCIA and AMC boards.

Support for the VME and PCMCIA products are only supported on 32-bit operating systems. The API is also supported with National Instruments® LabVIEW and LabVIEW R/T via Abaco System's LV-1553 product.

1.2 Interface to Other Languages

BusTools/1553-API is written in the "C" programming language. Abaco Systems supplies all source files, example programs, and documentation for using the BusTools/1553-API with languages other than "C". The "BusTools/1553-API Software User's Manual" provides information regarding use of the API with C#.

1.3 API Source Code

The BusTools/1553-API distribution includes all source code, both for reference and to allow adaptation to other platforms and operating systems. Programs written for Windows can link directly to the libraries provided in the distribution. For Linux and LynxOS, the libraries are built as part of the distribution deployment on the host. For VxWorks and Integrity, the API source must be built into a library and/or kernel image for use on the target platform.

1.4 Supported Hardware

The following table shows the boards supported by BusTools/1553-API. A detailed description of all products is available at the Abaco link below.

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

Table 1-1 Abaco Systems 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

1.5 Hardware Features

Abaco Systems MIL-STD-1553 boards have many features such as IRIG, discretes, DMA and triggers that are optionally available on the 1553 interface boards. The table below shows some of the features available on each 1553 interface board.

Table 1-2 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	Programmable	No	No	No	Program discretes
RPCIe-1553	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
MPCIe-1553	2	Yes	No	Programmable	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
R15-PMC	Optional	Optional	Optional	Optional	No	No	No	Optional
RXMC-1553	4 dedicated + 8 Discrete or PIO (optional)	4 optional	optional	Programmable	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	Yes	Order 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

2 • API Initialization and Global Routines

This chapter describes the initialization, shutdown, and general setup operations required by the hardware and API functions. *Global* functions adjust parameters that affect the entire board (such as IRIG or Discrete operation) or parameters that affect the characteristics of an entire channel (such as the 1553 bus coupling method or enabling external bus communications).

2.1 API Initialization

BusTools_API_OpenChannel and **BusTools_API_InitExtended**: These functions are used to open a session and initialize a channel on any Abaco Systems 1553 device. For all PCI-bus, PCI-Express bus and PCMCIA 1553 devices, BusTools_API_OpenChannel is recommended. For VME and VXI supported devices, BusTools_API_InitExtended is required.

This respective open session function is the first API function called when programming a channel on a 1553 device, configuring firmware registers on the board and hardware interface data structures in the API for the selected channel. You must verify that BusTools_API_OpenChannel or BusTools_API_InitExtended returns a status of API_SUCCESS before continuing to program a channel.

The following functions are also commonly called for general channel initialization.

BusTools_SetBroadcast: This function controls how RT address 31 is handled. RT address 31 is reserved for "broadcast" messages, however, it is possible to operate the board with "broadcast" operations disabled. In this case, RT address 31 is treated like a standard RT address.

BusTools_SetInternalBus: This function selects internal or external bus operation. By default, the API initializes the board to run on its internal 1553 bus to ensure that no traffic goes out until the application is ready to transmit. Call this function to enable the external bus after initialization (after invoking the respective BusTools_XX_Init function but before calling any of the BusTools_XX_StartStop functions. **Note** that it is possible to run internally, without using the external bus. On a multi-function card, you can run a complete 1553 topology on the internal bus.

BusTools_SetOptions: This function is used to set how the board handles various options. This function has changed significantly since BusTools/1553-API version 5.24. Several previously supported options were deleted, and several new options were added. This function supports:

- Suppress the Minor-Frame overflow warning
- Memory Dump on BM Stop
- Monitor Invalid Commands (not the same as Monitor Illegal Commands for RT discussed below). When enabled this option allows the Bus Controller to monitor

invalid commands. When disabled the Bus Controller will discard invalid commands. Added to API version 5.90 and later, and available with Firmware version 4.19 and later.

- BM Trigger on Message
- Trigger on synchronize Mode Code
- RT start on external sync
- Ignore High Word. This setting provides the fastest response time by disabling the firmware logic checking for high word count errors.
- Monitor Illegal Commands: *Removed for API revisions after 5.24*. This option specifies how an RT should respond to an illegal command (one with an invalid subaddress or word count). If the option is enabled, the RT returns a status word to the BC with the Message Error bit set (and suppresses or ignores any data words). If the option is disabled, the RT responds to the command word as required (receiving or transmitting data), but the data words are invalid. The Message Error bit is not set. The default setting for this parameter is disabled.
- Interrupt on Illegal Commands: *Removed for API revisions after 5.24*. This option specifies the action to take if an RT detects an illegal command (one with an invalid subaddress or word count). If the option is enabled, the RT generates an interrupt if an illegal command is detected. The default setting for this parameter is disabled.
- Treat Illogical Commands as Invalid: *Removed for API revisions after 5.24*. Illogical commands are undefined mode code commands. They should never be transmitted by the BC. If an illogical command is received, it may be treated as an illegal command or as an invalid command. If this option is enabled, illogical commands are treated as invalid (the command is ignored, and no message is processed by the RT). If this option is disabled, illogical commands are treated as illegal (the RT response depends on the setting of the "Monitor Illegal Commands" option). The default for this parameter is enabled.
- Reset Time tag on Sync: This option resets the board's internal timer if a Sync mode code is detected. Since the timer is global to all components simulated on the board, this option should only be used if the board is simulating a single RT. The default for this parameter is disabled (the Sync Mode Code has no effect on the board timer).

BusTools_SetSa31: According to MIL-STD-1553, the board translates messages to or from subaddress 0 as mode code messages. The specification provides the option of translating messages to or from subaddress 31 as mode code messages. The default setting for this parameter is to translate messages to or from subaddress 31 mode code messages. The setting for this feature can be modified by invoking this function after initialization, but before calling the RT initialization function BusTools_RT_Init.

BusTools_SetVoltage: This function sets the voltage level and electrical coupling for the board driving the external 1553 bus. The application can use either "direct

coupling" or "transformer coupling" for driving the 1553 bus. For each of these possible electrical connections, the board drives the 1553 bus at the specified voltage level. This option is used to test external hardware with low voltage situations. The default setting for "direct coupled" is 6.5 volts, for "transformer coupled" is 19.8 volts. Call this function after board initialization but before calling any of the BusTools_XX_StartStop functions. Not all products support software selectable coupling or variable transmit amplitude. Refer to the Hardware Installation Manual's section on your board for more information.

BusTools_TimeTagMode: This function is used to set the time tag mode and the display format for message time tags.

2.2 API Shutdown

The BusTools_API_Close function performs all operations necessary to shut down the API and release resources for the channel referenced. Not calling this function at the close of your application can leave resources allocated. This could affect other processes in the system. Multiple calls to BusTools_API_Close are harmless, and simply return a code indicating that the API has already been closed.

2.3 Global Parameter Routines

Most API function calls affect only the channel referenced by the card number parameter. However, functions for IRIG, Avionics Discretes, and Differential I/O are global to the board. There is a single IRIG input, control, and calibration setup. IRIG is selected on a channel-by-channel basis, but there is only one IRIG timer. All channels on a board share common Avionics Discretes and Differential I/O lines. All channels access the same discrete/differential I/O registers so take care in how you allocate and program these channels. The following list shows these functions.

2.3.1 IRIG-B Functions

BusTools_IRIG_Calibration

BusTools_IRIG_Config

BusTools_IRIG_SetTime

BusTools_IRIG_Valid

2.3.2 Discrete and Differential I/O Functions

BusTools_DiffTriggerOut

BusTools_DiscreteGetIO

BusTools_DiscreteRead

BusTools_DiscreteSetIO

BusTools_DiscreteTriggerIn

BusTools_DiscreteTriggerOut

BusTools_DiscreteWrite

BusTools_RS485_TX_Enable

BusTooks_RS485_Set_TX_Data

BusTools_RS485_ReadReg

Several parameters control global operation of the hardware and firmware for a specific channel. These parameters are set or reset by calling the functions described below. While there are default values for each of these parameters, it is a good idea to call these functions with the desired settings to ensure known, repeatable behavior. Call these functions after initializing the board, but before calling any other BusTools-API function that uses 'cardnum' as an input argument.

2.4 General Purpose Routines

BusTools_StatusGetString: All API functions return a status code indicating if the function executed successfully. Successful completion of the function is indicated by the code API_SUCCESS (zero). If the status is non-zero, an error was detected. Call this function to obtain a text string description of the error code. All status codes returned by BusTools API functions are supported, even those created after this manual was last updated.

BusTools_DataGetString: This function performs engineering unit conversions and places the result in a string. This function can be called at any time.

BusTools_TimeGetString: This function is used to convert a BusTools time structure into an ASCII character string for display. This time structure is contained in BM and RT message buffers. This function can be called at any time. Note, this function uses 64-bit arithmetic and may not be compatible with operating systems not supporting 64-bit arithmetic. Starting with F/W version 6.0, time resolution is 1 nanosecond. In order to convert a nanosecond resolution time-tag you must pass the characters "NANO" in the string that will receive the converted time.

3 • Application Development

3.1 Developing Applications in Supported Environments

The following table shows all supported environments and the required definitions to compile BusTools/1553-API for those operation systems.

Table 3-1 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
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.

3.2 Common Header Files

The following are files common to all supported operating systems:

- **Busapi.h**: This is the header file associated with BusTools/1553-API. It contains all structure definitions, subfunction prototypes, and parameter definitions required to build applications incorporating the API. Include this file in any "C" source file that accesses API functions.
- **Target_defines.h**: Busapi.h includes the file target_defines.h. This file has target-specific information the API needs to build under different environments. It requires compiler defines that may or may not be automatically defined in most enviroments.
- **cei_types.h**: This include file defines the common data types used across the Abaco Systems avionics software distributions.

3.3 Developing Windows Applications

When developing Windows applications, use the following files:

- **Target_defines.h**: It requires compiler defines of _WIN32 or WIN32 for any Windows operating system.
- **Busapi32.lib/Busapi64.lib**: These are the Windows API dynamic link library (DLL) files for 32-bit and 64-bit Windows operating systems, respectively. They contain all API function exports contained in the respective DLL file. Follow your development tool's directions for adding library files to your application project.
- **Busapi32.dll/Busapi64.dll**: These are the 32-bit and 64-bit BusTools/1553-API DLL files containing the executable version of the API functions. During installation these files are installed in the corresponding Windows system directory.
- **cei_install.dll**: This DLL provides initialization support for Windows applications. During installation, it is copied into the corresponding Windows system directory.

3.4 Developing Linux and LynxOS Applications

When developing Linux and LynxOS applications, there are specific defines applicable to the respective processor (see Table 3-1).

- **Libbusapi.so**: Link this library file with your application program. There are two library options, a shared library, libbusapi.so, and a static library, libbusapi.a. Each link the same way by using -lbusapi as a command line argument to the compiler or in your Makefile. The only difference between the two libraries is that the shared library does not require you to re-compile your application if you re-build the BusTools/1553-API.
- **Libceill.so**: Linux only. This library contains the common low-level library for all Abaco Systems APIs. Link this along with libbusapi.so.

3.5 Developing VxWorks Applications

When developing VxWorks applications, there are specific defines applicable to the respective processor (see Table 3-1). There are also specific device driver options, as described in the "VxWorks Support" chapter of the *BusTools/1553-API User's Manual*.

3.6 Developing Integrity Applications

When developing Integrity applications, there are specific defines applicable to the respective processor, see Table 3-1. There are also specific device driver options, as described in the "Integrity Support" chapter of the *BusTools/1553-API User's Manual*.

3.7 General Development Notes

- Be sure to set your compiler to generate "2-byte alignment" of structures. The normal default is typically 4- or 8-byte alignment; this alignment does *not* interface with the BusTools library correctly.
- Be sure to add the appropriate library in the project link command.
- Be sure to use the latest version of Busapi.h when re-compiling your program, since various structures and calling sequences may have been changed. Using the latest version gives the compiler a chance to flag any changes that might affect your program.
- Make sure that you specify the correct target define block within target_defines.h before re-compiling the API. See the *BusTools/1553-API User's Manual* for details. This ensures that all the correct defines are set for the target OS. Using the target define block allows you to customize the API build to include or exclude API features that you may or may not need.
- When porting to a non-supported system, you can start with a target define block similar to your target system, then modify according to the description in the *BusTools/1553-API User's Manual*. Define the unique identifier for that target define block when you build your application. For example, by defining "VXW_PCI_PPC", you compile for a PMC or QPMC on a PowerPC by selecting the block associated with that target.
- Verify that the calling convention that is being used by your program matches the library (as defined by the Busapi.h file). This is especially important if you are using another language, such as Visual Basic, which does not use the Busapi.h interface file. The current Busapi.h file is always up to date, while the information in this chapter was current only at the time this manual was released.
- The error codes listed for each function are the most typical errors returned by that function. Additional error codes are possible. See Chapter 6, "Return Codes" for a complete listing of all possible error codes. The Busapi.h file from the API software distribution contains the current list of error codes, since error codes are regularly added to the API. All error codes can be converted to printable ASCII messages by calling the BusTools_StatusGetString function.

For more details, see the *BusTools/1553-API Software User's Manual*.

3.7.1 Number of Channels Supported

BusTools/1553-API supports the number of channels in any combination of boards specified by the parameter MAX_BTA. This value is set for each target environment in target_defines.h. The default value is 64 for Windows and 16 for any other operating system.

3.7.2 API Disk Space Requirements

Disk requirements vary depending on the application. You should have ample disk space to install the API and support software and data. Operation of the BusTools/1553 GUI requires additional disk space. Bus Topology files (.btd) take only a few KBytes, but the Bus Monitor files (.bmd or .bmdx) can be large. The size of these files depends on the complexity and speed of the bus traffic recorded and can easily grow to several hundred megabytes or more. Make sure there is ample disk space before recording data with the Bus Monitor. If the recording fills the disk, disk recording stops, and the data is lost.

3.7.3 Operating System Requirements

BusTools/1553-API supports 32-bit and 64-bit Windows, Linux, LynxOS, VxWorks, and Integrity. When developing time critical and real-time applications, you must consider the effect of the operating system on performance. Operating systems such as Windows or UNIX can cause delays in application processing that can make real-time programming difficult. If you have requirements for deterministic timing or want to process high data rates, you may want to consider operating systems with smaller delays (e.g., Linux) or a real-time operating system (e.g., VxWorks or Integrity).

3.8 Hints and Tips

- If you have a single-function 1553 interface, you can run only one operational mode (BC, BM, or RT) for each channel at one time. Any attempt to start multiple functions results in an API_SINGLE_FUNCTION_ERR (231) warning.
- If you have a dual-function 1553 interface, you can only run the RT or BC along with the BM. Any attempt to run the BC and the RT together results in API_DUAL_FUNCTION_ERR (234) warning.
- The function call that initializes the board may return an error if the mode variable (flag or flag) is set to 0 (API_DEMO_MODE).
- After initialization, “Internal Bus” is selected by default. To ensure that 1553 bus traffic appears external to the Abaco Systems board, select “External Bus”. See “BusTools_SetInternalBus”.
- Most API functions affect only the channel addressed by the card number. However, there are functions for IRIG, Avionics Discretes, and Differential I/O that are global to the board. There is a single IRIG input, control, and calibration setup. You can select whether you are using IRIG on a channel-by-channel basis, but there is only one IRIG timer. Similarly, there are a set number of Avionics Discretes and Differential I/O lines on a board that are shared by all channels. When calling these functions, all channels access the same discrete registers so take care not to allow one channel to overwrite other channels’ configuration.
- There is a single interrupt per device for PCI, PCIe, Mini-PCIe, Express Card, and AMC devices. When a single application is running all channels on a device, the BusTools/1553-API and the associated driver determines which channels on the

board are generating interrupts and notifies the corresponding interrupt callback function, If there are multiple applications running those channels, it is recommended that only one of those applications utilize hardware interrupts. The other application(s) should use software (timer-based) interrupts.

- Several parameters control global operation of the hardware and firmware on a channel. These parameters are set or reset by calling API functions. While there are default values for each of these parameters, it is a good idea to call these functions with the desired settings to ensure known, repeatable behavior.
- Each of the message buffers defined for the board (BC/BM/RT) contain a word called the "interrupt status word". This 32-bit word is a composite of the status of all the words in a message. "Interrupt" because it contains information which is valid after a message is completed. It contains bus errors, information about the message including message type, retry, error conditions, and message completion.
- The function BusTools_DumpMemory reads the contents of board memory and writes it to a text file. This information provides a snapshot of the board setup and operation for debugging an application. This function is only available for target systems that have a file system.
- The *BusTools/1553-API User's Manual* contains descriptions of code examples for setting up each mode (BC/BM/RT), as well as many other topics. Refer to the example source code for demonstrations regarding the use of the data structures and functions needed to set up each mode of operation.

4 • BusTools/1553-API Routines

This chapter describes each function in the BusTools/1553-API library. For each function, its operation is described, the calling sequence is defined, and a list of return status codes is presented. The status codes are listed mnemonically. A description of each status or error code is provided in Chapter 6, "Return Codes".

4.1 BusTools_API_Close

Description

BusTools_API_Close terminates BusTools/1553-API operations by releasing all resources allocated by the API, terminating interrupt service and timer functions and stopping any registered user threads. You must call this function before exiting a program using BusTools/1553-API functions. Failure to call this function can have unpredictable effects. Multiple calls to this function are harmless.

OS Support

Core API Function

Syntax

wStatus = BusTools_API_Close (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel as defined by the application as the *cardnum* parameter in the invocation of BusTools_API_InitExtended, or as the value returned in the *chnd* pararmeter upon return from the invocation of either BusTools_API_OpenChannel or BusTools_API_OpenDeviceChannel.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.2 BusTools_API_InitExtended

Description

BusTools_API_InitExtended initializes VME and VXI bus 1553 boards. It supports the following boards:

- QVME-1553
- RQVME2-1553
- QVXI2-1553X

Initialization allocates the resources required by the API and sets up default parameters for both the API and the board. Interrupt service processing is set up if supported.

This initialization function is required to initialize any Abaco Systems 1553 VME/VXI interface card under the various supported operating systems. The API relies on the user-supplied values in this function to specify the specific board type used.

The *cardnum* parameter designates the 1553 interface/channel initialized by BusTools_API_InitExtended. All subsequent calls to the API use this number as a logical channel reference to that 1553 channel. The *cardnum* parameter should start at 0 and increment by 1 for each 1553 channel session initiated in the system. Many Abaco Systems 1553 boards have multiple 1553 channels and each channel will be assigned a unique *cardnum* value.

The *flag* parameter is used to define one of these states:

a) API_DEMO_MODE (0x0) – "software-only" or H/W simulation.

b) API_SW_INTERRUPT (0x01) – normal operation using software polling. See BusTools_API_SetPolling to change the poll time.

c) API_HW_INTERRUPT (0x02) – normal operation, enabling hardware interrupts in addition to software polling.

d) API_HW_ONLY_INT (0x03) – normal operation, enabling hardware interrupts only.

e) API_MANUAL_INT (0x04) – No interrupt processing

To select MIL-STD-1553A operation, add the value API_A_MODE (0x80) to *flag*, (MIL-STD-1553B is the default).

OS Support

Core API function

Syntax

wStatus = BusTools_API_InitExtended (cardnum, base_address, ioaddr, flag, platform, boardType, carrier, slot, mapping);

wStatus — (BT_INT) status returned from this function.

cardnum — (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

base_address — (BT_U32BIT) the A24 or A32 base address for the memory window allocated to the VME/VXI device.

ioaddr — (BT_UINT) the A16 base I/O address selected by jumpers on the board.

flag — (BT_UINT*) pointer to a flag used to specify the operational mode for the channel. OR the Operational Mode and Interrupt Mode from the values below:

1553 Operational Mode:

API_A_MODE	- MIL-STD-1553A
API_B_MODE	- MIL-STD-1553B

Interrupt Mode:

API_SW_INTERRUPT	- S/W Polled Interrupt
API_HW_ONLY_INT	- H/W Only Mode
API_HW_INTERRUPT	- H/W and S/W Mode
API_MANUAL_INT	- No interrupt processing

platform — (BT_UINT) execution platform: PLATFORM_VMIC or PLATFORM_USER.

boardType — (BT_UINT) QVME1553 or RQVME2 (use RQVME2 for the QVXI2-1553X).

carrier — (BT_UINT) NATIVE

slot — (BT_UINT) channel select, valid options are:

- CHANNEL_1
- CHANNEL_2
- CHANNEL_3
- CHANNEL_4

mapping — (BT_UINT) carrier memory map, valid options are:

- CHANNEL_1
- CARRIER_MAP_DEFAULT
- CARRIER_MAP_LARGE
- CARRIER_MAP_A32
- CARRIER_MAP_A24

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_INITED
BTD_CHAN_NOT_PRESENT
BTD_ERR_BADCFG
BTD_ERR_BADDETECT
BTD_ERR_PARAM
BTD_NEW_SERIAL_PROM
BTD_NO_SUPPORT
API_OUTDATED_FIRMWARE
API_HARDWARE_NOSUPPORT

Notes

The errors listed above are the most common errors detected by BusTools_API_InitExtended. This function also detects and reports other errors. See Chapter 6, "Return Codes", for a complete listing of all error codes. See the following section for Initialization Examples and more detailed notes.

Variable Voltage boards are initialized to full output voltage and transformer coupling by this function. See the function BusTools_SetVoltage to change these defaults.

The RQVME2-1553 is a RoHS redesign of the QVME-1553. You can use either QVME1553 or RQVME2-1553 for initialization.

BusTools_API_OpenChannel is recommended for all other form-factors installed on Windows, Linux, VxWorks, and Integrity platforms.

4.2.1 User Supplied External Addressing Mode

The Windows version of BusTools/1553-API supports an externally supplied addressing mode. This mode supports hardware interfaces or bus repeaters that are not supported by the normal API device driver by using a user-supplied board mapping function in place of the API's built-in low-level device driver.

You must name this external function "BusTools_API_InitExternal" and locate it in a user-specified DLL. The API calls this external function during initialization of the board. The API expects it to return the addresses of the various areas of the specified board or an error code.

During shutdown of the board, the API again calls the specified DLL function. This gives the DLL a chance to release any mapping resources and return the system to its initial state.

For a complete programming description of this function, see the function definition "BusTools_API_InitExternal".

4.2.2 Initialization Examples

This section explains how to set the BusTools_API_InitExtended parameters to initialize a Abaco Systems 1553 interface channel. Many Abaco Systems 1553 boards have multiple channels. This function must be called for each channel.

Function Prototype:

```
wStatus = BusTools_API_InitExtended (cardnum,
          base_address, ioaddr, flag, platform,
          boardType, carrier, slot, mapping);
```

1. The *cardnum* input argument is a channel-based index defined by the application that represents a handle to the respective 1553 interface channel for BusTools_API_InitExtended. The value of this parameter should be used as the "*cardnum*" handle with all other BusTools/1553-API function calls to program the respective channel. *cardnum* should bein with the value 0 and increment by 1 for each 1553 interface channel initialized, regardless of the number of channels installed on each of the target 1553 boards.
2. The *flag* parameter is used to define one of these states:
3. API_DEMO_MODE (0x0) – "software-only" or H/W simulation.
4. API_SW_INTERRUPT (0x1) – normal operation using software polling. See BusTools_API_SetPolling to change the poll time.
5. API_HW_INTERRUPT (0x2) – normal operation, enabling hardware interrupts in addition to software polling.
6. API_HW_ONLY_INT (0x3) – normal operation, enabling hardware interrupts only.
7. API_MANUAL_INT (0x4) – No interrupt processing
8. API_VXW_HLI (0x10) – (VxWorks Only) Use this when initializing VxWorks to run with high level (BusTools_RegisterFunction) interrupts.
9. 'OR' *flag* with API_A_MODE to select 1553A operation.
10. Setting *platform* to "PLATFORM_USER" allows you to supply a custom initialization DLL. The RQVME2-1553/QVME-1553 and QVXI2-1553X use this option when running with National Instruments VXI libraries. Abaco Systems supplies the BTVXIMAP.DLL for this option. When you select "PLATFORM_USER", pass the DLL name in the mapping parameter. "PLATFORM_USER" applies to Windows systems only.
11. Use the *carrier* option to specify the VME address space as A24 or A32; use NATIVE or NATIVE_24 for A24, NATIVE_32 for A32.
12. For the *slot* parameter, BusTools/1553-API uses "channel" and "slot" interchangeably. Use the channel parameter (i.e., CHANNEL_1, CHANNEL_2, etc.) for most boards. For single channel boards only CHANNEL_1 is valid.

4.2.3 RQVME2/QVME : One, Two or Four-channel Native Boards

The QVME-1553/RQVME2-1553 are native VME boards. The QVXI2-1553X is an RQVME2-1553 installed in a VXI sleeve. These boards are supported on Windows systems with the National Instruments MXI-2 interface and on VxWorks systems using the legacy VxWorks VME driver interface.

base_address	The base address of the memory window for the board's host interface, either A24 or A32. This is programmable.
ioaddr	Address of A16 configuration space (ID * 0x40 +0xc000 where ID is set by onboard jumpers.
platform	Windows systems use PLATFORM_USER; for VxWorks use PLATFORM_PC.
boardType	QVME-1553
carrier	NATIVE, NATIVE_24, NATIVE_32 (NATIVE and NATIVE_24 for A24 addressing, NATIVE_32 for A32 addressing).
slot	(BT_UINT) the channel on the device to reference, valid options are: CHANNEL_1, CHANNEL_2, CHANNEL_3, CHANNEL_4
mapping	(BT_UINT)("BTVXIMAP.DLL") for Windows, CARRIER_MAP_A24 or CARRIER_MAP_A32 for VxWorks.

The following example initializes the first 1553 interface channel on a QVME-1553. Initializing with BTVXIMAP.DLL uses National Instruments NI-VXI libraries. The user specifies A32 addressing by using the carrier type NATIVE_32. The address 0x01000000 is programmable and must not interfere with other VME devices. The wIOaddress value is based on the JB1 jumper settings. In this case, this jumper block is set to 0x0f.

Example 1: Initialize Channel 1 on a QVME-1553

```
cardnum = 0; // start at 0

Windows

Status = BusTools_API_InitExtended (0, 0x01000000,
          0xc3c0, &flag, PLATFORM_USER, QVME1553,
          NATIVE_32, CHANNEL_1,
          (BT_UINT)("BTVXIMAP.DLL"));

VxWorks

Status = BusTools_API_InitExtended (0, 0x08000000,
          0xc3c0, &flag, PLATFORM_PC, QVME1553,
          NATIVE_32, CHANNEL_1, CARRIER_MAP_A32);
```

4.3 BusTools_API_InitExternal

Description

Supported under Windows only, this function is a user authored function defined in a user-provided Dynamic Link Library (DLL). This function is used when the built-in board mapping function supplied by the API is insufficient to access the board. This provides a mechanism for extending the API to support those cases.

The invocation of this function occurs during the execution of BusTools_API_InitExtended, when specified by the invoking application. The name of the DLL that contains the function BusTools_API_InitExternal is passed as a parameter to BusTools_API_InitExtended.

The *cardnum* parameter designates the Abaco Systems 1553 board to be opened; the value passed into BusTools_API_InitExtended is passed to this function.

The address, wIOaddress, platform, cardType, carrier and slot values passed into BusTools_API_InitExtended are passed transparently to this function.

This function must compute and return the following addresses:

For the PCI/PMC boards:

- PageAddr[0] maps the board base address
- PageAddr[1…3] are not used

For the VME boards:

- PageAddr[0] maps the A24/A32 board base address
- PageAddr[1] maps the A16 base address.
- PageAddr[2…3] are not used.

For all of the IP-D1553/carrier combinations:

- PageAddr[0] maps the IP-D1553 dual-port memory.
- PageAddr[1] points to the I/O register.
- PageAddr[2] contains the ID PROM address, or the ID PROM revision level.
- PageAddr[3] is not used.

The API uses these addresses directly to access the board. It does not attempt to translate them in any manner. If any of these addresses are incorrect, the result is undefined.

The API also calls this function when the API function BusTools_API_Close is called. This gives the function a chance to terminate gracefully and release any resources it might have allocated. The exact function prototype is contained in the Busapi.h file.

OS Support

Windows

Syntax

wStatus = BusTools_API_InitExternal (cardnum, wOpen, address, wIOaddress, cardType, carrier, slot, PageAddr);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) BusTools card number (0-based).
address	(BT_U32BIT) base memory address of the board from BusTools_API_InitExtended.
wIOaddress	(BT_U32BIT) base I/O address of the board from BusTools_API_InitExtended.
wOpen	(BT_UINT) Flag, 1=open device, 0=close device.
cardType	(BT_UINT) board type from BusTools_API_InitExtended.
carrier	(BT_UINT) board type from BusTools_API_InitExtended.
slot	(BT_UINT) slot from BusTools_API_InitExtended.
PageAddr[4]	*(void *) pointer to an array of pointers to void which receives the base addresses of the board as computed by this function.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_INITED
BTD_ERR_NOACCESS
API_NO_OS_SUPPORT

Notes

Your program is responsible for returning API_SUCCESS (and correctly setting the PageAddr pointers) or returning an error code. See Chapter 6, "Return Codes", for a listing of all error codes translated to strings by the BusTools_StatusGetString function. The user function can return one of these error codes, or it may define a new error code.

4.4 BusTools_API_OpenChannel

Description

BusTools_API_OpenChannel opens a session to and initializes a MIL-STD-1553 channel on the specified device. The application must initialize a channel before invoking other BusTools/1553-API functions containing a *cardnum* in its parameter list.

This function supports the following 1553 devices on Windows and Linux, and most PCI/PCIe devices on VxWorks, and Integrity systems:

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

To initialize a MIL-STD-1553 channel on a specific 1553 board, the application will invoke this function with the assigned device number of the 1553 board and the desired channel on that board. For Windows systems, the device number of all 1553 boards is assigned during installation. For all other operating systems, the device number is determined by the device discovery order on the PCI/PCIe bus. If there is a single Abaco Systems board installed in your system, it is always device 0. If there are multiple devices installed in your system, use BusTools_FindDevice to obtain the device number for the respective 1553 board type, or determine the fixed configuration of the boards and their device index value as installed on the host.

The value in the *chnd* parameter returned from this function is a **logical channel reference** to the session opened for the respective *devid* (board) and *channel* values provided by the application. On subsequent BusTools/1553-API function calls, use the *chnd* parameter value as a reference the respective channel via the *cardnum* parameter.

During initialization, channels on boards supporting variable voltage are initialized to maximum output voltage by this function. Channels on boards supporting programmable transformer/direct coupling are initialized to transformer coupling by this function.

OS Support

BusTools_API_OpenChannel Core API Function

Syntax

wStatus = BusTools_API_OpenChannel (chnd, mode, devid, channel);

wStatus (BT_INT) status returned from this function.

chnd (BT_UINT *) Pointer to an unsigned integer. If successful, this value represents a logical channel reference used to reference the session opened with the specified *channel* and *devid* combination. Use this value for all channel-based BusTools/1553-API function calls requiring a "card number" as a parameter. Valid range is 0 to 63.

mode (BT_UINT) a flag used to specify the operational mode for the channel, defined as a combination of the desired MIL-STD-1553 Operational Mode and the desired Interrupt Mode. The *mode* is defined by the application as a logical OR of the following field values:

1553 Operational Mode:

API_A_MODE	- MIL-STD-1553A
API_B_MODE	- MIL-STD-1553B

Interrupt Mode:

API_SW_INTERRUPT	- S/W Polled Interrupt
API_HW_ONLY_INT	- H/W Only Mode
API_HW_INTERRUPT	- H/W and S/W Mode
API_MANUAL_INT	- No interrupt processing

devid (BT_UINT) the device number of the board on which the desired 1553 channel is located. For Windows operating systems the device number is assigned during installation, for all other operating systems this value is assigned in the PCI/PCIe bus device discovery order. The valid range for *devid* is operating system dependent; see the value assigned to MAX_BTA in target_defines.h for the respective o/s.

channel (BT_UINT) the channel on *devid* to reference, valid options are:

CHANNEL_1
CHANNEL_2
CHANNEL_3

CHANNEL_4

Return Value

API_SUCCESS
API_BUSTOOLS_INITED
API_INIT_NO_SUPPORT
API_INSTALL_INIT_FAIL
API_BAD_PRODUCT_LIST
API_BAD_DEVICE_ID
API_CHANNEL_OPEN_OTHER
API_HARDWARE_NOSUPPORT
API_MAX_CHANNELS_INUSE
API_NO_BUILD_SUPPORT
API_OUTDATED_FIRMWARE
BTD_CHAN_NOT_PRESENT
BTD_ERR_BADCFG
BTD_ERR_BADDETECT
BTD_ERR_INUSE
BTD_ERR_NOMEMORY
BTD_ERR_PARAM
BTD_NEW_SERIAL_PROM
BTD_NO_SUPPORT

Notes

This function detects and reports many other errors related to opening a session and initializing a board/channel. The errors listed above are the most common. See Chapter 6, "Return Codes" for a complete listing of all error codes.

The API function BusTools_FindDevice can be used to determine the device number of a specific Abaco MIL-STD-1553 board. You can embed an invocation of this function into BusTools_API_OpenChannel invocation as shown below to open a session to a channel on a specific board type:

```
mode = API_B_MODE | API_SW_INTERRUPT;

status = BusTools_OpenChannel(&cardnum, mode,
BusTools_FindDevice(PCI1553,2),CHANNEL_1);
```

where the invocation to BusTools_FindDevice as shown returns the device number for the second instance of a PCI-1553 installed in your system.

4.5 BusTools_API_ShareChannel

Description

BusTools_API_ShareChannel configures a previously initialized channel for sharing by other applications/processes. Channel sharing allows one application to initialize and share a channel, and other applications to join with the shared channel. Only multi-function boards support channel sharing. This feature is restricted to a single application/process for each 1553 function, Bus Controller, Remote Terminal and Bus Monitor, on any single channel, limited to a single instance of each 1553 function.

Channel sharing permits you to create independent applications for Bus Controller, Bus Monitor, and Remote Terminal functions all running on the same channel. When a channel is shared, the channel mapping and status data is written to a common memory location on the board. When other channels join, they initialize using the memory mapping and status information stored in the common memory.

Once a channel is configured for sharing, another application joins that channel by calling BusTools_API_JoinChannel. Applications joining a shared channel must not initialize the channel. Initialization will overwrite the existing mapping and corrupt the sharing operation.

The application that initializes and shares a channel can use any available interrupt mode, software, hardware, or hardware only. The joining channels are only allowed to use software interrupts. Before a channel can be shared it must first be initialized using one of the initialization functions, BusTools_API_OpenChannel or BusTools_API_InitExtended.

OS Support

Core API Function

Syntax

wStatus = BusTools_API_ShareChannel (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel. Valid range is 0 to 63.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINTED
API_HARDWARE_NOSUPPORT
API_SINGLE_FUNCTION_ERR
API_CHANNEL_SHARED

4.6 BusTools_API_JoinChannel

Description

BusTools_API_JoinChannel allows an application to join a shared channel. The channel specified by the channel parameter on device wDevice, must already be initialized by another application prior to calling this function. You need to coordinate with the initializing application to ensure the channel is initialized.

When opening a session to a shared channel, this function invocation replaces the open/initialization function. A joining application must not call any initialization functions; instead, the preexisting channel parameters are acquired from a common memory location on the board.

Once an application joins a channel, the application can program 1553 functions using BusTools/153-API. Since applications joining shared channels do not have access to hardware interrupts, they can only utilize the Software Interrupt mode.

Channel sharing allows only a single instance of each supported 1553 function: Bus Controller, Remote Terminal and Bus Monitor; with a maximum of three applications permitted to share a given channel. For example, an application can initialize and share a channel then configure and execute as a Bus Controller. Another application can join, configure, and execute as one or more Remote Terminals, while yet another application can join, configure, and execute as a Bus Monitor.

OS Support

Core API Function

Syntax

wStatus = BusTools_API_JoinChannel (chnd, device, channel);

wStatus	(BT_INT) status returned from this function.
chnd	(BT_UINT *) channel reference to the respective 1553 board/channel previously established via prior session open and initialization and used in an invocation of BusTools_API_ShareChannel. Valid range is 0 to 63.
device	(BT_UINT) The device number of the board on which the desired 1553 channel is located. For Windows operating systems his number is assigned during installation, for all other operating systems this value is assigned in the PCI/PCIe bus device discovery order. The valid range for *devid* is operating system dependent; see the value assigned to MAX_BTA in target_defines.h for the respective o/s.

channel (BT_UINT) the channel on the *device* to reference, valid options are:

CHANNEL_1
CHANNEL_2
CHANNEL_3
CHANNEL_4

Return Value

API_SUCCESS
API_CHANNEL_NOTSHARED
API_HARDWARE_NOSUPPORT
API_MAX_CHANNELS_INUSE
API_NO_INTERRUPT_SUPPORT
BTD_ERR_NOACCESS
BTD_ERR_NOMEMORY

4.7 BusTools_API_QuitChannel

Description

BusTools_API_QuitChannel terminates a session with a joined or shared channel, invoked in place of BusTools_API_Close for all joined channel sessions and prior to the invocation of BusTools_API_Close for the sharing channel session. When calling this function, pass an indicator via *qFlag* that defines the terminating application's 1553 function implemented on the channel (BC/BM/RT mode). For the initial sharing channel application reference, it should also indicate termination of the shared session by adding the value SHARE_QUIT to the implemented function value in the *qFlag* parameter.

OS Support

Core API Function

Syntax

wStatus = BusTools_API_QuitChannel (cardnum, qFlag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
qFlag	(BT_UINT) termination flag referencing the 1553 function the application implemented on the shared/joined channel session: RT_QUIT BM_QUIT BC_QUIT SHARE_QUIT

Return Value

API_SUCCESS
API_CHANNEL_NOTSHARED

4.8 BusTools_API_LoadUserDLL

Description

This function opens the specified user interface DLL and maps all of the user-defined entry points. This gives the user "hooks" into the operation of the API, without having to directly modify the API source code and re-compile the API library.

See Chapter 5 "Extending the API", for a complete discussion about this function.

OS Support

Windows

Syntax

wStatus = BusTools_API_LoadUserDLL (cardnum, szDLLName);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
szDLLName	(const char *) Pointer to a comma-delimited list of DLL names that are opened sequentially

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_CANT_LOAD_USER_DLL
API_NO_OS_SUPPORT

4.9 BusTools_BC_AperiodicRun

Description

All Abaco Systems 1553 boards support aperiodic message transmission by the Bus Controller function, refer to the manual section "Aperiodic 1553 BC Messages" in the *BusTools/1553-API Software User's Manual* for a detailed explanation of the setup, transmission, and limitations surrounding aperiodic message transmit queues.

Prior to invoking this function, initialize the channel using one of the Initialization functions and initialize the Bus Controller with invocations of both BusTools_BC_Init and BusTools_BC_MessageAlloc. Additionally, invoke BusTools_BC_MessageWrite for each aperiodic message in the *messageid* list, to initialize the messages.

BusTools_BC_AperiodicRun supports two execution methods:

1. Block execution until the aperiodic message queue is empty
2. Return immediately after initiating periodic message processing

When execution is blocked while aperiodic messages are being processed, this function will not return until the last message in the queue begins transmission. Since the last message has not completed transmission on return, the application should refrain from modifying the content within the aperiodic message queue until transmission is complete. For this reason, it is advised to insert a NO-OP message as the last message in the aperiodic message queue.

When execution is not blocked, this function will return immediately after initiating periodic message processing. In this operating mode the function BusTools_BC_AperiodicTest can be invoked to determine if transmission of the respective message queue has been completed.

OS Support

Core API Function

Syntax

```
wStatus = BusTools_BC_AperiodicRun ( cardnum, messageid, Hipriority, WaitFlag,
          wWaitTime );
```

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messageid	(BT_UINT) the first BC message number indicating the beginning of the aperiodic message list ("0" based).
Hipriority	(BT_UINT) priority to assign to the aperiodic message, valid options are: 1 = High Priority, 0 = Low Priority.

WaitFlag — (BT_UINT) 1 = request this function blocks execution until the channel's BC function completes processing the aperiodic message list; 0 = this function will not block execution; instead, it will return from processing aperiodic messages immediately after initiating the transaction.

wWaitTime — (BT_UINT) duration to wait for the channel to complete processing the aperiodic message, in milliseconds. Valid range is 1 to 4294967295. This parameter is ignored if *WaitFlag* is zero.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTRUNNING - BC is not running and wait was specified
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_APERIODIC_RUNNING
API_BC_APERIODIC_TIMEOUT

Notes

The High Priority and Low Priority message lists are independent; both can be active at the same time.

For current 1553 products, the minor frame counter resolution is 1µs (microsecond). For legacy V5.x and earlier firmware, the minor frame counter resolution is 25µs. This should be considered when assigning a value to *WaitTime*.

The *messageid* passed to this function is the first message in a list of messages. You can use any legal message, conditional, Stop BC or No-op in this list. The list can contain any number of messages. The only restriction in the message list is the value 0xFFFF must be applied the next message pointer in the last message.

Messages passed to this function must NOT have either the BC_CONTROL_MFRAME_BEG or the BC_CONTROL_MFRAME_END bits set.

See the function definitions for "BusTools_BC_MessageAlloc and "BusTools_BC_MessageWrite" for additional information.

BusTools_BC_AperiodicRun supports a single instance of low or high priority messages list at a time. If this function is invoked to process a message list matching the priority of a currently transmitting list, it will return a status value of API_BC_APERIODIC_RUNNING. When an application supports multiple aperiodic messages of the same priority, use *WaitFlag* or BusTools_BC_AperiodicTest to ensure transmission of the current aperiodic message list is complete.

4.10 BusTools_BC_AperiodicTest

Description

BusTools_BC_AperiodicTest reports if the Bus Controller has started transmitting the last BC message in an aperiodic message queue, and if the BC is ready to accept another set of aperiodic messages.

When an aperiodic message queue is enabled, the Bus Controller inserts the queued messages into the running bus list when periodic transmission in the minor frame is inactive, on an individual message basis. This function polls the hardware register that indicates the status of execution of the aperiodic messages. It returns either API_BC_APERIODIC_RUNNING if the specified list is still running, or API_SUCCESS if the aperiodic processing is complete. An application invocation of BusTools_BC_AperiodicTest is useful when using a long list of low priority aperiodic messages that require several minor frames to complete transmission.

Prior to calling this function, you must initialize the channel using one of the BusTools/1553-API Initialization functions, initialize the Bus Controller using the BusTools_BC_Init function, define an aperiodic message queue via invocation of BusTools_BC_MessageWrite, and activate that list via BusTools_BC_AperiodicRun.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_AperiodicTest (cardnum, Hipriority);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
Hipriority	(BT_UINT) aperiodic message queue to query: 1 selects the High Priority message queue, 0 selects the Low Priority message queue.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_APERIODIC_RUNNING
API_BC_NOTINITED

4.11 BusTools_BC_AutoIncrMessageData

Description

BusTools_BC_AutoIncrMessageData creates an interrupt invoked thread that automatically increments a data word in a specified BC→RT receive message. Since this function creates a thread to increment the data word, upon completion of the respective data word increment sequence the application must also invoke this function to stop the respective thread. Note this function is only supported on host operating systems compatible with BusTools_RegisterFunction.

The application specifies the message number and data word (0 – 31) with respect to the specific BC message and data word to be modfied. The application also specifies an increment value, a start value, an increment rate and a maximum value. This function will set the specified data word to the start value and create a thread that increments the data word by the increment value, at a periodic occurrence repeated at *rate* times the message is transmitted. Setting the *rate* value to one causes the thread to increment the data word value for every message transmission. The specified data word will be incremented from the start value to the maximum value and then reset back to the start value.

Restrictions in the use of BusTools_BC_AutoIncrMessageData are described in the following paragraphs.

- Only a single data word in each BC message buffer can be autoincremented. If an application invokes BusTools_BC_AutoIncrMessageData to auto-increment more that one data word in a BC message buffer, this function will return the error code API_BC_AUTOINC_INUSE.
- This function can be used with software interrupts for frame rates slower than 10 milliseconds. For frame rates faster than 10 milliseconds, the hardware interrupt option must be used. The interrupt method is defined in the call to the respective BusTools/1553-API Initialization function.
- You can only use auto increment within the first 512 BC message buffers.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Controller using the BusTools_BC_Init function.

OS Support

Core API Function.

Syntax

```
wStatus = BusTools_BC_AutoIncrMessageData ( cardnum,   messno,  data_wrd,
          start,  incr,  rate,  max,  flag );

    wStatus          (BT_INT) status returned from this function.
```

cardnum	(BT_INT) channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messno	(BT_INT) the BC message number, valid range is 1 to 512.
data_wrd	(BT_INT) the data word to increment, valid range is 0 to 31.
start	(BT_U16BIT) data word starting value, valid range is 0 to 0xFFFF.
incr	(BT_U16BIT) data word increment value, valid range is 0 to 0xFFFF.
rate	(BT_INT) data word increment rate, valid range is 1 to 4294967295.
max	(BT_U16BIT) data word maximum value, valid range is 0 to 0xFFFF.
sflag	(BT_INT) auto-increment thread control flag, valid values are: 0 = terminate the thread, 1 = initiate the thread.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_FIFO_BAD
API_BUSTOOLS_FIFO_DUP
API_BUSTOOLS_NOTINITED
API_BUSTOOLS_NO_OBJECT
API_BUSTOOLS_TOO_MANY
API_BC_AUTOINC_INUSE
API_BC_NOTINITED
API_BC_NOTMESSAGE
API_MEM_ALLOC_ERR
API_REGISTERFUNCTION_OFF

4.12 BusTools_BC_Checksum1760

Description

BusTools_BC_Checksum1760 calculates a checksum according to the algorithm described in Appendix B, Section B.4.1.5.2.1 of the *Department of Defense Interface Standard for Aircraft/Store Electrical Interconnect Systems* MIL-STD-1760C Manual.

When each data word (including the checksum word) of a message is rotated right cyclically by a number of bits equal to the number of preceding data words in the message, and all the resultant rotated data words are summed using modulo 2 arithmetic to each bit (no carries), the sum shall be zero.

The application will pass a pointer to the respective API_BC_MBUF structure and a pointer to an unsigned short integer in which to store the calculated checksum. Prior to calling this function, you must fill in the data into the API_BC_MBUF structure. The function calculates the checksum and writes it into the last location in the message buffer.

Two examples show messages satisfying the checksum algorithm.

Example 1

Four Word Message:

1st Word 0000-0000-0000-0001 (0001 hex) data

2nd Word 1100-0000-0000-0000 (C000 hex) data

3rd Word 0000-1111-0000-0000 (0F00 hex) data

4th Word 0001-1110-0000-1011 (lE0B hex) checksum word

Example 2

Six Word Message:

lst Word 0001-0010-0011-0100 (1234 hex) data

2nd Word 0101-0110-0111-1000 (5678 hex) data

3rd Word 1001-1010-1011-1100 (9ABC hex) data

4th Word 1101-1110-1111-0000 (DEF0 hex) data

5th Word 0000-0000-0000-0000 (0000 hex) data

6th Word 1000-1111-0010-0000 (8F20 hex) checksum word

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_Checksum1760 (mbuf, cksum);

wStatus	(BT_INT) status returned from this function.
mbuf	(API_BC_MBUF *) pointer to BC message definition structure.
cksum	(BT_U16BIT *) reference to a 16-bit location to store the calculated checksum.

Return Value

API_SUCCESS

4.13 BusTools_BC_ControlWordRead

Description

BusTools_BC_ControlWordRead reads the control word for the specified Bus Controller message buffer. Other message content such as noop, stop, or conditional buffers, are not supported by the function.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions. Initialize the Bus Controller message buffers using BusTools_BC_Init, BusTools_BC_MessageAlloc, and BusTools_BC_MessageWrite.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_ControlWordRead (cardnum, messageid, api_control_word);

wStatus (BT_INT) status returned from this function.

cardnum (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

messageid (BT_UINT) BC message number (0-based).

api_control_word (BT_U16BIT *) reference to a 16-bit location to store the control word read.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_NOTMESSAGE

4.14 BusTools_BC_ControlWordUpdate

Description

BusTools_BC_ControlWordUpdate allows the application to modify specific parameters in the control word of a BC message without writing the entire BC message. Call this function to change the bus (A or B), data buffer (A or B), switch interrupts on or off, switch retries on or off, set interrupt-queue-only interrupts, and deactivate (NO-OP) or activate a message.

Prior to calling this function, the application must initialize the channel using one of the BusTools/1553-API Initialization functions, then initialize the Bus Controller using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_ControlWordUpdate (cardnum, messageid, controlWord, WaitFlag);

wStatus (BT_INT) status returned from this function.

cardnum (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

messageid (BT_UINT) BC message number (0-based).

controlWord (BT_U16BIT) New control word setting and ORed as follows:

BC_CONTROL_BUFFERA
BC_CONTROL_BUFFERB
BC_CONTROL_BUSA or
BC_CONTROL_CHANNELA
BC_CONTROL_BUSB or
BC_CONTROL_CHANNELB
BC_CONTROL_RETRY
BC_CONTROL_INTERRUPT
BC_CONTROL_INTQ_ONLY
BC_CONTROL_MESSAGE
BC_CONTROL_MSG_NOP
BC_CONTROL_NOP
(See page 350 for details regarding the control word definition.)

WaitFlag (BT_UINT) If non-zero, the function assures the Bus Controller isn't accessing the selected BC message before updating the control word. If zero, it updates the BC message immediately.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_UPDATEMESSTYPE
API_BC_NOTMESSAGE

4.15 BusTools_BC_DataBufferUpdate

Description

BusTools_BC_DataBufferUpdate is a method provided for use in updating the data portion of the specified BC data buffer. The data area to be updated is determined by the buffer address passed to this function.

While BusTools_BC_DataBufferUpdate can be invoked directly from an application, its specific purpose is to be used as a method to update a BC data buffer in response to a *BC Message Transaction* event when the BC is configured to use multiple BC data buffers. This application operational scenario is implemented by including an invocation of BusTools_BC_DataBufferUpdate within a user callback function created via invocation of BusTools_RegisterFunction. When the specific BC Message Transaction event occurs, the interrupt queue will contain 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). The callback function would use this value to designate the data buffer location to modify, specified in the parameter *buf_addr* in the invocation of BusTools_BC_DataBufferUpdate.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_BC_DataBufferUpdate (cardnum, buf_addr, dcount, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
buf_addr	(BT_UINT) data buffer address.
dcount	(BT_UINT) number of data word to write.
buffer	(BT_U16BIT *) array containing the application data to write.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_UPDATEMESSTYPE
API_BC_NOTMESSAGE

4.16 BusTools_BC_DataBufferWrite

Description

BusTools_BC_DataBufferWrite is a method provided for use in updating the data portion of the specified BC data buffer. The data area to be updated is determined by the message ID *mblock_id* and the Buffer ID *buffer_id* passed into this function.

BusTools_BC_DataBufferWrite is the preferred method for an application to update BC data buffer content when when the BC is configured to use multiple BC data buffers.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_BC_DataBufferWrite (cardnum, mblock_id, buffer_id, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mblock_id	(BT_UINT) BC message number (0-based).
buffer_id	(BT_UINT) BC message data buffer select, (0-based).
buffer	(BT_U16BIT *) array containing the application data to write.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_UPDATEMESSTYPE.

4.17 BusTools_BC_GetBufferCount

Description

BusTools_BC_GetBufferCount retrieves the number of data buffers that are allocated for a specific Bus Controller Message.

The application may invoke this function at any time after a channel is initialized using one of the BusTools/1553-API Initialization functions, the Bus Controller is initialized via BusTools_BC_Init, and BC Message Buffers are allocated via invocation of BusTools_BC_MessageAlloc.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_BC_GetBufferCount (cardnum, messno, count);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messno	(BT_UINT) BC message number (0-based).
count	(BT_U32BIT *) 32-bit location to write the data buffer count.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_UPDATEMESSTYPE

4.18 BusTools_BC_Init

Description

BusTools_BC_Init initializes Bus Controller operations on a channel. Invoke this function prior to calling any other functions accessing BC operations or structures. BusTools_BC_Init can be invoked multiple times to change the BC configuration, but the BC cannot be actively running. The application must terminate BC operations via invocation of BusTools_BC_StartStop prior to any attempt to alter the Bus Controller configuration with BusTools_BC_Init.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_Init (cardnum, bc_options, Enable, wRetry, wTimeout1, wTimeout2, frame, num_buffers);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
bc_options	(BT_UINT) Bus Controller options, (see the description for "**BC Options**" in the Notes below for details) REL_GAP - Relative gap, FIXED_GAP - Gap time from message start FRAME_START_TIMING - Frame start timing MSG_SCHD - Message scheduling FRAME_MESSAGING - Frame messaging MULTIPLE_BC_BUFFERS - Multiple BC data buffers (F/W Version 6.0 or greater). MFOVFL_INT_ENA - Enable the Minor frame overflow interrupts (F/W Version 6.0 or greater).
Enable	(BT_U32BIT) interrupt enable bits. For the interrupt enable bits definition, see the description for "**Interrupt Enable Bits**" in the Notes below for details, and Section 7.11, "Interrupt Enable / Message Status Bits (32 bit)".
wRetry	(BT_UINT) retry enable bits. For the retry enable bits definition, see the description for "**Retry Enable Bits**" in the Notes below for details, and Section 7.3, "BC Retry Parameters (BusTools_BC_Init argument)".

wTimeout1 (BT_UINT) timeout period for the "No Response" error -- must be between 4 and 31 μs; see the description for "**No Response Timeout**" in the Notes below for details.

wTimeout2 (BT_UINT) time-out period for the "Late Response" error -- must be between 4 and 31 μs; see the description for "**Late Response Timeout**" in the Notes below for details.

frame (BT_U32BIT) minor frame period in microseconds. See the description for "**Minor Frame Period**" in the Notes for details.

num_buffers (BT_UINT) number of data buffers associated with each BC message, either 1 or 2.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_RUNNING
API_BC_BADTIMEOUT1
API_BC_BADTIMEOUT2
API_BC_BADFREQUENCY

Notes

BC Options: The bc_options parameter sets the BC processing options. Currently, there are five Bus Controller options, Relative Gap timing (default), Fixed Gap timing, Frame Start timing, Message Scheduling, and Mulitple BC Buffers. Note only one timing option, Relative Gap, Fixed Gap or Frame Start timing, can be used on any channel. The following paragraphs provide a more detailed view of these options, which can be used individually or ORed together.

- **Relative Gap Timing** (REL_GAP) uses the gap-time to from the end of the message to determine when the next message transacts. That means the next message is always relative to the "end-of-message". If there is a no response or retries, the subsequent message is always the specified gap time from the end of message.
- **Fixed Gap Timing** (FIXED_GAP) sets the BC message gap timer to start from the beginning of the message transmission with a fixed timing gap from the previous message transaction. This option was added for API version 5.90 and greater using Firmware version 4.19 and greater. When the gap timer starts at the beginning of the message, the next message always transacts in a fixed time from the start of the previous message, regardless of any error condition. When the gap time starts from the end of the message (default), the next message transaction occurs after a time relative to the completion of the message. That can vary depending on the RT response. Fixed gap timing requires the application to consider the transaction time for a message. This includes the response time and any hardware retries. If the fixed gap time is not sufficient for

the message and any retries to complete, the next message is offset in time by the amount of transmission overlap. It is also possible to get a "No Inter-message Gap" error if the bus list violates the timing requirements.

- **Frame Start Timing** (FRAME_START_TIMING) sets the BC message gap timer to start from the beginning of the frame. This option was added starting with firmware version 5.00 and BusTools/1553-API version 6.42. The gap time applies to the current message. It determines when it starts in relation to the frame start. All messages have a precise start time within the frame. You can even delay the first message in the frame.
- **Frame Messaging** (FRAME_MESSAGING) allows the user to write 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.
- **Message Scheduling** (MSG_SCHD) allows you to program a start frame and repeat rate for each BC messages. It was added in BusTools/1553-API v6.20 running with Firmware version 4.40 or greater. With message scheduling, there is only a single instance of a message that is scheduled to transact in the specified frames. Previously, BC messages would need to be placed into the frames in which they were scheduled to run. This required allocating BC message buffers for each instance of the message. With message scheduling you use a single message with a message schedule. There are two changes to how you program a frame with message scheduling. Each BC message requires start frame and repeat rate parameters, and the end-of-frame marker (BC_CONTROL_MFRAME_END) is placed on a No-op message in the last message written in the frame. A begin-frame marker (BC_CONTROL_MFRAME_BEG) is still the first message in the frame. You can have multiple begin-frame messages if the first messages in the frame have different repeat rates and start frames, so long as there is only a single begin-frame marker for each frame iteration.
- **Multiple BC Buffers** (MULTIPLE_BC_BUFFERS) is available only with BusTools/1553-API version 8.00 running F/W version 6.0. When this option set, it allows an application to allocate any number of buffers for each message, limited only by memory available on the channel. Refer to the API function BusTools_BC_MessageBlockAlloc for details.
- **Minor frame overflow interrupts** (MFOVFL_INT_ENA) - Enable Minor Frame Overflow interrupts. The options available for Minor Frame Overflow interrupts include:
    - Minor frame overflow
    - BC_busy bit set on minor frame overflow
    - Low priority frame overflow
    - High priority frame overflow

Interrupt Enable Bits: The *Enable* parameter specifies the conditions causing a BC interrupt for the message to be recorded in the interrupt queue. When an enabled interrupt condition occurs, the interrupt message address is sent to the thread specified in BusTools_RegisterFunction.

Retry Enable Bits: The *wRetry* parameter specifies the conditions under which the Bus Controller retries a message. Starting with BusTools/1553-API version 6.20 this variable was changed from a BT_U16BIT (16-bits) to a BT_UINT (32-bits) to accommodate expanded retry conditions. See BusTools_BC_RetryInit for setting up multiple retries.

No Response Timeout: The *wTimeout* parameter specifies the "No Response" time-out period (in microseconds). If the RT in the BC command does not respond to the command within this time-out period, the BC logs a "No Response" error (and an interrupt is generated if that interrupt condition is enabled). This value ranges between 4 and 31 μs. The normal value for MIL-STD-1553B is 14μs.

Late Response Timeout: The *wTimeout2* parameter specifies the "Late Response" time-out period (in .5-μs increment). If the RT in the BC command responds to the command after this time-out period, but before the No Response time-out, the BC logs a "Late Response" error (and an interrupt is generated if that interrupt condition is enabled). You must specify a value between 4 and 31 μs, but less than the "No Response" time-out period. The normal value for MIL-STD-1553B is 12μs.

Minor Frame Period: The *frame* parameter specifies the minor frame period in μs. Starting with firmware version 5.00 and API version 6.42 this is a 32-bit value with a 1-μs resolution. The BC starts each minor frame in the BC message list according to this parameter. Use a value between 250 and 4294967295 in 1-μs steps. For F/W versions earlier than 5.0, use a value between 250 and 1638375 μs in steps of 25μs. If the time to complete any minor frame is larger than the time specified by the minor frame period, the global error "Minor Frame Overflow" occurs.

Number of BC Data Buffers: The *num_buffers* parameter specifies the number of BC data buffers to use. You can set *num_buffers* to 1 or 2. If set to 1, the API generates one Bus Controller Data buffer per message and both pointers in the BC message structure point to the same data buffer. Selecting one reduces the number of words required by each BC message by 33 words, enabling the definition and use of more BC messages.

4.19 BusTools_BC_IsRunning

Description

BusTools_BC_IsRunning returns a flag indicating the active state of the Bus Controller function on the specified channel. If the returned flag is non-zero, the BC is running. If the returned flag is zero, the BC is idle.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_IsRunning (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_UINT*) location to store the channel BC status: 0 = not running 1 = running

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED

4.20 BusTools_BC_IsRunning2

Description

The BusTools_BC_IsRunning2 function return value indicates the active state of the Bus Controller function on the specified channel, so applications can embed the function in another API function call. Successful execution values returned by BusTools_BC_IsRunning2 are API_BC_IS_RUNNING or API_BC_IS_STOPPED; otherwise, the returned value is an indication an error was encountered during execution.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_IsRunning2 (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_BC_IS_RUNNING
API_BC_IS_STOPPED
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.21 BusTools_BC_MessageAlloc

Description

BusTools_BC_MessageAlloc allocates onboard memory for the number of BC message buffers specified in the parameter *count*. It numbers the message buffers from 0 to *count*-1 and clears the allocated buffers. Create your bus list using BusTools_BC_MessageWrite to initialize and link these messages into a chain.

This function creates BC message buffers by allocating either 1 or 2 data buffers, depending on the value of *num_buffers* passed to BusTools_BC_Init. Specifying one buffer reduces the size of the BC messages, allowing you to create more messages.

Normally, this function is called only once. After this first call, if you need to allocate more message buffers you must re-initialize memory management by calling BusTools_BM_Init and BusTools_BC_Init, and then allocating additional BC message buffers. It is a good practice to allocate all the buffers you may need at the start of the application rather than adding more when needed.

If you are using aperiodic messages, make sure you allocate enough messages for both your periodic bus list and your aperiodic bus list since both reside within memory allocated for the respective channel. Create the bus lists using BusTools_BC_MessageWrite to initialize and link these messages into a chain. Leave the unused message buffers at the end of the list for the aperiodic messages. When you create the aperiodic messages, use the first of these extra messages for the first aperiodic message, point it to the next message, initialize that message, and continue until you complete all aperiodic messages. Ensure that the last aperiodic message has a next message pointer of 0xFFFF (see BusTools_BC_AperiodicRun).

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init. In addition, this function must be called prior to starting the BC with BusTools_BC_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageAlloc (cardnum, count);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
count	(BT_UINT) number of BC messages to be allocated.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_RUNNING
API_BC_MBUF_ALLOCD
API_BC_MEMORY_OFLOW

4.22 BusTools_BC_MessageBlockAlloc

Description

BusTools_BC_MessageBlockAlloc allocates onboard memory for a single block of BC message buffers plus the number of data buffers specified for the message. This function is available only with BusTools/1553-API version 8.00 and an Abaco Systems 1553 board programmed with F/W version 6.0. Call this function for each message block in the bus list. It references the BC message buffers starting from the value "0" and increments the message buffer count for each subsequent invocation.

On the initial invocation, BusTools_BC_MessageBlockAlloc creates a BC message control block and the number of data buffers passed in the count parameter. At least one data buffer must be allocated, but there is no upper limit other than what will fit in the available memory. The application must complete allocating all BC message blocks before creating the Bus Monitor buffers. You cannot allocate additional BC message control blocks after allocating the Bus Monitor buffer.

The bus list is created using BusTools_BC_MessageWrite to initialize and link these messages into a chain, and BusTools_BC_DataBufferWrite to fill in the data buffers.

If you are using aperiodic messages, assure enough messages are allocated for both the periodic bus list and the aperiodic bus list, as both reside in channel memory.

When utilizing aperiodic messages, reference the first message not previously allocated for periodic messages as the first aperiodic message, link it to the next message, initialize that message, and continue until you complete all aperiodic messages. Ensure that the last aperiodic message has a next message pointer of 0xFFFF (see BusTools_BC_AperiodicRun).

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init. This function must be called prior to starting the BC with BusTools_BC_StartStop.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_BC_MessageBlockAlloc (cardnum, bufID, count);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
bufID	(BT_UINT) identifies the buffer: BC_BLOCK_NEXT BC_BLOCK_LAST

count	(BT_UINT) the number of data buffers assigned to this message.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_RUNNING
API_HARDWARE_NOSUPPORT
API_BC_MULTI_BUFFER_ERR
API_BC_MEMORY_OFLOW
API_BAD_PARAM

4.23 BusTools_BC_MessageGetaddr

Description

BusTools_BC_MessageGetaddr provides the board-referenced address offset to the specified channel's BC message number. This address offset is a byte offset from the beginning of board memory and ranges from 0x20000 to the largest memory offset supported by the board. Pass this address to BusTools_MemoryRead2 or BusTools_MemoryWrite2 to perform direct access to the BC message structure.

Normally, applications don't require access to absolute memory addresses on Abaco Systems 1553 boards. Use this function for debugging or for implementing operations not provided by the high-level API functions. One example is changing a 1553 message command word without calling BusTools_BC_MessageWrite to re-write the entire message.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageGetaddr (cardnum, messageid, addr);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messageid	(BT_UINT) BC message number (0-based).
addr	(BT_U32BIT *) location to write the hardware address offset of the specified BC message.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK

4.24 BusTools_BC_MessageGetid

Description

BusTools_BC_MessageGetid converts an Abaco Systems 1553 board offset/address to a BC message number. The specified address is a byte offset from the start of board memory and ranges from 0x00000000 to 0x0003FFFF.

The typical use of this function is in support of BC message data buffer processing based on BC interrupt events. An application callback function will acquire the BC message address from the interrupt queue, then invoke this function to convert that address to the corresponding message number. With the message number the application can retrieve the message from the board via BusTools_BC_MessageRead.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageGetid (cardnum, addr, messageid);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
addr	(BT_U32BIT) board memory offset.
messageid	(BT_UINT*) pointer to returned BC message number.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_MBLOCK_NOMATCH

4.25 BusTools_BC_MessageNoop

Description

BusTools_BC_MessageNoop can toggle the specified BC message transaction between states of No Operation and Active Message. This is useful when the application is required to enable or disable a message in a running bus list for a given period. Prior to calling this function, you must set up the specified message as a normal 1553 message using BusTools_BC_MessageWrite.

This function can modify the active state of any normal message, including conditional and stop BC messages; however, this function cannot modify the definition of a message originally defined as a No-op message. There are two options when modifying the active state if a message, a standard no-op and a timed no-op. The standard no-op completely removes the message and message timing from the corresponding bus list, where the timed no-op removes the message from the bus list but preserves the message timing.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageNoop (cardnum, messageid, NoopFlag, WaitFlag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messageid	(BT_UINT) BC message number (0-based).
NoopFlag	(BT_UINT) Set message options: NOOP – Omits the message transaction and any respective message timing during execution of the bus list. TIMED_NOOP – Omits the message transaction during execution of the bus list but preserves the message timing. MSG_OP – Includes the message transaction and message timing during execution of the bus list.
WaitFlag	(BT_UINT) Unused legacy parameter.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_NOTMESSAGE
API_BC_NOTNOOP
API_BC_CANT_NOOP

4.26 BusTools_BC_MessageRead

Description

BusTools_BC_MessageRead reads the specified BC message from the channel.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageRead (cardnum, messageid, api_message);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messageid	(BT_UINT) BC message number (0-based).
api_message	(API_BC_MBUF *) location to store the BC message content.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_ILLEGALMESSAGE
API_BC_MESS1_COND
API_BC_BAD_COND_ADDR

Notes

In addition to BC→RT and RT→BC messages, the BC monitors RT→RT message data and records it in the BC message data buffer.

For F/W version 4.99 and earlier, message gap time is saved in the gap_time structure member; otherwise, the message gap time data is stored in the long_gap structure member.

4.27 BusTools_BC_MessageBufferRead

Description

BusTools_BC_MessageBufferRead is a method provided for use in reading a specific BC Message Buffer from a channel based on its location in channel memory. The data contained therein is returned in the caller supplied structure.

While BusTools_BC_MessageBufferRead can be invoked directly from an application, its specific purpose is to be used as a method to read a BC data buffer in response to a *BC Message Transaction* event when the BC is configured to use multiple BC data buffers. This application operational scenario is implemented by including an invocation of BusTools_BC_MessageBufferRead within a user callback function created via invocation of BusTools_RegisterFunction. When the specific BC Message Transaction event occurs, the interrupt queue will contain 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). The callback function would use this value to designate the data buffer location to read, specified in the parameter *addr* in the invocation of BusTools_BC_MessageBufferRead.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_BC_MessageBufferRead (cardnum, addr, api_message);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
addr	(BT_U32BIT) address of the data buffer generating an interrupt.
api_message	(API_BC_MBUF *) location to store the BC message content.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_MESS1_COND
API_BC_BAD_COND_ADDR

Notes

In addition to BC→RT and RT→BC messages, the BC monitors RT→RT message data and records it in the BC message data buffer.

4.28 BusTools_BC_MessageReadData

Description

BusTools_BC_MessageReadData reads the data buffer of the specified BC message from the Abaco Systems 1553 board. If BusTools_BC_Init was invoked for the respective channel with a *num_buffers* parameter value of 2, this function reads the data from the active buffer designated by the value of the A/B buffer bit in the BC Message Buffer Control Word; otherwise it reads from the single active buffer.

If the BC Function on the respective channel is configured to use Multiple Data Buffers, this function will return an error; instead, the application must use the function BusTools_BC_ReadDataBuffer to read data from a BC message data buffer.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageReadData (cardnum, messageid, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messageid	(BT_UINT) BC message number (0-based).
buffer	(BT_U16BIT *) location to store the BC message data buffer content.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_MESS1_COND
API_BC_BAD_COND_ADDR

Notes

In addition to BC→RT and RT→BC messages, the BC monitors RT→RT message data and records it in the BC message data buffer.

4.29 BusTools_BC_MessageReadDataBuffer

Description

This function reads the specified BC message data buffer from the specified BC message on an Abaco Systems 1553 board. If BusTools_BC_Init was invoked for the respective channel with a *num_buffers* parameter value of 2, this function reads the data from the specified A/B buffer; however, if the channel is configured to use a single channel, this function will read from the respective single BC data buffer.

If the BC Function on the respective channel is configured to use Multiple Data Buffers, this function will return an error; instead, the application must use the function BusTools_BC_ReadDataBuffer to read data from a BC message data buffer.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageReadDataBuffer (cardnum, mblock_id, buffer_id, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mblock_id	(BT_UINT) BC message number (0-based).
buffer_id	(BT_UINT) BC message data buffer index; where 0 = Buffer A 1 = Buffer B
buffer	(BT_U16BIT *) location to store the BC message data buffer content.

Return Value

API_SUCCESS
API_BAD_PARAM
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_NOTINITED
API_BC_UPDATEMESSTYPE
API_BC_MBLOCK_NOMATCH

4.30 BusTools_BC_MessageUpdate

Description

BusTools_BC_MessageUpdate is a method provided for use in updating the specified BC data buffer. The message data buffer updated for the respective channel is based on the buffer configuration defined in the invocation of BusTools_BC_Init and indicated via the value of the A/B buffer control bit in the BC Message Buffer Control Word. If the BC is configured for two buffers, this function writes the new data to the inactive buffer and changes the active buffer. If the BC is configured for a single buffer, this function updates the single BC data buffer. The word count specified by the first 1553-command word in the message indexed by *mblock_id* determines the number of words written to the BC message buffer.

If the BC Function on the respective channel is configured to use Multiple Data Buffers, this function will return an error; instead, the application must use the function BusTools_BC_DataBufferUpdate to update a BC message data buffer. Any attempt to update the content of a NO-OP message will result in the error code API_BC_UPDATEMESSTYPE.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageUpdate (cardnum, mblock_id, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mblock_id	(BT_UINT) BC message number (0-based).
buffer	(BT_U16BIT *) location of the data buffer content to use.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_MULTI_BUFFER_ERR
API_BC_NOTINITED
API_BC_UPDATEMESSTYPE

4.31 BusTools_BC_MessageUpdateBuffer

Description

BusTools_BC_MessageUpdateBuffer updates the specified BC message data buffer for the specific BC message. The message data buffer updated for the respective channel is based on the buffer configuration defined in the invocation of BusTools_BC_Init and indicated via the value of the A/B buffer control bit in the BC Message Buffer Control Word. If the BC is configured for two buffers, this function writes the new data to the buffer referenced via *buffer_id*. the BC is configured for a single buffer, this function updates that BC data buffer regardless of the value of *buffer_id*. The word count specified by the first 1553-command word in the message indexed by *messageid* determines the number of words written to the BC message buffer.

If the BC Function on the respective channel is configured to use Multiple Data Buffers, this function will return an error; instead, the application must use the function BusTools_BC_DataBufferUpdate to update a BC message data buffer. Any attempt to update the content of a NO-OP message will result in the error code API_BC_UPDATEMESSTYPE.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageUpdateBuffer (cardnum, messageid, buffer_id, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messageid	(BT_UINT) BC message data buffer index; where 0 = Buffer A 1 = Buffer B
buffer_id	(BT_UINT) message data buffer to update 0=A, 1=B.
buffer	(BT_U16BIT *) location of the data buffer content to use.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM

API_BUSTOOLS_NOTINITED
API_BC_ILLEGAL_MBLOCK
API_BC_MULTI_BUFFER_ERR
API_BC_UPDATEMESSTYPE

4.32 BusTools_BC_MessageWrite

Description

BusTools_BC_MessageWrite writes a Bus Controller message or control structure to a specified Bus Controller message buffer on the Abaco Systems 1553 board. After allocating message buffers via invocation of BusTools_BC_MessageAlloc or BusTools_MessageBlockAlloc, invoke this function to copy the message or control information to the respective buffers on the board If the BC Function on the respective channel is configured to use two or more data buffers, this function can only be used to define the content in the first data buffer. Invoke the function BusTools_BC_DataBufferWrite to define message data content in the remaining data buffers.

There are two types of BC structures supported by the BC Message Buffer structure (API_BC_MBUF), a BC Control Buffer and BC Message Buffer. A structure defined by the application as a BC Control Buffer stores the data for No-ops, Timed No-ops, Stop messages, Conditional branches, Minor Frame start/end, and the logical end of a BC message list. A structure defined by the application as a BC Message Buffer stores the parameters associated with a specific 1553 message. Content within a BC Message Buffer includes RT address, subaddress, word count and data for Mode Codes, and message data for RT→BC, BC→RT, RT→RT, and Broadcast messages.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_MessageWrite (cardnum, messno, api_message);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messno	(BT_UINT) BC message number (0-based).
api_message	(API_BC_MBUF *) reference to the BC message definition structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_ILLEGAL_MBLOCK

API_BC_ILLEGAL_NEXT
API_BC_ILLEGAL_PREV
API_BC_ILLEGAL_BRANCH
API_BC_MESS1_COND
API_BC_BAD_COND_ADDR
API_BC_ILLEGAL_TARGET

Notes

This function is used to create all Bus Controller 1553 data and control messages. The required parameters for each message type are defined as follows:

4.32.1 BC 1553 Data Message

BC 1553 Data Message types support normal RT→BC, BC→RT, RT→RT messages, broadcast BC→RT and RT→BC messages, and mode codes. The application is required to define the following structure members in the API_BC_MBUF structure before invoking BusTools_BC_MessageWrite for a BC 1553 Data Message:

control	Should be set to BC_CONTROL_MESSAGE for a normal 1553 message, or BC_CONTROL_MSG_NOP to define the message in the NO-OP state. Add any of the following attributes to this initial control value to further define this message: BC_CONTROL_MFRAME_BEG: First minor frame message. BC_CONTROL_MFRAME_END: Last minor frame message. BC_CONTROL_INTERRUPT: Generate an interrupt. BC_CONTROL_RETRY: Retries are required for this message. BC_CONTROL_BUSA: Transmit message on the primary bus. BC_CONTROL_BUSB: Transmit message on the secondary bus.
messno	Message Number.
messno_next	Set to the index of the next BC message. If you are creating an aperiodic message list, and this is the last message in the list, set this parameter to 0xFFFF.
messno_prev	Not used.
errorid	Set to the index of a valid error injection buffer, or to zero for no error injection.
gap_time	Intended for use with F/W V4.66 or earlier only. A 16-bit gap time, specifying the delay applied after this message transaction completes before the next transaction is processed. Valid range is 4µs to 65535µs. This gap time is not applied if the message is the last message in a minor frame or aperiodic list.
long_gap	Intended for use with F/W V6.x or later only. A 24-bit gap time, specifying the delay applied after this message transaction completes before the next transaction is processed. Valid range

is 4μs to 16,777,215μs. This gap time is not applied if the message is the last message in a minor frame or aperiodic list.

data[0][0-31] Initial values for the first data buffer if this is an RT receive message.

data[1][0-31] Initial values for the second data buffer if this is an RT receive message and the BC is using double buffering.

mess_command1 Defined as follows:

mess_command1.rtaddr Set to the RT number (0 TO 30) or 31 for a broadcast message (if enabled).

mess_command1.tran_rec Set to 1 for transmit, 0 for receive.

mess_command1.subaddr Set to the RT subaddress, or 0 (or 31 if enabled) to define a mode code.

mess_command1.wcount Set to the number of data words to transfer (0 for 32 words), or the mode code number if this is a mode code.

mess_command2 Not used.

start_frame When Message Scheduling is active, a start frame must be defined for each message specifying the frame number in which the message is first processed. A start frame of 1 is a request for the message to be processed in the first frame. A start frame of 0 disables the message.

rep_rate When Message Scheduling is active, the repeat rate defines how often a message is processed. A repeat rate of 1 is a request for the message to be processed in every frame starting with the *start_frame*. A repeat rate of *n* is a request for the message to be processed every *n*th frame beginning with the *start_frame*. If the repeat rate is 0 and the start frame is greater than 0, then the message is processed only once in the *start_frame*.

When the control structure member is defined as a BC_CONTROL_MESSAGE, the API ignores the remaining items in the API_BC_MBUF structure.

Example 1: First message in a minor frame

```
API_BC_MBUF bcmessage;          //Create a BC Message Structure
bcmessage.messno = messno;
bcmessage.messno_next = (BT_U16BIT)(messno + 1);
bcmessage.control = BC_CONTROL_MESSAGE;     //This is a Control msg
bcmessage.control |= BC_CONTROL_INTERRUPT;  //Enable Interrupt
bcmessage.control |= BC_CONTROL_BUFFERA;    //Use Buffer A
bcmessage.control |= BC_CONTROL_MFRAME_BEG; //Begin Minor Frame
bcmessage.mess_command1.rtaddr   = RT_ADDR;
bcmessage.mess_command1.subaddr  = SUB_ADDR;
```

```
bcmessage.mess_command1.wcount   = MSG_WORD_COUNT;
bcmessage.mess_command1.tran_rec = RECEIVE;
bcmessage.errorid = 0;  // Default error injection buffer, no errors
bcmessage.gap_time = 8; // 8-µs inter-message gap.
for ( j = 0; j < MSG_WORD_COUNT; j++ )
{
   bcmessage.data[0][j] = messageData[j]; //Fill in the data
}
status = BusTools_BC_MessageWrite(cardnum, messno, &bcmessage);
return status;
```

Example 2: Message in a minor frame

```
API_BC_MBUF bcmessage;   //Define BC Message Structure

bcmessage.messno = messno;
bcmessage.messno_next = (BT_U16BIT)(messno + 1);

bcmessage.control = BC_CONTROL_MESSAGE;      //Control msg
bcmessage.control |= BC_CONTROL_INTERRUPT;   //Enable Int
bcmessage.control |= BC_CONTROL_BUFFERA;     //Use Buffer A

bcmessage.mess_command1.rtaddr   = RT_ADDR;
bcmessage.mess_command1.subaddr  = SUB_ADDR;
bcmessage.mess_command1.wcount   = MSG_WORD_COUNT;
bcmessage.mess_command1.tran_rec = TRANSMIT;

bcmessage.errorid = 0; // Default error injection buffer, no errors
bcmessage.gap_time = 8;    // 8-µs inter-message gap.

status = BusTools_BC_MessageWrite(cardnum, messno, &bcmessage);
return status;
```

Example 3: Mode Code

```
API_BC_MBUF bcmessage;   //Define BC Message Structure

bcmessage.messno = messno;
bcmessage.messno_next = (BT_U16BIT)(messno + 1);

bcmessage.control = BC_CONTROL_MESSAGE;     //Control msg
bcmessage.control |= BC_CONTROL_INTERRUPT;  //Enable Int
bcmessage.control |= BC_CONTROL_BUFFERA;    //Use Buffer A

bcmessage.mess_command1.rtaddr  = RT_ADDR;
bcmessage.mess_command1.subaddr = 0; //Also 31 if programmed
bcmessage.mess_command1.wcount  = MODE_CODE;
bcmessage.mess_command1.tran_rec = T/R; //Defined by Mode code

bcmessage.errorid = 0; // Default Error Inj. Buffer, no errors
bcmessage.gap_time = 8; // 8-µs inter-message gap.

status = BusTools_BC_MessageWrite(cardnum, messno, &bcmessage);
return status;
```

Example 4: Message Scheduling

```
API_BC_MBUF bcmessage;   //Define BC Message Structure

bcmessage.messno = messno;
bcmessage.messno_next = (BT_U16BIT)(messno + 1);
```

```
bcmessage.control = BC_CONTROL_MESSAGE;      //Control msg
bcmessage.control |= BC_CONTROL_INTERRUPT;   //Enable Int
bcmessage.control |= BC_CONTROL_BUFFERA;     //Use Buffer A

bcmessage.mess_command1.rtaddr   = RT_ADDR;
bcmessage.mess_command1.subaddr  = SUB_ADDR;
bcmessage.mess_command1.wcount   = MSG_WORD_COUNT;
bcmessage.mess_command1.tran_rec = TRANSMIT;

bcmessage.errorid = 0; // Default error injection buffer, no errors
bcmessage.gap_time = 8;    // 8-µs inter-message gap.
bcmessage.start_frame = 1; // Start message in the first frame
bcmessage.rep_rate = 3;    // Repeat the message every third frame.

status = BusTools_BC_MessageWrite(cardnum, messno, &bcmessage);
return status;
```

Example 1 shows how to set up the first message in a minor frame. Notice the example ORs in the BC control bits including BC_CONTROL_MFRAME_BEG bit. You can use the same code for setting any 1553 message in the minor frame, omitting the begin-minor-frame bit. For the last message in the minor frame set the BC_CONTROL_MFRAME_END bit. In addition, this is a receive command that sends data from the Bus Controller to the Remote Terminal.

Example 2 shows how to set up a transmit command that requests data from the remote terminal and no data is filled into the data array.

Example 3 shows the mode code format. You can use either subaddress 0 or 31 depending on the board setup (see BusTools_SetSa31). The word count field specifies the mode code, (see the mode code definitions in the MIL-STD-1553 Tutorial). The Mode Code defines the Transmit/Receive bit setting, data words and availability for broadcast.

Example 4 is the same as example 2; however, it uses the start_frame and rep_rate parameters to schedule the message to start in the first frame and run every third frame. These two parameters are only valid when the MSG_SCHD option is set in the call to BusTools_BC_Init. Message scheduling parameters apply to all Bus Controller messages, but not to control messages such as noop, timed noop, stop, or conditional branches.

For broadcast commands, set the RT value to 31. Broadcast commands are receive-only commands. The Remote Terminals do not return status on broadcast commands. You can also send a broadcast mode code by using the mode code format described in example 3 and addressing RT 31.

4.32.2 1553 RT Messages (RT→RT, RT→RT Broadcast)

RT→RT and RT→RT Broadcast messages require the following parameters in the API_BC_MBUF structure be defined before invoking BusTools_BC_MessageWrite:

control — Defined the same as the BC 1553 Data Message control structure member.

messno — Message Number.

messno_next — Set to the index of the next BC message. If you are creating an aperiodic message list, and this is the last message in the list, set this parameter to 0xFFFF.

messno_prev — Not used.

errorid — Set to the index of a valid error injection buffer, or to zero for no error injection.

gap_time — Defined the same as the BC 1553 Data Message gap_time structure member.

long_gap — Defined the same as the BC 1553 Data Message long_gap structure member.

data[0][0-31] — does not need initialization.

data[1][0-31] — does not need initialization.

mess_command1 — 1553 RT Rx Command Word, should be defined as follows:

- mess_command1.rtaddr — set to the RT number (0-30) that RECEIVES the data, or 31 for a broadcast RT→RT message.
- mess_command1.tran_rec — must be set to 0.
- mess_command1.subaddr — set to the receiving RT subaddress.
- mess_command1.wcount — set to the number of data words to transfer.

mess_command2 — 1553 RT Tx Command Word, should be defined as follows:

- mess_command2.rtaddr — set to the RT number (0-30) that TRANSMITS the data. May *not* be the same RT number specified in mess_command1.
- mess_command2.tran_rec — must be set to 1.
- mess_command2.subaddr — set to the transmitting RT subaddress.
- mess_command2.wcount — set to the number of data words to transfer.

bcmessage.start_frame — Start message in the first frame

bcmessage.rep_rate — Repeat the message every third frame.

The API ignores the remaining elements in the API_BC_MBUF structure.

Example 5: RT→RT message

```
API_BC_MBUF bcmessage;   //Define BC Message Structure

messno++;
bcmessage.messno = messno;
bcmessage.messno_next = (BT_U16BIT)(messno + 1);

bcmessage.control = BC_CONTROL_MESSAGE;      //Control msg
bcmessage.control |= BC_CONTROL_INTERRUPT;   //Enable Int
bcmessage.control |= BC_CONTROL_BUFFERA;     //Use Buffer A
bcmessage.control |= BC_CONTROL_RTRTFORMAT   //RT→RT

bcmessage.mess_command1.rtaddr   = RT_ADDR_1;
bcmessage.mess_command1.subaddr  = SUB_ADDR_1;
bcmessage.mess_command1.wcount   = MSG_WORD_COUNT;
bcmessage.mess_command1.tran_rec = RECEIVE;

bcmessage.mess_command2.rtaddr   = RT_ADDR_2;
bcmessage.mess_command2.subaddr  = SUB_ADDR_2;
bcmessage.mess_command2.wcount   = MSG_WORD_COUNT;
bcmessage.mess_command2.tran_rec = TRANSMIT;

bcmessage.errorid = 0;     // Default error injection buffer (no
errors)
bcmessage.gap_time = 8;    // 8-µs inter-message gap.
bcmessage.start_frame = 1; // Start message in the first frame
bcmessage.rep_rate=3;      // Repeat the message every third frame.

status = BusTools_BC_MessageWrite(cardnum, messno, &bcmessage);
return status;
```

The above example shows a RT→RT message that commands one RT to receive data and another RT to transmit data. The code ORs in the RT→RT format bit in the BC control word. Message command 1 contains the receive RT data and message command 2 contain the transmit RT data.

You program a broadcast RT→RT message by replacing RT_ADDR_1 in the above example with the broadcast RT address, 31. This designates a single RT to transmit data while the other RTs receive this data.

4.32.3 Conditional Message

Conditional messages allow the application to program conditional execution paths within a bus list. Conditional "messages" do not generate any 1553 bus traffic, nor do they inject inter-message gaps or generate interrupts. Conditional messages provide a branch point in the bus list where you can conditionally execute certain 1553 messages based on the value of a data word. That data word can either be a location in onboard RAM or part of a previous message transaction. The application is required to define the following structure members in the API_BC_MBUF structure before invoking BusTools_BC_MessageWrite for a Conditional Message.

control	Should be set to one of the following Branch options: BC_CONTROL_BRANCH branch immediate to a message. BC_CONTROL_CONDITION conditional branch on the previous message.

	BC_CONTROL_CONDITION2 conditional branch on a specific address. BC_CONTROL_CONDITION3 conditional branch on a specific message and data word value. Optionally BC_CONTROL_INTERRUPT can be added to the branch if interrupt generation is desired.
data_value	Set to the data value the compare matches.
data_mask	Enable bits to be tested by setting the data mask bits to 1.
messno_next	Set to the message index of the BC message to process if the condition tested is false (not equal).
messno_branch	Set to the message index of a valid BC message that will be executed next if the condition is true (equal).
cond_count_val	Number of times condition is true before processing the branch. A value of 0 results in a branch on every matching condition, 1 is every other match. This value is loaded into the counter whenever the branch occurs. Maximum value is 65,536.
cond_counter	Initial counter setting. Usually set the same as cond_count_val.

For Conditional Branch and Conditional Branch 3

messno_compare	Set the index of the message containing the data word specified in “address”.
address	Set to the word number of the previous message or specified message to compare for BC_CONTROL_CONDITION or BC_CONTROL_CONDITION3 conditional branch messages. Where: 0 = Command Word 1 1 = Command Word 2 (RT to RT only) 2 = Status Word 1 3 = Status Word 2 (RT to RT only) 4-35 = Data Words

For Conditional Branch 2

test_address	The byte address of the memory word tested by BC_CONTROL_CONDITIONAL2 conditional branch message; otherwise, this structure member is ignored by the API.
spare	An unused parameter set to zero for compatibility with future enhancements.

The API ignores the remaining elements in the API_BC_MBUF structure.

Example 6: Conditional Branch 2

```
API_BC_MBUF bcmessage;   //Define BC Message Structure
messno++;

bcmessage.control = BC_CONTROL_CONDITION2;
bcmessage.messno  = messno;
bcmessage.messno_next   = (BT_U16BIT)(messno + 1);
bcmessage.messno_branch = (BT_U16BIT)(messno + 3);

bcmessage.test_address = TEST_ADDRESS;
bcmessage.data_value   = DATA_VALUE;
bcmessage.data_mask    = MASK;

status = BusTools_BC_MessageWrite (cardnum, messno, &bcmessage);
return status;
```

The above example shows a conditional branch using a memory location, TEST_ADDRESS. During executions, the BC compares the memory location TEST_ADDRESS with the data, DATA_VALUE using the mask. If there is a match, the BC executes the branch message; otherwise, the BC executes the next message.

Example 7: Conditional Branch or Conditional Branch 3

```
API_BC_MBUF bcmessage;   //Define BC Message Structure
messno++;

bcmessage.control = BC_CONTROL_CONDITION3;
bcmessage.messno  = messno;
bcmessage.messno_next   = (BT_U16BIT)(messno + 1);
bcmessage.messno_branch = (BT_U16BIT)(messno + 3);

bcmessage.address = MSG_ADDR; // value between 0 and 35
bcmessage.messno_compare = PREV_MESS // Message number for a
                           // previously transacted message.
                           // Condition Branch 3 only

bcmessage.data_value   = DATA_VALUE;
bcmessage.data_mask    = MASK;

status = BusTools_BC_MessageWrite (cardnum, messno, &bcmessage);
return status;
```

The above example shows a conditional branch using data from a previous message transaction. PREV_MESSAGE refers to the message using the assigned message number (messno) set in a call to BusTools_BC_MessageWrite. MSG_ADDRESS points to the word within the message. As mentioned previously, this is a 0-based number with 0 pointing to the command word and 4 pointing to the first data word. During execution, the BC compares the message word pointed to by PREV_MESSAGE and MSG_ADDRESS with the data, DATA_VALUE using the mask. If there is a match, the BC executes the branch message; otherwise, the BC executes the next message.

For BC_CONTROL_CONDITION, the API defaults to the previous message and ignores messno_compare.

4.32.4 Stop BC

The last logical message executed in the BC message list is a stop BC or a BC_CONTROL_BRANCH specifying the beginning message in the list as the messno_next. The Stop BC message doesn't cause an interrupt or create a gap time. When executed, the Stop BC message turns off the BC RUN bit in the control and status register. Use the following parameters to setup the Stop BC message:

control	Set to BC_CONTROL_LAST. Optionally BC_CONTROL_INTERRUPT can be added to the branch.
messno_next	Should be set to the index of a valid BC message that executes if the BC RUN bit is set without re-initializing the BC.

The API ignores the remaining elements in the API_BC_MBUF structure.

Example 8: Last BC Message

```
API_BC_MBUF bcmessage;   //Define BC Message Structure
messno++;

bcmessage.control     = BC_CONTROL_LAST;
bcmessage.messno_next = NEXT_MESSAGE;

status = BusTools_BC_MessageWrite (cardnum, messno, &bcmessage);
return status;
```

4.32.5 No-op Message

The NO-OP message is a placeholder; its only function is to transfer processing to the next message. It does not generate any 1553 traffic, cause an interrupt, or inject a gap time. The following parameters define a NO-OP message:

control	BC_CONTROL_NOP.
messno_next	Set to the index of a valid BC message to process next.

The API ignores the remaining elements in the API_BC_MBUF structure.

Example 9: No-op Message

```
API_BC_MBUF bcmessage;   //Define BC Message Structure
messno++;
bcmessage.control = BC_CONTROL_NOP;
bcmessage.messno_next   = NEXT_MESSAGE;

status = BusTools_BC_MessageWrite (cardnum, messno, &bcmessage);
return status;
```

4.32.6 Timed No-op Message

The Timed NO-OP message is a placeholder like the NO-OP. The difference between Timed NO-OP and NO-OP is that the gap delay from the previous message

and gap delay programmed for the Timed NO-OP are included in the execution of the bus list. The Timed NO-OP can be utilized to extend the gap time between messages, as well as generate an interrupt. The following parameters define a Timed NO-OP message:

control	BC_CONTROL_TIMED_NOP. Optionally BC_CONTROL_INTERRUPT can be added.
messno_next	Set to the index of a valid BC message to run next.
gap_time	Intended for use with F/W V4.66 or earlier only. A 16-bit gap time, specifying the delay applied after this message transaction completes before the next transaction is processed. Valid range is 4µs to 65535µs. This gap time is not applied if the message is the last message in a minor frame or aperiodic list.
long_gap	Intended for use with F/W V6.x or later only. A 24-bit gap time, specifying the delay applied after this message transaction completes before the next transaction is processed. Valid range is 4µs to 16,777,215µs. This gap time is not applied if the message is the last message in a minor frame or aperiodic list.

Example 10: Timed No-op Message

```
API_BC_MBUF bcmessage;   //Define BC Message Structure
messno++;
bcmessage.control = BC_CONTROL_TIMED_NOP;
bcmessage.messno_next    = NEXT_MESSAGE;
bcmessage.gap_time = 100;     // 100-µs inter-message gap.

status = BusTools_BC_MessageWrite (cardnum, messno, &bcmessage);
return status;
```

4.32.7 Mode Codes and Dynamic Bus Control

The 1553 Message format described above supports Mode Codes and Mode Code 0 (Dynamic Bus Control). When using Mode Code 0, the RT returns the dynamic bus control acceptance (DBCA) bit in its status word and if properly configured changes to the Bus Controller. To transfer the BC function to an external RT, program the Dynamic Bus Control mode code as the last command in a frame and follow it with a Stop BC command. It is a good idea to precede the Stop BC command with a conditional command, which tests for the DBCA status word bit before executing the Stop BC command. See the "Remote Terminals" section in the *BusTools/1553-API Software User's Manual* for more details on Dynamic Bus Control.

The Stop BC command should point to the BC message that you want to execute when this unit converts back to the Bus Controller. This first message might begin with a null minor frame to delay putting out the first command, as the two units

aren't synchronized. Otherwise, the new BC puts out its first command around 10-20 μs after the mode code status word transmits.

4.33 BusTools_BC_ReadDataBuffer

Description

BusTools_BC_ReadDataBuffer is a method provided for use in reading a specific BC Message Data Buffer from a channel based on its location in channel memory.

While BusTools_BC_ReadDataBuffer can be invoked directly from an application, its specific purpose is to be used as a method to read the data portion of a BC data buffer in response to a *BC Message Transaction* event when the BC is configured to use multiple BC data buffers. This application operational scenario is implemented by including an invocation of BusTools_BC_ReadDataBuffer within a user callback function created via invocation of BusTools_RegisterFunction. When the specific BC Message Transaction event occurs, the interrupt queue will contain the address of the data buffer that generated the interrupt, (the address of that buffer is stored in the API_INT_FIFO entry fifo[tail].buff_off). The callback function would use this value to designate the data buffer location to read, specified in the parameter *bufaddr* in the invocation of BusTools_BC_ReadDataBuffer.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_BC_ReadDataBuffer (cardnum, bufaddr, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
bufaddr	(BT_UINT) address offset of the data buffer to read.
buffer	(BT_U16BIT *) location to store the data buffer contents.

Return Value

API_SUCCESS
API_BC_NOTINITED
API_BUSTOOLS_NOTINITED
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT

4.34 BusTools_BC_ReadLastMessage

Description

BusTools_BC_ReadLastMessage returns the last Bus Controller message recorded in the interrupt queue matching the criteria in the argument list. The function parameter list allows the application to filter the last BC message search using the message RT address, subaddress, and transmit or receive setting. The *rt_addr* and *subaddress* input arguments are bit-encoded values. For example, to select RT0 or subaddress 0 use the LSB (0x0001). The function then searches in reverse order through the interrupt queue for a message matching the settings. It returns API_BC_READ_NODATA if no matching message was found.

On the initial invocation of this function, it searches the entire contents of the interrupt queue for a match. Subsequent invocations result in a search of the interrupt queue for a match between the current queue pointer and the last queue pointer referenced in the previous invocation. This function must be invoked at a periodic frequency such that the interrupt queue does not wrap, and queue entries are lost. There are 296 interrupt queue entries for F/W V4/5, and 512 queue entries for F/W V6 and later.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc, and start the Bus Controller using BusTools_BC_Start or BusTools_BC_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_ReadLastMessage (cardnum, rt_addr_mask, subaddress, tr, pBC_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rt_addr	(BT_INT) selects the RT address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
subaddress	(BT_INT) selects the subaddress (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
tr	(BT_INT) select the transaction type as transmit or receive. 0 = Receive; 1 = Transmit; -1 = don’t care.

pBC_mbuf (API_BC_MBUF *) location where the message buffer contents will be written if a matching message is detected.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_READ_NODATA
API_BC_MBLOCK_NOMATCH

Notes

This function Bus Controller messages in the interrupt queue, where only those BC messages configured to generate interrupts will reside.

To generate interrupts on BC messages, perform following BC initialization:

1. Enable BC Interrupts by setting *Enable* in the call to BusTools_BC_Init. Set at least BT1553_INT_END_OF_MESS in this word to ensure an interrupt on every BC message.
2. Set the BC_CONTROL_INTERRUPT bit in the bus message's control word when the bus message is written to the BC memory allocated on this channel (see BusTools_BC_MessageWrite).

Example

The following code demonstrates how to use this function.

```
API_BC_MBUF bcmessage;

BT_U32BIT BIT = 1;
BT_UINT RT = BIT << 4;  // Select RT Address 4
BT_UNIT SA = -1;        // Select any subaddress
BT_UINT TX_RX = 1;      // Select a transmit transaction type

status = BusTools_BC_ReadLastMessage(cardnum, RT, SA, TX_RX,
                                     &bcmessage);

if (status == 0)
{
   // bcmessage will contain the new data
```

The above code returns the last transmit message to RT address 4.

4.35 BusTools_BC_ReadLastMessageBlock

Description

This function returns message buffer contents of all Bus Controller messages in the interrupt queue that fit the criteria provided in the application-supplied argument list. The function parameter list allows the application to filter the BC message search using the message RT address, subaddress, and transmit or receive setting. The *rt_addr_mask* and *subaddr_mask* input arguments are bit-encoded values. For example, to select RT0 or subaddress 0 use the LSB (0x0001). The function then searches in reverse order through the interrupt queue for all messages matching the settings. The function returns API_BC_READ_NODATA if a matching message is not found; otherwise, the function fills an array of BC Message Buffer (API_BC_MBUF) structures with the messages found. The function also returns a count of the messages found.

On the initial invocation of this function, it searches the entire contents of the interrupt queue for matching entries. Subsequent invocations result in a search of the interrupt queue for matching entries between the current queue pointer and the last queue pointer referenced in the previous invocation. This function must be invoked at a periodic frequency such that the interrupt queue does not wrap, and queue entries are lost. There are 296 interrupt queue entries for F/W V4/5, and 512 queue entries for F/W V6 and later.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc, and start the Bus Controller using BusTools_BC_Start or BusTools_BC_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_ReadLastMessageBlock (cardnum, rt_addr_mask, subaddr_mask, tr, mcount, pBC_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rt_addr	(BT_INT) selects the RT address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
subaddress	(BT_INT) selects the sub address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.

tr	(BT_INT) select the transaction type as transmit or receive. 0 = Receive; 1 = Transmit; -1 = don't care.
mcount	(BT_UINT *) location to store the number of messages found.
pBC_mbuf	(API_BC_MBUF *) *) location where the message buffer contents will be written if a matching message is detected.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_READ_NODATA
API_BC_MBUF_NOMATCH

Notes

This function finds only Bus Controller messages that are in the interrupt queue. The interrupt queue records only BC messages that are programmed to generate interrupts.

To generate interrupts on BC messages, perform the following operatios during BC initialization:

1. Enable BC Interrupts by setting the *Enable* parameter to the desired interrupt type for BusTools_BC_Init. Set at least the BT1553_INT_END_OF_MESS bit in this word to ensure an interrupt on every BC message.
2. Set the BC_CONTROL_INTERRUPT bit in the bus message's control word when the bus message is written to BC memory allocated on this channel (see BusTools_BC_MessageWrite).

An array of API_BC_MBUF structures large enough to hold all the messages found by this function should be allocated by the application. The worst case is that a call to this function returns the entire interrupt queue. In that case, you need to pass an array of API_BC_MBUF structure with enough elements based on the F/W version of the board.

Example

The following code shows how to use this call.

```
API_BC_MBUF mbuf[296];
int i;
BT_UINT mess_cnt;
BT_U32BIT bit = 1;
BT_UINT RT = bit << 5;  // Select RT address 5
BT_UINT SA = -1;        // Select any subaddress
BT_UINT TX_RX = 1       // Select a transmit transaction type
```

```
status = BusTools_BC_ReadLastMessageBlock(cardnum, RT, SA,
                                          TX_RX, &mess_cnt,
                                          mbuf);

if (status == 0)
{
   for(i = 0; i < mess_cnt; i++)
   {
      // loop through all messages found
```

The above code returns all the transmit messages to RT address 5.

4.36 BusTools_BC_ReadNextMessage

Description

BusTools_BC_ReadNextMessage returns the next Bus Controller message recorded in the interrupt queue that fits the criteria in the argument list. Specify an RT address, subaddress, and transmit or receive command. The *rt_addr* and *subaddress* input arguments are bit-encoded values. For example, to select RT0 or subaddress 0 use the LSB (0x0001). The function keeps control and continually polls the interrupt queue until it either finds a message matching the settings or times out. If the function does not find a matching message and times out, it returns API_BC_READ_TIMEOUT. Time critical applications should use this function with caution.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc, and start the Bus Controller using BusTools_BC_Start or BusTools_BC_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_ReadNextMessage (cardnum, timeout, rt_addr, subaddress, tr, pBC_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
timeout	(BT_UINT) timeout value in milliseconds. Valid range is 10 to 65,535.
rt_addr	(BT_INT) selects the RT address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
subaddress	(BT_INT) selects the sub address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
tr	(BT_UINT) select the transaction type as transmit or receive. 0 = Receive; 1 = Transmit; -1 = don’t care.
pBC_mbuf	(API_BC_MBUF *) location where the message buffer contents will be written if a matching message is detected.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_MBUF_NOMATCH
API_BC_READ_TIMEOUT

Notes

This function will only search for Bus Controller messages in the interrupt queue, where only those BC messages configured to generate interrupts will reside.

To generate interrupts on BC messages, perform following BC initialization:

1. Enable BC Interrupts by setting *Enable* in the call to BusTools_BC_Init. Set at least BT1553_INT_END_OF_MESS in this word to ensure an interrupt on every BC message.
2. Set the BC_CONTROL_INTERRUPT bit in the bus message's control word when the bus message is written to BC memory allocated on this channel (see BusTools_BC_MessageWrite).

Timing accuracy differs between systems, where PC systems accuracy may be no better than 10 milliseconds. Timing accuracy of the host system must be considered when selecting a timeout value, especially when developing a deterministic application.

Example

The following code shows how to use this function.

```
API_BC_MBUF mbuf;

BT_UINT timeout;

Timeout = 100; // 100 millisecond timeout

status = BusTools_BC_ReadNextMessage(cardnum, timeout,
                                     0x40, -1, 1, &mbuf);

if (status == 0)
{
   // mbuf will contain the message
```

The above code returns the next transmit message to RT address 6.

4.37 BusTools_BC_RetryInit

Description

BusTools_BC_RetryInit configures the hardware retry attribute on a channel BC function. Hardware retry automatically retransmits a message if any pre-selected error conditions occur.

Three functions provide a method to configure the hardware retry attribute:

- BusTools_BC_Init *wRetry* parameter. This function parameter defines the conditions where a retry will occur. For example, use this parameter to retry on SRQ. BusTools_BC_Init configures a single retry on either the same bus as the command causing the retry or on the opposite bus.
- BusTools_BC_MessageWrite. Enables a retry on a message-by-message basis by setting BC_CONTROL _RETRY in the BC control word. This allows control on which individual BC messages will attempt a retry. The API allows retry on any 1553 message except for aperiodic messages. See BusTools_BC_MessageWrite for details of configuring BC messages.
- BusTools_BC_RetryInit. This function can configure from 1 to 8 actions in which a BC responds to a retry condition, with each action specifying which bus the channel transmits the retry. The actions are “retry on the same bus”, “retry on the alternate bus”, and “terminate retry attempt”.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_RetryInit (cardnum, bc_retry);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
bc_retry	(BT_U16BIT *) location of an array specifying retry options. Options are: RETRY_SAME_BUS RETRY_ALTERNATE_BUS RETRY_END

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

API_BC_NOTINITED
BTD_ERR_PARAM

Example

Setting Up A Multiple Retry Operation

The following code example programs three retry operations. The first retry occurs on the same bus as the original message. The next retry transmits on the alternate bus, and the last retry transmits back on the original bus. The hardware retry is programmed to occur only if there is a no-response or the RT returns a status word with either the busy bit or message error bit set. This example assumes that the bus list has been programmed to enable retries on all the messages and that the Abaco Systems 1553 board has been initialized.

```
BT_UINT retry[8];
wRetry = BC_RETRY_NRSP | BC_RETRY_BUSY | BC_RETRY_ME;
wStatus = BusTools_BC_Init ( cardnum, wChannel, Enable, wRetry,
          wTimeout1, wTimeout2, frame, num_buffers );
retry[0] = RETRY_SAME_BUS;
retry[1] = RETRY_ALTERNATE_BUS;
retry[2] = RETRY_SAME_BUS;
retry[3] = RETRY_END
wStatus = BusTools_BC_RetryInit( cardnum, retry );
```

4.38 BusTools_BC_SelectBufferRead

Description

Abaco Systems 1553 boards programmed with V6 firmware support Bus Controller message blocks with multiple data buffers. BusTools_BC_SelectBufferRead provides a method to read message data from a specific Bus Controller data buffer based on the corresponding message number and buffer number.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_BC_SelectBufferRead (cardnum, messno, buf_num, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messno	(BT_UINT) message number, (“0” based).
buf_num	(BT_UINT) buffer number, (“0” based).
buffer	(BT_U16BIT *) location to write the message data.

Return Value

API_SUCCESS
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT
API_BC_ILLEGAL_MBLOCK
API_BC_BAD_DATA_BUFFER
API_BC_UPDATEMESSTYPE

4.39 BusTools_BC_SelectBufferUpdate

Description

Abaco Systems 1553 boards programmed with V6 firmware support Bus Controller message blocks with multiple data buffers. BusTools_BC_SelectBufferUpdate uses the supplied message number and buffer number to update the data in a specific Bus Controller data buffer. This function only writes the data portion of the Bus Controller data buffer.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_BC_SelectBufferUpdate (cardnum, messno, buf_num, dcount, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messno	(BT_UINT) message number, (“0” based).
buf_num	(BT_UINT) buffer number, (“0” based).
dcount	(BT_UINT) number of words to write. Valid range is 1 to 32.
buffer	(BT_U16BIT *) location of the message data to write.

Return Value

API_SUCCESS
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT
API_BC_ILLEGAL_MBLOCK
API_BC_BAD_DATA_BUFFER
API_BC_UPDATEMESSTYPE

4.40 BusTools_BC_SetFrameRate

Description

The initial frame rate assigned to the BC function on a channel is configured via BusTools_BC_Init. BusTools_BC_SetFrameRate allows the application to modify the Bus Controller fame rate at any time. If invoked prior to starting the BC, the BC will use the new frame rate when started. If this function is invoked while the BC is executing, the new frame rate will take affect at the start of the next frame. With Abaco Systems 1553 boards programmed with V6 firmware the frame rate change may require an additional frame to take effect.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller via BusTools_BC_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_SetFrameRate (cardnum, frame);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
frame	(BT_U32BIT) minor frame period in microseconds. Valid range is 250 to 1638375 μs (0.61 and 4000.0 Hz).

Return Value

API_SUCCESS
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BUSTOOLS_BADCARDNUM
API_BC_BADFREQUENCY
API_HARDWARE_NOSUPPORT

Notes

The R15-USB does not support the ability to change BC frame rate while the BC is actively executing.

4.41 BusTools_BC_Start

Description

BusTools_BC_Start initiates BC execution at a specific message in the defined bus list.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions, and initialize the Bus Controller and message buffer(s) using BusTools_BC_Init and BusTools_BC_MessageAlloc.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_Start (cardnum, message_num);

wStatus (BT_INT) status returned from this function.

cardnum (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

message_num (BT_UINT) a 0-based message number referencing the entry in the BC message list at which the BC will begin processing messages. Valid range is 0 to 1 less than the number of messages allocated to the BC on this channel.

Return Value

API_SUCCESS
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_RUNNING
API_BUSTOOLS_BADCARDNUM
API_BC_ILLEGAL_MBLOCK

4.42 BusTools_BC_StartStop

Description

BusTools_BC_StartStop controls execution of the Bus Controller function on the specified channel. The default method to terminate BC execution is BC_STOP, supported by all firmware versions. As a result of executing the BC_STOP request the BC waits until the end of a minor frame before terminating processing. If the minor frame doesn't end within two seconds, the BC is forced to the stopped state. For Abaco Systems 1553 boards programmed with Firmware V6.x and later, the option BC_HALT is available, which will immediately terminate BC processing.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Controller using the BusTools_BC_Init function.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_StartStop (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_UINT) requested state for the Bus Controller: BC_STOP Stop BC processing at end of minor frame. BC_START Start BC processing at beginning of bus list. BC_HALT Stop BC processing immediately.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_NOTRUNNING
API_BC_RUNNING
API_BC_HALTERROR

4.43 BusTools_BC_Trigger

Description

BusTools/1553-API supports a programmable BC execution trigger option for software control of the Bus Controller bus list message processing. The function BusTools_BC_Trigger provides a method for the application to define this BC trigger mode.

The default BC trigger mode starts the Bus Controller immediately upon invocation of BusTools_BC_Start or BusTools_BC_StartStop(BC_START). If an application configures the BusTools/1553-API to any other trigger option, execution of the bus list will be delayed until receipt of an external trigger(s). An application should use this function to synchronize the Bus Controller to an external pulse. For information about hardware triggering of the Bus Controller, see the *BusTools/1553-API Software User's Manual*.

BusTools/1553-API supports four methods of triggering execution:

- BC_TRIGGER_IMMEDIATE: The Bus Controller will begin processing bus list immediately upon execution of BusTools_BC_Start or BusTools_BC_StartStop. There is no API control over BC execution with this default method.
- BC_TRIGGER_ONESHOT: Following execution of BusTools_BC_Start or BusTools_BC_StartStop, the API will initiate Bus Controller processing upon receipt of a trigger pulse on the hardware trigger input. Thereafter, it will process the bus list as programmed in the channel's BC configuration.
- BC_TRIGGER_REPETITIVE: Following execution of BusTools_BC_Start or BusTools_BC_StartStop, the Bus Controller will be enabled to process a single minor frame upon receipt of a trigger pulse on the hardware trigger input. The API will then pause the Bus Controller at the end of the minor frame, until receipt of the next trigger pulse. For a multi-frame bus list, the receipt of a trigger pulse will result in execution of each minor frame of the bus list in sequence. In this case, the frame rate programmed by BusTools_BC_Init must be greater than the trigger pulse frequency.
- BC_TRIGGER_USER: After the BC Run bit is set, upon receipt of a trigger pulse on the hardware trigger input, control over execution by the Bus Controller is defined according to the user frame setup. The user application/callback controls Bus Controller processing. The subsequent trigger pulse starts the Bus Controller running again. This mode requires the user application to setup a callback function via BusTools_RegisterFunction.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Controller using the BusTools_BC_Init function.

OS Support

Core API Function

Syntax

wStatus = BusTools_BC_Trigger (cardnum, trigger_mode);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
trigger_mode	(BT_INT) requested state for BC: BC_TRIGGER_IMMEDIATE BC_TRIGGER_ONESHOT BC_TRIGGER_REPETITIVE BC_TRIGGER_USER

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_BC_NOTRUNNING
API_BC_RUNNING
API_BC_HALTERROR

4.44 BusTools_BIT_CableWrap

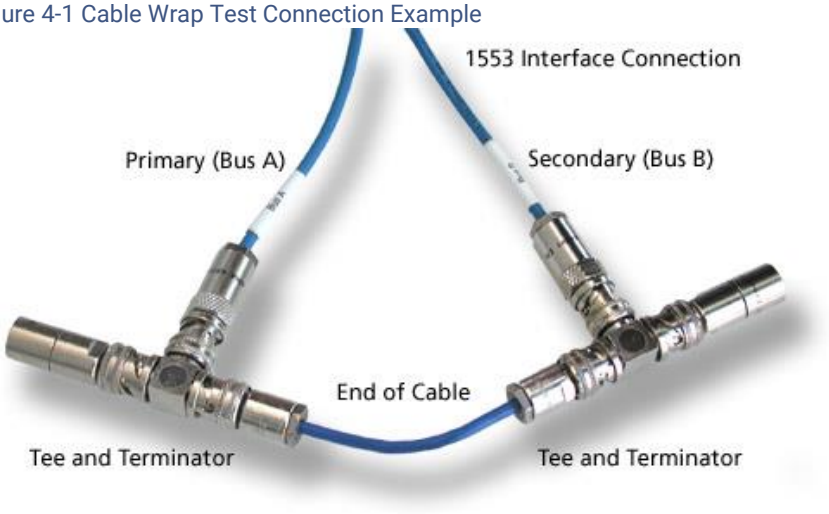
Description

BusTools_BIT_CableWrap performs a cable integrity test on the primary and secondary bus cables for a single 1553 channel. The primary and secondary buses on a single 1553 channel are referred to as Bus A and Bus B. Prior to executing this test, connect the primary and secondary bus cables from a single 1553 dual redundant channel in the manner as shown in Figure 4-1.

NOTE
Be sure to terminate the cable connection properly.

Figure 4-1 shows an example of a direct-coupled cable wrap test connection. This is not a legal MIL-STD-1553 configuration; however, transformer coupling may be used. Use this configuration only for the cable self-test. This test is supported on all single-, dual-, and multi-function 1553 boards.

Figure 4-1 Cable Wrap Test Connection Example



It is recommended an application execute the BusTools_BIT_InternalBit before executing BusTools_BIT_CableWrap. This combination of these two invocations will provide a complete test of 1553-interface integrity and eliminates the chance of an interface failure causing the cable wrap test to fail.

Do not attach the interface or cabling to any other 1553 interfaces while the test is running. You can use direct or transformer coupling depending on your channel termination configuration. Initialize the channel by calling one of the BusTools/1553-API Initialization functions and select direct or transformer coupling before calling this function.

This function enables the External Bus and configures the BC and API to execute *NumMessages* number of test iterations. For each iteration, the BC sends two transmit messages, the first on the primary bus and the second on the secondary bus. The function checks for the BT1553_INT_TWO_BUS status error. If that error is set,

the function returns API_SUCCESS; otherwise, it returns either API_BIT_FAIL_PRI or API_BIT_FAIL_SEC.

When this function returns the board is still open, but any previous board setup is lost. The application must restore all channel configuration to the desired settings.

OS Support

Core API Function

Syntax

wStatus = BusTools_BIT_CableWrap (cardnum, NumMessages);

wStatus (BT_INT) status returned from this function.

cardnum (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

NumMessages (BT_INT) number of test iterations to execute.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BIT_FAIL_PRI
API_BIT_FAIL_SEC

4.45 BusTools_BIT_InternalBit

Description

BusTools_BIT_InternalBit executes an internal wrap test of a single 1553 channel, with the number of test iterations desigated by the application via the *NumMessages* parameter. When this function completes the session with the board is still open, but the board configuration is lost. The application must restore all channel configuration to the desired settings.

Prior to invoking this function, call one of the BusTools/1553-API Initialization functions to initialize the channel.

OS Support

Core API Function

Syntax

wStatus = BusTools_BIT_InternalBit (cardnum, NumMessages);

wStatus (BT_INT) status returned from this function.

cardnum (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

NumMessages (BT_INT) number of messages to test.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BIT_FAIL_PRI
API_BIT_FAIL_SEC

4.46 BusTools_BIT_TwoBoardWrap

Description

BusTools_BIT_TwoBoardWrap performs an external wrap test between two 1553 devices. You can use two separate 1553 cards or 2-channels on a multi-channel interface card. Connect the primary bus of one channel to the primary bus of the other channel and the secondary bus of one channel to the secondary bus of the other channel. Wire and terminate both buses per MIL-STD-1553 requirements. Prior to calling this function, call one of the BusTools/1553-API Initialization functions to initialize each channel.

This function enables the external bus, sets one channel as a BC, and the other channel as an RT. For each of *NumMessages* iterations, the BC sends two transmit messages to the RT, one on the primary bus and one on the secondary bus. The software compares the data received by the BC to the data in the RT transmit buffers. If they match, the test passes. The BM data from both channels is also tested, and if the data match, the test passes. The test is then repeated, swapping the BC and RT functions. If all iterations of the test pass, this function returns API_SUCCESS; otherwise, it returns an error code.

When this function returns, the board is still open, but the board setup is lost. It is necessary the application invoke all setup functions again, beginning with BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BIT_TwoBoardWrap (FirstCard, SecondCard, TestPrimary, TestSecondary, NumMessages, RT_addr, RT_subaddr);

wStatus	(BT_INT) status returned from this function.
FirstCard	(BT_UINT) logical channel reference to the first 1553 board/channel session. Valid range is 0 to 63.
SecondCard	(BT_UINT) logical channel reference to the second 1553 board/channel session. Valid range is 0 to 63.
TestPrimary	(BT_UINT) Flag, if set test the primary bus, if clear test the secondary bus
TestSecondary	(BT_UINT) Flag, if set test the secondary bus, if clear test the primary bus
NumMessages	(BT_INT) number of messages to test.
RT_addr	(BT_INT) RT address to use during test.

RT_subaddr (BT_INT) RT subaddress to use during test.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BIT_FAIL_PRI
API_BIT_FAIL_SEC

4.47 BusTools_BIT_StructureAlignmentCheck

Description

BusTools_BIT_StructureAlignmentCheck checks for the correct structure alignment of critical structures used to transfer data to and from onboard memory. These structures must have 2-byte alignment and packing. If the alignment is incorrect, the data will be corrupted during the transfer.

There is also a version of this function for boards running firmware version 6.0 or greater. This function is BusTools_BIT_StructureAlignmentCheckV6. The version 6 firmware structures require a four-byte alignment. The calling arguments are the same.

This function is a check used for development on non-supported systems or modification of the build environment. You do not need to run this test on standard installation of supported systems.

Run this test at any time. It does not require initialization. As a software check, it does not require a board installed in the system.

OS Support

Core API Function

Syntax

wStatus = BusTools_BIT_StructureAlignmentCheck (BT_INT flag)

wStatus	(BT_INT) status returned from this function.
flag	(BT_INT) Print Flag 0 = no print; 1 = print structure sizes.

Return Value

API_SUCCESS
API_STRUCT_ALIGN

4.48 BusTools_BM_Checksum1760

Description

BusTools_BM_Checksum1760validates a MIL-STD-1760C implementation of a message received by the Bus Monitor 1553 function by comparing a calculated checksum of the buffer to the last location in the buffer (e.g., the received message checksum data word). The checksum is calculated according to the algorithm described in Appendix B Section B.4.1.5.2.1 of the *Department of Defense Interface Standard for Aircraft/Store Electrical Interconnect Systems* MIL-STD-1760C Manual.

The application is required to provide an API_BM_MBUF structure and a reference to unsigned short integer to store the calculated checksum.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_Checksum1760 (mbuf, cksum);

wStatus	(BT_INT) status returned from this function.
mbuf	(API_BM_MBUF) reference to a Bus Monitor message structure containing a MIL-STD-1760C message.
cksum	(BT_U16BIT *) Pointer to an unsigned short integer location to hold the calculated checksum.

Return Value

API_SUCCESS
API_BM_1760_ERROR

4.49 BusTools_BM_FilterRead

Description

The BM records messages detected on the 1553 bus subject to a filter on the command word of each message. Each possible subunit, a combination of RT address, subaddress, transmit/receive flag, and word count (or mode code), can be enabled or disabled for BM recording.

BusTools_BM_FilterRead reads the specified BM Filter buffer from channel memory for the specific RT subunit and returns the information to the caller-supplied structure. The enable/disable information is in the form of a 32-bit word. Each bit enables or disables a specific word count (or mode code) for the specified RT subunit.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using the BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_FilterRead (cardnum, rtaddr, subaddr, tr, cbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address (0 - 31), not bit-encoded.
subaddr	(BT_UINT) RT subaddress (0 - 31), not bit-encoded.
tr	(BT_UINT) Transmit/receive flag (0 = receive)
cbuf	(API_BM_CBUF*) pointer to BM filter structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_ILLEGAL_ADDR
API_BM_ILLEGAL_SUBADDR
API_BM_ILLEGAL_TRANREC

4.50 BusTools_BM_FilterWrite

Description

The BM records messages detected on the 1553 bus subject to a filter on the command word of each message. Each possible subunit, a combination of RT address, subaddress, transmit/receive flag, and word count (or mode code), can be enabled or disabled for BM recording.

BusTools_BM_FilterWrite creates or updates the Filter buffer information for the specified RT subunit. If this is the first call to this function for the specified RT subunit since the call to BusTools_BM_Init function, the API allocates the Filter buffer in the channel's Bus Monitor memory. If you have already allocated this Filter buffer, the API updates the buffer with the new information.

By default, the API enables all combinations of RT address, subaddress, transmit, receive, and word count. The API does this by pointing every entry of the filter buffer to a default BM filter buffer enabling all combinations. The API also creates a default BM filter buffer that disables all word count combinations. Calling this function to disable all word counts for a specified RT address, subaddress, and transmit/receive combination results in the use of this second default buffer.

Since the API enables all Bus Monitor addresses, it is not necessary to call this function unless you want to filter some RT address combinations. Not calling this function conserves memory on the Abaco Systems 1553 board.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using the BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_FilterWrite (cardnum, rtaddr, subaddr, tr, cbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address (0 - 31), not bit-encoded.
subaddr	(BT_UINT) RT subaddress (0 - 31), not bit-encoded.
tr	(BT_UINT) Transmit/receive flag (0 = receive)
cbuf	(API_BM_CBUF*) pointer to BM Filter Buffer (API_BM_CBUF).

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_MEMORY_OFLOW
API_BM_ILLEGAL_ADDR
API_BM_ILLEGAL_SUBADDR
API_BM_ILLEGAL_TRANREC

4.51 BusTools_BM_Init

Description

BusTools_BM_Init performs initialization of the Bus Monitor function on a channel. If full initialization is executed (default), this function also initializes and resets the memory management functions for this channel's memory. If the application enables the Bus Monitor function on a channel, this function must be invoked prior to initializing the Bus Controller or RemoteTerminal. This function also resets all error counters.

The bm_ctrl1 parameter allows the user to limit the level of initialization performed. This is useful if the application is required to reinitialize the Bus Monitor without affecting the RT or BC functions. The application can also elect to have the Bus Monitor refrain from generating hardware interrupts, where BM message will be entered in the interrupt queue, but the interrupt is not driven. The default configuration is for BM message to generate interrupts.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions. In addition, if you call this function after other BusTools/1553-API activity has occurred, the caller must ensure that the BC, BM, or RT is not currently running.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_Init (cardnum, bm_ctrl1, bm_ctrl2);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
bm_ctrl1	(BT_UINT) control parameter for initialization. BM_NO_SEG1_INIT – Omits seg1 initialization. BM_NO_HW_INT – Disables H/W interrupts (v6 Only) BM_DISABLE_BUS – Disables Monitoring on Bus A(v4 only)
bm_ctrl2	(BT_UINT) control parameter for initialization BM_DISABLE_BUS – Disables Monitoring on Bus B (v4 only)

Notes

Previous versions of this function had *enable_a* and *enable_b* parameters in place of bm_ctrl1 and bm_ctrl2. Starting at firmware version 5.0 and BusTools/1553-API v6.20 those options are no longer available. If you are running a F/W version below v5.0 and want to disable monitoring on Bus A or Bus B, use BM_DISABLE_BUS in bm_ctrl1 for bus A and bm_ctrl2 for bus B.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_RUNNING
API_BM_RUNNING

4.52 BusTools_BM_MessageAlloc

Description

BusTools_BM_MessageAlloc allocates and initializes the specified number of BM Message buffers in the channel memory allocated to the Bus Monitor function. If the specified number of message buffers doesn't fit, (or the caller specified "1" buffers), the function allocates as many buffers as will fit into the remaining memory. The function returns the number of message buffers allocated to the caller.

For Boards running the firmware version 6.x, there are no longer fixed size Bus Monitor messages. This function converts the requested number of buffers into a byte allocation by multiplying the requested buffer number by the maximum size message (32-words). The BM messages are then written end-to-end as they transact. When a message reaches the last buffer location it wraps to the starting buffer address. You can no longer use BusTools_BM_MessageGetAddr or BusTools_BM_MessageGetid since there are no defined message addresses. When using this function, you must pass a message address rather than message number. Message addresses are read from the interrupt queue. For a function that automatically reads the interrupt queue use BusTools_RegisterFunction or BusTools_BM_ReadLastMessageBlock.

Be sure to allocate enough message buffers or bytes to hold those messages that the BM receives. This depends on the expected level of 1553 message traffic to monitor. For the R15-USB board you will to need to allocate considerably more buffers than for other devices. On heavily loaded buses you may need to allocate 3000 buffers.

This function places the Interrupt Enable Bits specified by the caller into all message buffers.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor by calling BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_MessageAlloc (cardnum, mbuf_count, mbuf_actual, enable);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mbuf_count	(BT_UINT) requested number of BM Message buffers.

mbuf_actual (BT_UINT *) pointer to actual number of buffers allocated (returned by function).

enable (BT_U32BIT) interrupt enable flags for BM activity.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_RUNNING

4.53 BusTools_BM_MessageGetaddr

Description

BusTools_BM_MessageGetaddr converts a BM message number to a 1553 board hardware address offset. The returned address offset is a byte offset from the beginning of memory on the Abaco Systems 1553 board. The address offset range is from 0x00000000 to the largest memory offset supported by the board. Pass this address to BusTools_MemoryRead or BusTools_MemoryWrite to perform direct access to the BM message structure.

This function is not supported on boards programmed with F/W version 6.0. The version 6 firmware uses variable size BM buffers, and there are no fixed address offsets associated with BM buffers.

Normally, you don't need to access absolute memory addresses on the Abaco Systems 1553 board. Use this function for debugging operations. It is also possible to use this function in implementing operations that are not possible using the higher-level API functions.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using the BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_MessageGetaddr (cardnum, mbuf_id, addr);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mbuf_id	(BT_UINT) BM message number (0-based).
addr	(BT_U32BIT *) pointer to returned address of specified BM message.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_ILLEGAL_MBUFID

4.54 BusTools_BM_MessageGetid

Description

BusTools_BM_MessageGetid converts a BusTools hardware address to a BM Message buffer number. The passed address is a byte offset from the beginning of memory on the Abaco Systems 1553 board. The address range is "0-based" and has a range of 0x00000000 to the number of message buffers allocated by the BusTools_BM_MessageAlloc function, less one; with a maximum of 0x0003FFFF.

This function is not supported on boards programmed with F/W version 6.0. The version 6 firmware uses variable size BM buffers, and there are no fixed address offsets associated with BM buffers.

Typically, the interrupt thread handling the BM interrupt message uses this function. This function converts the BM message address to a message number. Then, use BusTools_BM_MessageRead to retrieve the message from the board.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_MessageGetid (cardnum, addr, messageid);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
addr	(BT_U32BIT) address of BM Message buffer (as supplied to the task handling BM interrupts).
messageid	(BT_UINT *) address of returned BM Message buffer number (returned by function).

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_MBUF_NOMATCH

4.55 BusTools_BM_MessageRead

Description

BusTools_BM_MessageRead transfers the contents of the specified BM Message buffer to the caller-supplied structure.

For boards programmed with Firmware V6 or later, the application must specify the buffer address offset of the buffer to read, (typically acquired from the interrupt queue). Use either BusTools_RegisterFunction with a user callback or BusTools_BM_ReadLastMessageBlock to obtain the buffer address offset.

For boards programmed with Firmware V5 or earlier, the application must specify a message buffer number in the range of the available message buffers as specified by the call to BusTools_BM_MessageAlloc.

This function returns the data in the API_BM_MBUF message buffer.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_MessageRead (cardnum, mbuf_id, mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mbuf_id	(BT_UINT) (Firmware V5 or earlier) BM Message buffer number (“0” based). (Firmware V6 or Later) BM Message Buffer address offset.
mbuf	(API_BM_MBUF *) Address of message buffer structure to be filled by this function.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_ILLEGAL_MBUFID

Notes

Using BusTools_RegisterFunction with a user callback to get the buffer address:

```
messno = sIntFIFO->fifo[tail].bufferID;

status = BusTools_BM_MessageRead(ch_id, messno, &mbuf);
```

4.56 BusTools_BM_MessageReadBlock

Description

BusTools_BM_MessageReadBlock transfers all BM Message buffers recorded in a channel Bus Monitor memory since the previous invocation of this function. The caller supplies a pointer to an array of API_BM_MBUF structures. The function returns the number of messages read from the buffer and written to that array. In addition, this function updates the cumulative error counters (to get the current error counter values, invoke the BusTools_ErrorCountGet function).

The API receives an event every 500 ms or whenever there are 64K bytes of message data available. This function never returns more than 65535 bytes of data in a single call. Your application should ensure the supplied array of message buffers is large enough to handle the volume of messages expected between calls.

You must call BusTools_RegisterFunction using the EVENT_RECORDER option prior to calling this function, or the API does not collect and save BM messages.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using BusTools_BM_Init.

OS Support

Core API function

Syntax

wStatus = BusTools_BM_MessageReadBlock (cardnum, api_mbuf, size, curpos, ret_count);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
api_mbuf	(API_BM_MBUF *) address of the beginning of the caller's message array.
size	(BT_UINT) number of message structures in the "pMessages" array.
curpos	(BT_UINT) current position within the "pMessages" array (this is the message number within the array where this function stores the first message transferred).
ret_count	(BT_UINT *) number of messages transferred returned by function.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_WRAP_AROUND

4.57 BusTools_BM_ReadLastMessage

Description

BusTools_BM_ReadLastMessage returns the last Bus Monitor message recorded in the interrupt queue that fits the criteria in the argument list. Specify an RT address, subaddress, and transmit or receive. The RT address and subaddress input arguments are bit-encoded values. For example, to select RT0 or subaddress 0 use the LSB (0x0001). The function then searches backwards in the interrupt queue for a message matching the settings. The function returns API_BM_READ_NODATA if no matching message is found.

On the initial call, this function searches the entire interrupt queue for messages. On later calls, it searches only the section between the current queue pointer and the queue pointer on the last call. For best results, you must call this function at a high enough rate to prevent the interrupt queue from overflowing. There are 296 interrupt queue entries for F/W V4/5, and 512 queue entries for F/W V6 and later.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions. Initialize the Bus Monitor by calling BusTools_BM_Init and start the Bus Monitor by calling BusTools_BM_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_ReadLastMessage (cardnum, rt_addr, subaddress, tr, pBM_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rt_addr	(BT_INT) selects the RT address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
subaddress	(BT_INT) selects the subaddress (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
tr	(BT_INT) select the transaction type as transmit or receive. 0 = Receive; 1 = Transmit; -1 = don’t care.
pBM_mbuf	(API_BM_MBUF *) pointer to a BM Message Buffer (API_BM_MBUF) structure. The function fills this structure if it finds a matching message.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_READ_NODATA
API_BM_MBUF_NOMATCH

Notes

This function finds only Bus Monitor messages that are in the interrupt queue. The interrupt queue records BM messages only if the BM is programmed to generate interrupts during BM initialization. To do this, set the *Enable* parameter in the BusTools_BM_MessageAlloc call to the desired interrupts, including at least BT1553_INT_END_OF_MESS

Example

The following code shows how to use this call.

```
API_BM_MBUF mbuf;
BT_U32BIT bit = 1;
BT_UINT RT = bit << 3;  // Select RT address 3
BT_UINT SA = -1;        // Select any subaddress
BT_UINT TX_RX =1;       // Select a transmit transaction type

status = BusTools_BM_ReadLastMessage(cardnum, RT, SA,
                                     TX_RX, &mbuf );
if (status == 0)
{
   // This is new data
```

The above code returns the last transmit message to RT address 3.

4.58 BusTools_BM_ReadLastMessageBlock

Description

BusTools_BM_ReadLastMessageBlock returns all the Bus Monitor messages in the interrupt queue that fit the criteria in the argument list. Specify an RT address, subaddress, and transmit or receive. The RT address and subaddress input arguments are bit-encoded values. For example, to select RT0 or subaddress 0 use the LSB (0x0001). The function searches the interrupt queue for all messages matching the settings. The function returns API_BM_READ_NODATA if a matching message is not found. Otherwise, the function returns a BM Message Buffer (API_BM_MBUF) array containing the found messages. The function also returns a count of the messages found.

On the initial call, this function searches the entire interrupt queue for messages. On later calls, it searches only the section between the current queue pointer and the queue pointer on the last call. You must call this function at a rate fast enough so the interrupt queue pointer or Bus Monitor buffers do not wrap. There are 512 interrupt queue entries for firmware version 6 and greater and 296 entries for firmware 3.x, 4.x and 5.x. The BM buffer size is set by BusTools_BM_MessageAlloc.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions. Initialize the Bus Monitor by calling BusTools_BM_Init and start the Bus Monitor by calling BusTools_BM_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_ReadLastMessageBlock (cardnum, rt_addr_mask, subaddr_mask, tr, mcount, pBM_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rt_addr	(BT_INT) selects the RT address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
subaddress	(BT_INT) selects the subaddress (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
tr	(BT_INT) select the transaction type as transmit or receive. 0 = Receive; 1 = Transmit; -1 = don’t care.

mcount	(BT_UINT *) pointer to that holds the count of messages found.
pBM_mbuf	(API_BM_MBUF *) pointer to an array of BM Message Buffer (API_BM_MBUF) structures. The function fills these structures if it finds matching messages.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_READ_NODATA
API_BM_MBUF_NOMATCH

Notes

This function will only search for Bus Monitor messages in the interrupt queue, where only those BM messages configured to generate interrupts will reside. To confgire this on a channel, set the BusTools_BM_MessageAlloc *Enable* parameter to the desired interrupts, including at least BT1553_INT_END_OF_MESS.

You must provide an array of API_BM_MBUF structures large enough to hold all the messages found by this function. The worst case is that a call to this function returns the entire interrupt queue. In that case, you need to pass an array of API_BM_MBUF structure with 512 elements.

Example

The following code shows how to use this call.

```
API_BM_MBUF mbuf[296];
int i;
BT_U32BIT bit = 1;
BT_UINT RT = bit << 4;  // Select RT address 4
BT_UINT mess_cnt;
BT_UINT SA = -1;        // Select any subaddress
BT_UINT TX_RX = 1;      // Select a transmit transaction type
status = BusTools_BM_ReadLastMessageBlock(cardnum, RT, SA,
                                          TX_RX, &mess_cnt,
                                          mbuf);
if (status == 0)
{
   for(i = 0; i < mess_cnt; i++)
   {
      // loop through all messages found
```

The above code returns all the transmit messages to RT address 4.

4.59 BusTools_BM_ReadNextMessage

Description

BusTools_BM_ReadNextMessage returns the next Bus Monitor message recorded in the interrupt queue that fits the criteria in the argument list. Specify an RT address, subaddress, and transmit or receive. The RT address and subaddress input arguments are bit-encoded values. For example, to select RT0 or subaddress 0 use the LSB (0x0001).

This function continually polls the interrupt queue until the timeout period expires or it finds a matching message. This function keeps control until it finds a matching message or times out. If the function does not find a matching message and times out, it returns API_BM_READ_TIMEOUT. Time critical applications should use this function with caution.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions. Initialize the Bus Monitor by calling BusTools_BM_Init and start the Bus Monitor by calling BusTools_BM_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_ReadNextMessage (cardnum, timeout, rt_addr, subaddress, tr, pBM_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
timeout	(BT_UINT) timeout value in milliseconds. Valid range is 10 to 65,535.
rt_addr	(BT_INT) selects the RT address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
subaddress	(BT_INT) selects the subaddress (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
tr	(BT_INT) select the transaction type as transmit or receive. 0 = Receive; 1 = Transmit; -1 = don’t care.
pBM_mbuf	(API_BM_MBUF *) pointer to a BM Message Buffer (API_BM_MBUF) structure. The function fills this structure if it finds a matching message.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_MBUF_NOMATCH
API_BM_READ_TIMEOUT

Notes

This function will only search for Bus Monitor messages in the interrupt queue, where only those BM messages configured to generate interrupts will reside. The interrupt queue records BM messages only if the BM is programmed to generate interrupts during BM initialization. To do this, set the *Enable* parameter in the BusTools_BM_MessageAlloc call to the desired interrupts, including at least BT1553_INT_END_OF_MESS

Timing accuracy differs between systems. Usually, most PC systems have accuracy no better than10 milliseconds. You must consider the timing accuracy of your system when selecting a timeout value, especially if you are developing a deterministic application.

Example

This function hides the structure of the interrupt queue. The following code shows how to use this call.

```
API_BM_MBUF mbuf;

BT_UINT timeout;

Timeout = 100; // 100 millisecond timeout
BT_U32BIT bit = 1;
BT_UINT RT = bit << 8;  // Select RT address 8

status = BusTools_BM_ReadNextMessage(cardnum, timeout,
                                     RT, -1, 1, &mbuf);
if ( status == 0)
{
   // Data return
```

The above code returns the next transmit message to RT address 8.

4.60 BusTools_BM_SetRT_RT_INT

Description

BusTools_BM_SetRT_RT_INT allows an application to select which RT on an RT→RT transaction the Bus Monitor will generate an interrupt. An RT→RT message has both transmit and receive RTs. The default setting is for the BM to interrupt on the receiving RT message completion. Using this function, you can select to interrupt on the transmitting RT.

OS Support

Core API Function.

Syntax

wStatus = BusTools_BM_SetRT_RT_INT (cardnum, iflag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
iflag	(BT_UINT) requested state for BM: 0 = interrupt on the receiving RT. 1 = interrupt on the transmitting RT.

Return Value

API_SUCCESS

4.61 BusTools_BM_StartStop

Description

BusTools_BM_StartStop starts and stops the Bus Monitor. The parameter *flag* specifies the state. You can also start and reset the time-tag.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_StartStop (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_UINT) requested state for BM: BM_STOP = Stop the BM. BM_START = Start the BM. BM_START_TT_RESET = BM Start + Time-Tag Reset.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_BM_NOTRUNNING
API_BM_RUNNING

4.62 BusTools_BM_TriggerWrite

Description

BusTools_BM_TriggerWrite writes the contents of the caller-supplied API_BM_TBUF structure to the BM Trigger buffer in channel memory. The structure defines the BM trigger options.

Using the API_BM_TBUF structure, you may select different triggering options: external trigger inputs, external trigger outputs, triggering on a command word (CW) or combination of command word, status word, or data word. For complete details, see the *BusTools/1553-API Software User's Manual* and *MIL-STD-1553 Universal Core Architecture Reference Manual*.

When the bus monitor is running, it captures 1553 messages and stores them in BM buffers. The interrupt queue is updated for each message as long as the BM interrupt is enabled. BM Triggering allows the user to enable/disable Bus Monitor interrupts when the specified trigger event occurs. A trigger event may be any selected command, status or data word on the 1553 bus, a logical AND/OR of multiple bus events, an ARMING/ARMED bus event combination or an external pulse.

"Event 1 AND Event 2" implies that both events must happen in any order before the trigger occurs.

"Event 1 ARMED BY Event 2" implies that Event 2 must happen before Event 1 for the trigger to occur.

"Event 1 LINKED TO Event 2" or "Event 1 WITH Event 2" indicates that Event 1 is a command word and Event 2 is a data word or status word in the message specified by Event 1.

"Event 1 OR Event 2" implies that the trigger occurs when either Event 1 or Event 2 occurs.

A trigger event or event combination can be used as a start trigger or a stop trigger. When a start trigger is used, no BM messages are inserted into the interrupt queue until the trigger event occurs. When a stop trigger is used, BM messages stop going into the interrupt queue when the trigger event occurs. While waiting for the trigger event, 1553 bus traffic continues, and information is recorded in the BM buffers. The trigger event affects only the BM interrupt enable bit. Both start and stop triggers can be set up with the same BusTools_BM_TriggerWrite call.

The API_BM_TBUF structure contains definitions for four start trigger events and four stop trigger events. Start events are programmed by modifying the four "capture" array entries (Events 1-4 or A-D) in the API_BM_TBUF. Stop events are programmed by modifying the four "stop" array entries (Events 1-4 or E-H) in the API_BM_TBUF. The type of start event is programmed by setting the "control1" variable to a code from the following table. The type of stop event is programmed

by setting the "control2" variable to a code from the following table. **Note** that these codes are different from those in the *MIL-STD-1553 Universal Core Architecture Reference Manual*. Use these codes when the BusTools/1553-API controls BM Triggering.

Table 4-1 Trigger Event Codes

Code	Trigger Event
0	Start Always
1	If Event1
2	If Event1 AND Event2
3	If Event1 OR Event2
4	If Event1 LINKED TO Event2
5	If Event2 ARMED BY Event1
6	If (Event2 LINKED TO Event1) AND Event3
7	If (Event2 LINKED TO Event1) OR Event3
8	If Event3 ARMED BY (Event1 LINKED TO Event2)
9	If (Event1 LINKED to Event2) ARMED BY Event3
10	If Event 1 OR Event2 OR Event3
11	If Event1 AND Event2 AND Event3
12-15	Reserved

Each "capture" or "stop" array entry contains five variables: type, mask, value, word, and count.

Trigger Type – Set to 0 (no trigger), 1 (trigger on command word), 2 (trigger on status word), 3 (trigger on data word), 4 (not used), 5 (trigger on the lsb of the BM interrupt status word*), or 6 (trigger on the msb of the BM interrupt status word*). *F/W version 4.22 or higher.

Mask – The data selected by the trigger type is masked using this word.

Value – Value of the command, status or data word for triggering

Word – The number of the data word within the message to test for trigger. To select the first word in the data buffer, use 0x0.

Count – The number of times the event must occur before the trigger is recognized. To stop on the first occurrence, this value must be set to 0x1.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_BM_TriggerWrite (cardnum, tbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
tbuf	(API_BM_TBUF*) address of the Trigger buffer structure to be transferred to channel memory allocated to the Bus Monitor function.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED

Notes

Trigger On Data setup requirements:

- You must supply the command word, the data word, data value and the bit mask to the firmware. The command word is the first event and the data word is the second event.
- To enable the discrete TTL output trigger, set the “trig_ext_output” variable in the “API_BM_TBUF” structure to 1 for a single trigger, and 2 for a repetitive trigger output.
- To enable the external trigger TTL input, set the “trig_ext” variable in the “API_BM_TBUF” to a one.

4.63 BusTools_BoardHasIRIG

Description

BusTools_BoardHasIRIG determines if the channel reference has access to IRIG timer/time-stamp functionality on the associated board. It will return the value API_FEATURE_SUPPORT if the channel is IRIG enabled, API_SUCCESS if the channel is not IRIG enabled, or an applicable error status code otherwise.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_BoardHasIRIG (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_SUCCESS
API_FEATURE_SUPPORT
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.64 BusTools_BoardIsMultiFunction

Description

BusTools_BoardIsMultiFunction returns the 1553 channel functionality available with the specified channel reference. The 1553 channel functionality is defined as API_SINGLE_FUNCTION, API_DUAL_FUNCTION, or API_MULTI_FUNCTION, depending on the encompassing board configuration on which the channel referenced is hosted.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_BoardIsMultiFunction (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_SINGLE_FUNCTION
API_DUAL_FUNCTION
API_MULTI_FUNCTION
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.65 BusTools_BoardIsUSBMon

Description

BusTools_BoardIsUSBMon reports whether the channel referenced is hosted on an R15-USB-MON monitor only board. The function return status will be API_FEATURE_SUPPORT if the board hosting the channel is an R15-USB-MON, API_SUCCESS if the board hosting the channel is not an R15-USB-MON, or an applicable error status code otherwise.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_BoardIsUSBMon (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_FEATURE_SUPPORT
API_SUCCESS

4.66 BusTools_BoardIsV6

Description

BusTools_BoardIsV6 reports whether the channel referenced is hosted on a board programmed with UCA32 V6 firmware. The function return status will be API_FEATURE_SUPPORT for a board running V6 firmware, API_SUCCESS if the board hosting the channel is not running V6 firmware, or an applicable error status code otherwise.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_BoardIsV6 (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_FEATURE_SUPPORT
API_SUCCESS

4.67 BusTools_Checksum1760

Description

BusTools_Checksum1760 calculates a checksum according to the algorithm described in Appendix B Section B.4.1.5.2.1 of the *Department of Defense Interface Standard for Aircraft/Store Electrical Interconnect Systems* MIL-STD-1760C Manual.

When each data word (including the checksum word) of a message is rotated right cyclically by a number of bits equal to the number of preceding data words in the message, and all the resultant rotated data words are summed using modulo 2 arithmetic to each bit (no carries), the sum shall be zero.

The application will pass a pointer to the respective generic message data array and the number of words in the message to include in the calculation. This function will calculate and return the checksum in the function return value.

The following examples demonstrate messages satisfying the checksum algorithm.

Example 1

Four Word Message:

1st Word	0000-0000-0000-0001 (0001 hex.) data
2nd Word	1100-0000-0000-0000 (C000 hex.) data
3rd Word	0000-1111-0000-0000 (0F00 hex.) data
4th Word	0001-1110-0000-1011 (lE0B hex.) checksum word

Example 2

Six Word Message:

lst Word	0001-0010-0011-0100 (1234 hex.) data
2nd Word	0101-0110-0111-1000 (5678 hex.) data
3rd Word	1001-1010-1011-1100 (9ABC hex.) data
4th Word	1101-1110-1111-0000 (DEF0 hex.) data
5th Word	0000-0000-0000-0000 (0000 hex.) data
6th Word	1000-1111-0010-0000 (8F20 hex.) checksum word

OS Support

Core API Function

Syntax

wChksum = BusTools_Checksum1760 (mbuf, wdcnt);

wChksum	(BT_INT) the calculated checksum value.
mbuf	(BT_U16BIT *) pointer to array of message data.
wdcnt	(BT_U16BIT) number of elements in the array.

Return Value

1760 Checksum

4.68 BusTools_CreateIntFifo

Description

BusTools_CreateIntFifo creates an Interrupt Function Structure and assigns the supplied function reference as the User Interrupt Service Routine instantiated in a new thread.

The application invokes this function, providing a pointer to the API_INT_FIFO structure. This function will dynamically allocate memory for the API_INT_FIFO structure and assign a reference to the supplied callback-function. When the API_INT_FIFO structure is created, the callback-function reference is stored within the FIFO structure.

This function is used for C# wrapper compatibility only and is not needed for any other application.

OS Support

Windows C# compatibility only

Syntax

pFifo = BusTools_CreateIntFifo (*function(cardnum, *pFIFO));

pFifo	(API_INT_FIFO *) an API_INT_FIFO handle.
function	(BT_INT (_stdcall *function)(BT_UINT cardnum, struct api_int_fifo *pFIFO)) function pointer designating the ISR callback function.

Return Value

Pointer to a structure of the type API_INT_FIFO

4.69 BusTools_DestroyIntFifo

Description

BusTools_DestroyIntFifo relinquishes the dynamically allocated memory resources containing a previously created API_INT_FIFO structure via invocation of BusTools_CreateIntFifo.

This function is used for C# wrapper compatibility only and is not needed for any other application.

OS Support

Windows C# compatibility only

Syntax

void BusTools_DestroyIntFifo(API_INT_FIFO *pFifo);

pFifo	(API_INT_FIFO *) reference to an existing Interrupt Function Structure.

Return Value

Not Applicable

4.70 BusTools_DataGetString

Description

BusTools_DataGetString performs engineering unit conversions as specified and places the result in a string. The input parameter is a structure containing all the information needed to perform the conversion. This structure is shown below:

```
typedef struct data_convert
   {
   BT_U16BIT  wDatatype;
   BT_U16BIT  wDecimals;
   float fFactor;
   BT_U16BIT  wFactortype
   float fOffset;
   BT_U16BIT  trlateItems;
   UINT  *uiTranslateRaw;
   float *fTranslateDis;
   void* value;
   BT_INT status;
   char * string;
   }
DATA_CONVERT;
```

wDatatype – (Used for ALL conversions) – This field determines what type of conversion is to be done. The table below shows the constants (defined in busapi.h) that can be used for this field and describes the associated conversions.

Table 4-2 wDatatype Constants

Constant	Conversion
DATATYPE_16_SDEC	16-bit signed decimal. Uses the fields wDatatype, value, status, string.
DATATYPE_16_UDEC	16-bit unsigned decimal. Uses the fields wDatatype, value, status, string.
DATATYPE_HEX	16-bit hexadecimal. Uses the fields wDatatype, value, status, string.
DATATYPE_16_OCTAL	16-bit octal. Uses the fields wDatatype, value, status, string.
DATATYPE_16_BINARY	16-bit binary. Uses the fields wDatatype, value, status, string.
DATATYPE_16_BCD	16-bit binary coded decimal. Uses the fields wDatatype, value, status, string.
DATATYPE_16_BCD_2	Converts the 16-bit value to a string with the high byte in BINARY and the low byte in binary coded decimal. Uses the fields wDatatype, value, status, string.
DATATYPE_16_USCALE	16-bit unsigned scaled value. Uses the fields wDatatype, wDecimals, fFactor, fOffset, value, status, string.
DATATYPE_16_SSCALE	16-bit signed scaled value. Uses the fields wDatatype, wDecimals, fFactor, fOffset, value, status, string.
DATATYPE_16_TRANSLATE	Conversion using a 16-bit lookup table. Uses the fields wDatatype, trlateItems, uiTranslateRaw, fTranslateDis, value, status, string.

Constant	Conversion
DATATYPE_32_SDEC	32-bit signed decimal. Uses the fields wDatatype, value, status, string.
DATATYPE_32_MSWF_SDEC	32-bit signed decimal with most significant word first. Uses the fields wDatatype, value, status, string.
DATATYPE_32_UDEC	32-bit unsigned decimal. Uses the fields wDatatype, value, status, string.
DATATYPE_32_MSWF_UDEC	32-bit unsigned decimal with most significant word first. Uses the fields wDatatype, value, status, string.
DATATYPE_32_IEEE	32-bit IEEE floating point. Uses the fields wDatatype, value, status, string.
DATATYPE_32_HEX	32-bit hexadecimal. Uses the fields wDatatype, value, status, string.
DATATYPE_32_BCD	32-bit binary coded decimal. Uses the fields wDatatype, value, status, string.
DATATYPE_32_USCALE	32-bit unsigned scaled value. Uses the fields wDatatype, wDecimals, fFactor, fOffset, value, status, string.
DATATYPE_32_MSWF_USCALE	32-bit unsigned scaled value with most significant word first. Uses the fields wDatatype, wDecimals, fFactor, fOffset, value, status, string.
DATATYPE_32_SSCALE	32-bit signed scaled value. Uses the fields wDatatype, wDecimals, fFactor, fOffset, value, status, string.
DATATYPE_32_MSWF_SSCALE	32-bit signed scaled value with most significant word first. Uses the fields wDatatype, wDecimals, fFactor, fOffset, value, status, string.
DATATYPE_32_1750	32-bit MIL-STD-1750 floating point. Uses the fields wDatatype, value, status, string.
DATATYPE_32_TRANSLATE	Conversion using a 32-bit lookup table. Uses the fields wDatatype, trlateItems, uiTranslateRaw, fTranslateDis, value, status, string.
DATATYPE_48_LATLONG	48-bit latitude/longitude. Uses the fields wDatatype, value, status, string.

wDecimals – (Used for scaled conversions) – This field determines the number of decimal points to show to the right of the decimal point.

fFactor – (Used for scaled conversions) – This field provides the number that is multiplied or divided into the raw data (determined by wFactortype, see below). This is the slope (m) in the linear equation "y = mx + b".

wFactortype – (Used for scaled conversions) – If this field is a 1, then the input data is MULTIPLIED by fFactor. If this field is a 2, then the input data is DIVIDED by fFactor.

fOffset – (Used for scaled conversions) – This value is added to the scaled value. This is the intercept (b) in the linear equation "y = mx + b".

trlateItems – (Used in defining the conversion) – This field determines the number of items in the lookup table.

uiTranslateRaw – (Used in defining the conversion) – This field provides a pointer to the table of raw data values for the lookup.

fTranslateDis – (Used in defining the conversion) – This field provides a pointer to the table of data entries corresponding to the raw values.

value – (Used for ALL conversions) – This field provides a pointer to the source data to be used for the conversion. This normally points to the appropriate location in an array of 16-bit data words from a 1553 message.

status – (Used for ALL conversions) – This field is used to return a status code indicating success or failure of the function call. See the explanation ofReturn Value below.

string – (Used for ALL conversions) – This field provides a pointer to the location where the user would like the resulting string to be stored.

OS Support

Core API Function

Syntax

pString = BusTools_DataGetString (cdat);

pString	(char *) reference to the resulting string.
cdat	(DATA_CONVERT *) reference to a structure containing the conversion parameters.

Return Value

This function returns a reference to the string containing the conversion result, (to the memory allocated in the DATA_CONVERT structure *string* member).

The DATA_CONVERT structure also contains a status field, which returns additional information. This field is one of the following:

API_SUCCESS
API_BADDATATYPE
API_BADBCDDATA
API_BADFACTORTYPE
API_BADTRANSLATE

Example

```
#include "busapi.h"
#include <stdio.h>
void main
{
   DATA_CONVERT dc_info;
```

```
   int i;
   BT_U16BIT    data[32];
   char         string[128];
   // Initialize variables.
   memset( &dc_info, 0, sizeof(dc_info) );
   for (i = 3; i < 32; i++) {
      data[i] = 0;
   }
   // Setup data conversion info structure.
   data[0] = 0x1234;
   data[1] = 0x0056;
   data[2] = 0x0078;
   dc_info.wDatatype = DATATYPE_48_LATLONG;
   dc_info.value = data;
   dc_info.string = string;

   // Do the conversion.
   printf("Raw: %04X %04X %04X,  String: %s\n", data[0],
          data[1], data[2], BusTools_DataGetString(&dc_info));
}
```

4.71 BusTools_DiffTriggerOut

Description

Select Abaco System's 1553 boards programmed with Version 4.x or 5.x firmware provide differential I/O channel trigger capabilities (see Table 1-2). This function provides a method a configure any available differential I/O channel as an output driven as "external trigger out". Invoke this function passing the channel flag, which connects or disconnects the channel reference to the respective differential I/O channel. The differential enable flag enables or disables differential I/O output.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the Differential I/O.

OS Support

Core API Function (F/W Version 4.x or 5.x only)

Syntax

wStatus = BusTools_DiffTriggerOut (cardnum, chflag, diffen);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
chflag	(BT_INT) 0 = disconnect 1 = connect channel.
diffen	(BT_INT) differential enable 0 = disable output 1 = enable output

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.72 BusTools_DiscreteGetIO

Description

BusTools_DiscreteGetIO returns the configuration of the Avionics Discrete I/O on Abaco Systems 1553 boards supporting Avionics Discretes (see Table 1-2). The returned discrete state is a bit encoded value with 1 representing an output line and 0 representing an input line. The bit position corresponds to the discrete line, where bits 0 (0x0001) through 18 (0x20000) represent discrete lines 1 through 18, depending on discrete count available for your board type.

Discrete I/O lines can be programmed as either input or output via invocation of BusTools_DiscreteSetIO.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the Avionics Discretes.

OS Support

Core API Function

Syntax

wStatus = BusTools_DiscreteGetIO (cardnum, disDirValue);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
disDirValue	(BT_U32BIT *) reference to a value containing the bitwise representation of the direction assignment for each available discrete, where 1 = output, 0 = input.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

Notes

If the board uses Hardwired RT Addressing on channel 1, the first six discrete lines (bits 0-5) are the RT Address. The QPMC, QPM, AMC, and RCPIE-1553 also support Hardwired RT Addressing on Channel 2. In that case, discrete lines 9-14 (bits 8-13) are the channel 2 RT Address. The application should consider Hardwired RT addressing when programming the discrete lines. Hardwired RT addressing overrides any discrete I/O setting.

4.73 BusTools_PIO_GetIO

Description

BusTools_PIO_GetIO returns the configuration of the programmable I/O lines on Abaco Systems 1553 boards that support PIO (see Table 1-2). The returned PIO state is a bit encoded value with 1 representing an output line and 0 representing an input line. The bit position corresponds to the PIO line, where bits 0 through 7 represent PIO lines 1–8. The actual number of PIO lines available is dependent on the 1553 board type installed.

PIO lines can be programmed as either input or output via invocation of BusTools_PIO_SetIO.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the Programmable I/O.

OS Support

Core API Function (F/W Version 4.x and Version 5.x only)

Syntax

wStatus = BusTools_PIO_GetIO (cardnum, pDirVal);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
pDirVal	(BT_U32BIT *) reference to a value containing a bitwise representation of the programmable I/O direction assignment, where 1 = output, 0 = input.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.74 BusTools_DiscreteRead

Description

BusTools_DiscreteRead returns the current value of the discrete input. The application can request the API return all input discrete inputs or specify reading individual discrete inputs. See Table 1-2 for the list of boards that support Avionics Discretes. If the application selects a single discrete input, the function returns the *disValue* parameter with either 0 or 1, corresponding to the state of the input. If the application selects all discrete inputs, the function returns with a bit-wise value indicating the state of all input lines. When selecting all discrete inputs, note only bits corresponding to inputs are valid; bits corresponding to outputs will be zero. Invoke the function BusTools_GetDiscreteIO to determine the configuration of the discrete I/O.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the Avionics Discretes.

OS Support

Core API function

Syntax

wStatus = BusTools_DiscreteRead (cardnum, disSel, disvalue);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
disSel	(BT_INT) Select the discrete/PIO input to read.
disValue	(BT_U32BIT *) reference to the location in which this function will store the input value. If a single input is specified, this value reflects the bit value corresponding to the input. If "read all inputs" is specified, this value is bit encoded with each input line bit set to either 0 or 1.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BAD_DISCRETE
API_HARDWARE_NOSUPPORT

Notes

If the board uses Hardwired RT Addressing on channel 1, the first six discrete lines (bits 0-5) are the RT Address. The QPMC, QPM, AMC, and RCPIE-1553 boards can also have Hardwired RT Addressing on Channel 2. In that case, discrete lines 9-14

(bits 8-13) are the channel 2 RT Address. The application should consider Hardwired RT addressing when programming the discrete lines. Hardwired RT addressing overrides any discrete I/O setting.

4.75 BusTools_PIO_Read

Description

BusTools_PIO_Read return sthe current value of the programmable input. The application can request the API return all input programmable inputs or specify reading individual programmable inputs. See Table 1-2 for the list of boards that support Avionics PIO. If the application selects a single programmable input, the function returns the *pioValue* parameter with either 0 or 1, corresponding to the state of the input. If the application selects all programmable inputs, the function returns with a bit-wise value indicating the state of all input lines. When selecting all programmable inputs, note only bits corresponding to inputs are valid; bits corresponding to outputs will be zero. Invoke the function BusTools_PIO_GetIO to determine the configuration of the programmable I/O.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the Avionics PIO.

OS Support

Core API function

Syntax

wStatus = BusTools_PIO_Read (cardnum, pioSel, pioValue);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
pioSel	(BT_INT) Select the PIO input to read. An input select value of 0 reads all inputs; otherwise, this parameter will specify an individual input.
pioValue	(BT_U32BIT *) reference to the location in which this function will store the input value. If a single input is specified, this value reflects the bit value corresponding to the input. If "read all inputs" is specified, this value is bit encoded with each input line bit set to either 0 or 1.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BAD_DISCRETE
API_HARDWARE_NOSUPPORT

4.76 BusTools_DiscreteSetIO

Description

Most of Abaco Systems 1553 boards support Avionics Discrete I/O (see Table 1-2). BusTools_DiscreteSetIO provides a method to configure Discrete I/O lines as either inputs or outputs. All channels on a multi-channel board share these discrete lines. An application must allocate use of the Discrete I/O resources between channels residing on the same 1553 board.

Discrete I/O direction (input/output) is programmed by setting or clearing the respective bit in a 32-bit register, where each bit represents a discrete line. Set the respective bit to 1 to configure a Discrete line as an output or 0 to configure a Discrete line as an input. The API ignores any bits referenced above the maximum number of discrete lines on a board.

When invoking this function pass the discrete setting in *disFlag* and a mask value in *mask* that specifies which bits are set/cleared. This allows the application to preserve the Discrete I/O configuration from prior definition.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the discrete lines.

OS Support

Core API Function

Syntax

wStatus = BusTools_DiscreteSetIO(cardnum, disSet, mask);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
disSet	(BT_U32BIT) bitwise configuration of the individual discrete lines: 0 = input 1 = output
mask	(BT_U32BIT) Mask specifying the bit(s) to set/clear.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

Notes

If the board uses Hardwired RT Addressing on channel 1, the first six discrete lines (bits 0-5) are the RT Address. The QPMC, QPM, AMC, and RCPIE-1553 also support Hardwired RT Addressing on Channel 2. In that case, discrete lines 9-14 (bits 8-13) are the channel 2 RT Address. The application should consider Hardwired RT addressing when programming the discrete lines. Hardwired RT addressing overrides any discrete I/O setting.

Coding Example

The following example code demonstrates various ways to configure a board with 10 discrete I/O channels.

The following invocation configures the discretes as alternating outputs and inputs.

```
status = BusTools_DiscreteSetIO( cardnum, 0x0155, 0x03FF  );
```

The following invocation configures discretes 5 – 8 to outputs while preserving the setting of the remaining discretes.

```
status = BusTools_DiscreteSetIO( cardnum, 0x00F0, 0x00F0 );
```

The following invocation configures discretes 5 – 8 to inputs while preserving the setting of the remaining discretes.

```
status = BusTools_DiscreteSetIO( cardnum, 0, 0x00F0);
```

4.77 BusTools_PIO_SetIO

Description

Some of Abaco Systems 1553 boards support programmable I/O lines (see Table 1-2). BusTools_PIO_SetIO provides a method to configure programmable I/O lines as either inputs or outputs. All channels on a multi-channel board share these PIO lines. An application must allocate use of the programmable I/O resources between channels residing on the same 1553 board.

Programmable I/O direction is configured by setting or clearing the respective bit in a 32-bit register, where each bit represents a PIO line. Set the respective bit to 1 to configure programmable I/O line as an output or 0 to configure a programmable I/O line as an input. The API ignores any bits referenced above the maximum number of PIO lines on a board.

When invoking this function pass the bitwise direction setting in *disSet* and a mask value in *mask* that specifies which bits are set/cleared. This allows you to preserve the bit settings from previous calls to this function.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the discrete lines.

OS Support

Core API Function

Syntax

wStatus = BusTools_PIO_SetIO (cardnum, disSet, mask);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
disSet	(BT_U32BIT) bitwise configuration of the individual programmable I/O lines: 0 = input 1 = output
mask	(BT_U32BIT) Mask specifying the bit to set/reset.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

Coding Example

The following example code configures the programmable I/O as alternating outputs and inputs (on a Board with 8 PIO lines).

```
status = BusTools_PIO_SetIO ( cardnum, 0x0055, 0x00FF );
```

4.78 BusTools_DiscreteTriggerIn

Description

Some Abaco Systems boards have Avionics Discrete lines that can be configured for trigger input signals (see Table 1-2). BusTools_DiscreteTriggerIn provides the method to select the respective discrete/differential input as the trigger input for the specified channel. Options available for triggers include single-ended input from either discrete 7 or discrete 8 (not both), and an RS-485 differential input. When configuring either discrete line 7 or 8 as an input trigger/clock, the discrete must be configured as an input via invocation of BusTools_DiscreteSetIO.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the discrete lines.

OS Support

Core API Function

Syntax

wStatus = BusTools_DiscreteTriggerIn (cardnum, trigFlag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
trigFlag	(BT_INT) defines the channel trigger/clock input source: TRIGGER_IN_485 - RS-485 input TRIGGER_IN_DIS_7 - Discrete 7 TRIGGER_IN_DIS_8 - Discrete 8 TRIGGER_IN_NONE - remove a previous trigger input configuration

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.79 BusTools_DiscreteWrite

Description

BusTools_DiscreteWrite defines the current state of the discrete output. The application can request the API configure all outputs or a select individual output. If the selected option is to configure a single output, the function configures only the specified output to the respective value of *disFlag*. If the selected option is to configure all outputs, the function configures all outputs to the respective value of *disFlag*. Multiple outputs cannot be configured other than by configuring all outputs in a single invocation.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the discrete lines.

OS Support

Core API function.

Syntax

wStatus = BusTools_DiscreteWrite (cardnum, disSel, disFlag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
disSel	(BT_U32BIT) This value selects the output to configure. A value of 0 selects all output lines; otherwise, the value is a bitwise selection of an individual output.
disFlag	(BT_U32BIT) state applied to the output. Setting *disFlag* value to 1 turns the low-side switch "on", completing the circuit by shorting it to ground. Setting this value to 0 turns the low-side switch "off", opening the circuit; in the absence of any external connection the discrete output will be pulled to 3.3V by a weak internal pull-up resistor.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BAD_DISCRETE
API_HARDWARE_NOSUPPORT

Notes

If the board uses Hardwired RT Addressing on channel 1, the first six discrete lines (bits 0-5) are the RT Address. The QPMC, QPM, AMC, and RCPIE-1553 also support

Hardwired RT Addressing on Channel 2. In that case, discrete lines 9-14 (bits 8-13) are the channel 2 RT Address. The application should consider Hardwired RT addressing when programming the discrete lines. Hardwired RT addressing overrides any discrete I/O setting.

4.80 BusTools_PIO_Write

Description

BusTools_PIO_Write defines the current state of the programmable output. The application can request the API configure all outputs or a select individual output. If the selected option is to configure a single output, the function configures only the specified output to the respective value of *pioFlag*. If the selected option is to configure all outputs, the function configures all outputs to the respective value of *pioFlag*. Multiple outputs cannot be configured other than by configuring all outputs in a single invocation.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the discrete lines.

OS Support

Core API function.

Syntax

wStatus = BusTools_PIO_Write (cardnum, pioSel, pioFlag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
pioSel	(BT_U32BIT) This value selects the output to configure. A value of 0 selects all output lines; otherwise, the value is a bitwise selection of an individual output.
pioFlag	(BT_U32BIT) state applied to the output. For programmable outputs: Setting *pioFlag* to 1 sets the select output line high. Setting this value to 0 clears the selected output line.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BAD_DISCRETE
API_HARDWARE_NOSUPPORT

4.81 BusTools_DiscreteTriggerOut

Description

Some Abaco Systems boards have Avionics Discrete lines (discrete output 7 or 8), that can be configured for external trigger output signals, (see Table 1-2). This function provides the method to select the respective discrete output as the external trigger output. The discrete must be configured as an output via invocation of BusTools_DiscreteSetIO.

Initialize the channel using one of the BusTools/1553-API Initialization functions before using the discrete lines.

OS Support

Core API Function

Syntax

wStatus = BusTools_DiscreteTriggerOut (cardnum, disflag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
disflag	(BT_INT) defines the external trigger output TRIGGER_OUT_DIS_7 - use Discrete 7 as trigger out TRIGGER_OUT_DIS_8 - use Discrete 8 as trigger out TRIGGER_OUT_DIS_NONE – remove a previous trigger output configuration

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.82 BusTools_DMA_Setup

Description

BusTools_DMA_Setup configures Direct Memory Access (DMA) for both the QPCX-1553 and QCP-1553 boards hosting a PLX 9056 PCI bus interface. This function can only configure the board for host-initiated block DMA, host-side configuration is not required. For setup by the host application, a virtual address is not allowed for either the host or the board; the PLX-9056 PCI interface component requires a host memory physical address and a board address that is a local bus address, (not the physical address of the board). See the "Universal Core Architecture" manual to find the local bus offsets to RAM on the respective board.

Besides the host and local addresses, the application must provide a byte count, a direction of transfer, (either board to host or host to board), and the DMA channel (0 or 1) to use.

Initialize the channel using one of the BusTools/1553-API Initialization functions before setting up a DMA scenario.

OS Support

VxWorks and Windows. BusTools/1553-API version 5.90 or later.

Syntax

wStatus = BusTools_DMA_Setup (cardnum, dma_channel, host_addr, board_addr, byte_count, dma_flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
dma_channel	(BT_UINT) DMA channel 0 : ch 0; 1 or > : ch 1.
host_addr	(BT_U32BIT) physical host address.
board_addr	(BT_U32BIT) local board address.
byte_count	(BT_U32BIT) Number of bytes to transfer.
dma_flag	(BT_U32BIT) Direction of transfer 0 : board to host, 1 or > : host to board.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.83 BusTools_DumpMemory

Description

BusTools_DumpMemory writes the contents of the specified onboard data buffers and/or register sets to a specified file in ASCII format. The application selects the specific items to log by setting a bitwise value in the *region_mask* parameter, (bit values not defined are ignored). If the file already exists, this function will overwrite it.

DUMP_HWREG	Hardware Registers
DUMP_RAMREG	RAM Registers
DUMP_BCMESS	BC Message Buffers
DUMP_BMFILTER	BM Filter Buffer
DUMP_BMTRIGGER	BM Trigger Buffer
DUMP_BMMESSAGE	BM Message Buffers
DUMP_BMCONTROL	BM Control Buffers
DUMP_BMDEFCBUF	BM Default Control Buffer
DUMP_RTADDRESS	RT Address Buffer
DUMP_RTFILTER	RT Filter Buffer
DUMP_RTDATA	RT Control and Message Buffers
DUMP_EI	Error Injection data area
DUMP_IQ	Interrupt queue data area
DUMP_RTMBUF_DEF	RT Default Message Buffers
DUMP_RTCBUF_DEF	RT Default Control Buffers
DUMP_RTCBUF_BRO	RT Broadcast Control Buffers
DUMP_PCI_RTDATA	RT MBUF
DUMP_DIFF_IO	Discrete I/O registers
DUMP_ALL	All buffers listed above

This function requires a host operating system that supports a File System. For operating environments not supporting a file system, this function writes to stdout.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API initialization functions.

OS Support

Requires files system or console output.

Syntax

wStatus = BusTools_DumpMemory (cardnum, region_mask, file_name, heading);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
region_mask	(BT_U32BIT) mask of memory regions to dump.
file_name	(char *) pointer to the ASCII name of the output file created. If the API cannot create or write to this file, an error is returned.
heading	(char *) pointer to a header string written to the output file.

Return Value

API_SUCCESS
API_BUSTOOLS_NO_FILE
API_BUSTOOLS_NOTINITED
API_BUSTOOLS_BADCARDNUM

4.84 BusTools_EI_EbufWrite

Description

For boards programmed with firmware version 4.66 or earlier, BusTools_EI_EbufWrite writes error injection data to the Error Injection buffer. See the "Error Injection" section in the *BusTools/1553-API Software User's Manual* for a complete description of the Error Injection buffer and its parameters.

This function is deprecated for firmware version 5.00 and greater, with the function BusTools_EI_EnhEbufWrite as its replacement. If a legacy application calls BusTools_EI_EbufWrite using BusTools/1553-API version 6.42 or greater, with a board running firmware 5.00 or greater, BusTools_EI_EnhEbufWrite will be invoked to perform the desired error injection buffer update. A board running firmware 5.00 or greater will not correctly generate all errors if the host application is built with a BusTools/1553-API version earlier than 6.42.

BusTools/1553-API normally creates buffer "0" having all elements initialized to zero, indicating no error injection, and uses this buffer for all BC and RT functions that support error injection.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using the BusTools_BM_Init.

OS Support

Core API Function (F/W Version 4.66 or earlier)

Syntax

wStatus = BusTools_EI_EbufWrite (cardnum, errorid, ebuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
errorid	(BT_UINT) target board Error Injection buffer number (0-based).
ebuf	(API_EIBUF*) a host application defined Error Injection Definition structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_EI_NOERRORS

API_EI_ILLERRORNO
API_EI_BADERRTYPE
API_EI_BADMSGTYPE

EI Buffer Example

The following example code defines an error injection buffer (#1). This error injection buffer injects a parity error into the command word of an RT Transmit message transmitted by the Bus Controller.

```
API_EBUF error_buffer;  // Allocate the temp structure

// Setup the command word error injection parameters
error_buffer.buftype = EI_BC_TRANS;
error_buffer.error[0].etype = EI_PARITY;
error_buffer.error[0].edata = 0;

// Clear the error injection parameters
// for the remaining message words.
for ( i = 1; i < EI_COUNT; i++ )
{
   error_buffer.error[i].etype = EI_NONE;
   error_buffer.error[i].edata = 0;
}

// Write the buffer to the hardware.
wStatus = BusTools_EI_EbufWrite(0, 1, &error_buffer);
if ( wStatus )
{
   printf("Error detected, %s\n",
           BusTools_StatusGetString(wStatus);
}
```

When creating the BC message that should inject this error, set the “errorid” parameter of the “API_BC_MBUF” structure to “1” to reference this error injection buffer.

4.85 BusTools_EI_EnhEbufWrite (DEPRECATED)

Description

This function should not be used. Use BusTools_EI_EbufWriteENH for all new applications.

This is a Legacy function for all boards running firmware version 5.00 or greater. This function writes error injection data to the Error Injection buffer. This function is the same as BusTools_EI_EbufWrite with the addition of new Bi-phase errors and enhanced zero-crossing errors. These new error injection features are compatible with boards running firmware version 5.0 or greater. You must use this function generate the new Bi-phase and zero-crossing errors. Use BusTools_EI_EbufWrite for board running firmware less than 5.00.

BusTools normally creates buffer "0" as all zero, indicating no error injection, and uses this buffer for all BC and RT functions that take a pointer to an error injection buffer.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using the BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_EI_EnhEbufWrite(cardnum, errorid, enhData, ebuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
errorid	(BT_UINT) Error Injection buffer number (0-based).
enhData	(BT_UINT) data for enhanced zero-crossing. See the MIL-STD-1553 Universal Core Architecture Manual Chapter 7, "Enhanced Zero-Crossing Error Injection Word" section for a description of how to set this value.
ebuf	(API_EIBUF*) pointer to Error Injection Definition (API_EIBUF) structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_EI_NOERRORS
API_EI_ILLERRORNO
API_EI_BADERRTYPE
API_EI_BADMSGTYPE

4.86 BusTools_EI_EbufWriteENH

Description

BusTools_EI_EbufWriteENH should be used for programming error injection on all boards running firmware version 5.00 or greater. See "Error Injection" section in the *BusTools/1553-API Software User's Manual* for a complete description of the Error Injection buffer and its parameters.

BusTools normally creates error injection buffer "0" having all elements initialized to zero, indicating no error injection, and uses this buffer for all BC and RT functions that support error injection.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using the BusTools_BM_Init.

OS Support

Core API Function (F/W Version 5.0 and greater)

Syntax

wStatus = BusTools_EI_EbufWriteENH (cardnum, errorid, ebuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
errorid	(BT_UINT) Error Injection buffer number (0-based).
ebuf	(API_ENH_EIBUF *) a host application defined Error Injection Definition structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_EI_NOERRORS
API_EI_ILLERRORNO
API_EI_BADERRTYPE
API_EI_BADMSGTYPE

EI Buffer Example

The following example code defines an error injection buffer (#1). This error injection buffer injects a parity error into the command word of an RT Transmit message transmitted by the Bus Controller.

```
API_ENH_EIBUF error_buffer; // Allocate a structure

// Setup the command word error injection parameters
error_buffer.buftype = EI_BC_TRANS;
error_buffer.error[0].etype = EI_PARITY;
error_buffer.error[0].edata = 0;

// Clear the error injection parameters
// for the remaining message words.
for ( i = 1; i < EI_COUNT; i++ )
{
   error_buffer.error[i].etype = EI_NONE;
   error_buffer.error[i].edata = 0;
}

// Write the buffer to the hardware.
wStatus = BusTools_EI_EbufWrite(0, 1, &error_buffer);
if ( wStatus )
{
   printf("Error detected, %s\n",
            BusTools_StatusGetString(wStatus);
}
```

When creating the BC message that should inject this error, set the "errorid" parameter of the "API_BC_MBUF" structure to "1" to reference this error injection buffer.

4.87 BusTools_EI_Getaddr

Description

BusTools_EI_Getaddr retrieves the byte offset of an Error Injection buffer from the beginning of memory on an Abaco Systems 1553 board. The Error Injection buffer is referenced by its index value, while the returned offset value will be within the range from 0 to the highest board address within the first segment.

Typically, this function is used only for debugging operations. Normally, you don't need to access absolute memory addresses on the Abaco Systems 1553 board.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_EI_Getaddr (cardnum, errorid, addr);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
errorid	(BT_UINT) Error Injection buffer number (0-based).
addr	(BT_U32BIT *) location to store the offset of the specified Error Injection buffer

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_EI_ILLERRORNO

4.88 BusTools_EI_Getid

Description

BusTools_EI_Getid converts an Abaco Systems 1553 board offset address to the respective Error Injection buffer number. The specified address is a byte offset from the beginning of memory on the board. The Address must range from 0x00000000 to 0x0003FFFF.

BusTools/1553-API uses this function internally to convert the memory address stored in the hardware to the EI buffer number returned in various message structures. Typically, this API function should not be used by application software.

Prior to invoking this function, initialize the channel using one of the BusTools/1553-API Initialization functions and initialize the Bus Monitor using BusTools_BM_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_EI_Getid (cardnum, addr, errorid);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
addr	(BT_U32BIT) memory address (0x00000000 - 0x0003FFFF).
errorid	(BT_UINT *) pointer to returned Error Injection buffer number (“0” based).

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BM_NOTINITED
API_EI_ILLERRORADDR

4.89 BusTools_ErrorCountClear

Description

BusTools/1553-API keeps a running count of messages detected while the Bus Monitor is recording. The API also keeps a cumulative count of each type of error detected during BM recording. The API resets the counters each time the Bus Monitor by invoking BusTools_BM_Init. This function allows the application to reset the API cumulative count of each type of error detected while the Bus Monitor is actively recording.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions

OS Support

Core API Function

Syntax

wStatus = BusTools_ErrorCountClear (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.90 BusTools_ErrorCountGet

Description

BusTools/1553-API maintains a running count of messages detected and a cumulative count of each type of error detected while the Bus Monitor is recording. The API resets the counters each time you re-initialize the Bus Monitor by calling BusTools_BM_Init. This function retrieves the current value of all of the counters. Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

The counters are all 32-bit values. The buffer supplied by the caller must allocate memory for at least 32 32-bit entries. The cumulative error count buffer entries are defined as follows:

- buffer[0]: number of BT1553_INT_HIGH_WORD errors.
- buffer[1]: number of BT1553_INT_INVALID_WORD errors.
- buffer[2]: number of BT1553_INT_LOW_WORD errors.
- buffer[3]: number of BT1553_INT_INVERTED_SYNC errors.
- buffer[4]: number of BT1553_INT_MID_BIT errors.
- buffer[5]: number of BT1553_INT_TWO_BUS errors.
- buffer[6]: number of BT1553_INT_PARITY errors.
- buffer[7]: number of BT1553_INT_NON_CONT_DATA errors.
- buffer[8]: number of BT1553_INT_EARLY_RESP errors.
- buffer[9]: number of BT1553_INT_LATE_RESP errors.
- buffer[10]: number of BT1553_INT_BAD_RTADDR errors.
- buffer[11]: number of BT1553_INT_CHANNEL errors.
- buffer[12]: number of BT1553_INT_UNTERM_BUS errors.
- buffer[13]: number of BT1553_INT_WRONG_BUS errors.
- buffer[14]: number of BT1553_INT_BIT_COUNT errors.
- buffer[15]: number of BT1553_INT_NO_IMSG_GAP errors.
- buffer[16]: number of BT1553_INT_END_OF_MESS conditions
- buffer[17]: number of BT1553_INT_BROADCAST messages.
- buffer[18]: number of BT1553_INT_RT_RT_FORMAT messages.
- buffer[19]: number of BT1553_INT_RESET_RT messages.
- buffer[20]: number of BT1553_INT_SELF_TEST messages.
- buffer[21]: number of BT1553_INT_MODE_CODE messages.
- buffer[22]: number of BT1553_INT_INVALID_CMDW errors.
- buffer[23]: number of Service Requests
- buffer[24]: number of Broadcast Received

- buffer[25]: number of busy responses
- buffer[26]: not used
- buffer[27]: number of Message Errors
- buffer[28]: number of Subsystem Flags
- buffer[29]: number of Terminal Flags
- buffer[30]: not used
- buffer[31]: message count

OS Support

Core API Function

Syntax

wStatus = BusTools_ErrorCountGet (cardnum, count, buf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
count	(BT_U32BIT *) pointer to a variable to receive the total number of messages processed.
buf	(BT_U32BIT *) pointer to a 32-element buffer to receive the 32 error counters.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.91 BusTools_ExtTrigIntEnable

Description

BusTools_ExtTrigIntEnable enables or disables hardware interrupts on an external trigger input. When an interrupt is encountered from the external trigger, the user callback function is invoked as defined by the host application. There is no message or time data associated with external trigger interrupt.

See the documentation for the function BusTools_RegisterFunction for more details on assigning a user callback function to an external trigger input.

Some Abaco Systems MIL-STD-1553 boards require configuration of a discrete input to be utilized as an external trigger. The application must assure the board is configured for an external trigger input and correctly connect the external trigger input to the associated discrete input.

Prior to calling this function, the channel must be initialized by calling one of the BusTools initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_ExtTrigIntEnable (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_U16BIT) Enable/disable flag: Options are EXT_TRIG_ENABLE EXT_TRIG_DISABLE

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.92 BusTools_ExtTriggerOut

Description

BusTools_ExtTriggerOut activates the external trigger output for a specified duration. The duration of the output pulse is measured using the Time Tag timer, so the width is accurate to within a few microseconds. For precise width measurements you may want to calibrate the pulse width with an oscilloscope.

The pulse width range is 1 to 1000 µs.

If your board supports an external trigger on programmable discretes (see Table 1-2), the application must set up the external trigger on either discrete 7 or 8 by invoking BusTools_DiscreteSetIO and BusTools_DiscreteTriggerOut prior to invoking this function.

OS Support

Core API Function

Syntax

wStatus = BusTools_ExtTriggerOut (cardnum, pwidth);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
pwidth	(BT_U16BIT) Pulse width in microseconds. Valid range is 1 to 1000.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.93 BusTools_FindDevice

Description

BusTools_FindDevice finds and returns the device number for the specified BusToos/1553-API supported board. This function accepts the card type and the instance of the desired board type as input arguments. The function returns the device number associated with the selected board. If there is a single Abaco Systems device installed in your system, it is always device 0. If multiple Abaco Systems boards are installed in a system, including non-1553 boards, an application can invoke this function to acquire the device number of the desired type of Abaco 1553 board.

The function returns either the device number if the device is found or -1 if the device is not found. This function may be invoked at any time.

OS Support

Windows only

Syntax

wDevice = BusTools_FindDevice (device_type, instance);

wDevice	(BT_INT) the device number (0 – 63) if the device was discovered, or -1 if the device was not found.
device_type	(BT_UINT) PCI1553, QPCI553, QPCX1553, PMC1553, QPMC1553, QPM1553, AMC1553, QCP1553, Q1041553, Q1041553P, ISA1553, R15EC, R15AMC, RXMC1553, R15XMC2, RPCIe1553, LPCIE1553, MPCIE1553
instance	(BT_INT) The instance of the specified device type in the system. This a 1-based number representing instance of the specified *device_type* installed in the system, when multiple boards of that type are installed. See note below.

Return Value

0 – 63 : the device number of the specified device type
-1 : No device matching the specified device type was found

Notes

If you have two PCI-1553 boards installed in your system, use the following calls to return the device number for each board:

```
device1 = BusTools_FindDevice(PCI1553,1);

device2 = BusTools_FindDevice(PCI1553,2);
```

You can embed this call into BusTools_API_OpenChannel to initialize a specific channel/device referenced by this function. See the code below:

```
status = BusTools_API_OpenChannel(&cardnum, mode,
         BusTools_FindDevice(PCI1553,2), CHANNEL_1);
```

This line of code initializes the first channel on the second PCI-1553 device installed in the system and returns a "handle" that must be used with all BusTools/1553-API calls having a *cardnum* parameter.

4.94 BusTools_FirmwareReload

Description

BusTools_FirmwareReload forces an FPGA program reload from flash memory, supported only with the RAR15-XMC-IT/RAR15XF boards.

A board's firmware is normally loaded during board power-up. This function allows the host application to programmatically force a firmware program reload. After this function is invoked, a reload command is sent to the board, causing the FPGA to reload the firmware from flash memory. As a result of reloading firmware, the board will be uninstalled by the operating system. A restart or reboot is required to re-install the board's device driver. Currently, this function only supports the firmware reload for the RAR15-XMC-IT/RAR15XF boards with firmware build number greater than 0x200. This function will return an error if invoked for any other board type or firmware revision.

OS Support

Core API Function

Syntax

wStatus = BusTools_FirmwareReload (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT
API_BUSTOOLS_NOTINITED

4.95 BusTools_FlashLogErase

Description

BusTools_FlashLogErase is one of the methods provided for user applications to manage application storage and retrieval of data from the Abaco Systems RAR15 combo-card product line flash memory. This function provides the method to erase a specific sector of user-accessible flash memory. Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

Each sector is 64 KBytes in size. This function accepts the *cardnum* channel reference and a sector number. Currently, only four sectors (1-4) can be selected with this function. It is recommended that an application record the number of BusTools_FlashLogErase function calls used for each sector, as the flash component is limited to 100,000 erases per sector program cycle. After 100,000 erase cycles are reached, the respective sector on the flash component may or may not function reliably. **Note** this function is only valid for RAR15-XMC-IT and RAR15XF boards. This function will return an error if it invoked for any other board type.

To ensure the integrity of flash operations, execution of the BusTools_FlashLogErase function must be protected by the critical section in concurrent application programming environments.

OS Support

Core API Function

Syntax

wStatus = BusTools_FlashLogErase (cardnum, sector);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
sector	(BT_INT) flash sector to erase. Valid range is 1 to 4.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT
API_BUSTOOLS_NOTINITED
API_BAD_PARAM

4.96 BusTools_FlashLogRead

Description

BusTools_FlashLogRead is one of the methods provided for user applications to manage application storage and retrieval of data from the Abaco Systems RAR15 combo-card product line flash memory. This function provides the method to read a specified number of data bytes from an offset in the designated flash sector. Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

Each flash sector is 64 KBytes in size. The BusTools_FlashLogRead function accepts *cardnum* channel reference, a sector number (1-4), a byte offset into the sector, and number of bytes to read. It returns the data from this read operation. **Note** that this function is only valid for RAR15-XMC-IT and RAR15XF boards. This function will return an error if invoked for any other board type.

To ensure the integrity of flash operations, execution of the BusTools_FlashLogRead function must be protected by the critical section in concurrent application programming environments.

OS Support

Core API Function

Syntax

wStatus = BusTools_FlashLogRead (cardnum, sector, paddr, bcount, rData);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
sector	(BT_INT) flash sector to read. Valid range is 1 to 4.
paddr	(BT_UINT) byte offset to begin reading within the sector. Valid range is 0 to 65534.
bcount	(BT_UINT) number of bytes to read. Valid range is 1 to 65535.
rData	(BT_U8BIT*) reference to the data buffer to be written by this function, consisting of a minimum memory allocation of *bcount* bytes.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT
API_BUSTOOLS_NOTINITED
API_BAD_PARAM

4.97 BusTools_FlashLogWrite

Description

BusTools_FlashLogWrite is one of the methods provided to user applications to manage application storage and retrieval of data from the Abaco Systems RAR15 combo-card product line flash memory. This function provides the method to write a single page of flash memory (up to 256 bytes) into a designated flash sector using the data values provided. The write action always occurs at the base location of the designated page. Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

Each flash sector is 64 KBytes in size and contains 256 pages each consisting of 256 bytes. The BusTools_FlashLogWrite function accepts *cardnum*, flash sector (1-4), page number (0-255) and data values to be written. The designated flash sector must be erased first before any write operations can be performed. **Note** that this function can only be used for board types RAR15-XMC-IT and RAR15XF. This function will return an error if invoked for any other board type.

To ensure the integrity of flash operations, execution of the BusTools_FlashLogWrite function must be protected by the critical section in concurrent application programming environments.

OS Support

Core API Function

Syntax

wStatus = BusTools_FlashLogWrite (cardnum, sector, pagenum, pageData);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
sector	(BT_INT) flash sector to write. Valid range is 1 to 4.
pagenum	(BT_INT) page number within *sector* to write. Valid range is 0-255.
bcount	(BT_UINT) number of bytes to write to the location(s) specified by *sector/pagenum*. Valid range is 0-255.
pageData	(BT_U8BIT*) application data buffer.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT
API_BUSTOOLS_NOTINITED
API_BAD_PARAM

4.98 BusTools_GetAddr

Description

BusTools_GetAddr provides the starting and ending memory addresses for a specified block of a 1553 board register segment or logical RAM memory block. The returned values are byte offsets from the beginning of the board host interface. The offsets range from zero to the highest board offset. Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

The available memory block types available include:

- Hardware registers
- Microcode registers
- BC Message Buffers
- BM Filter Buffer
- BM Trigger Buffer
- BM Message Buffers
- BM Control Buffers
- BM Default Control Buffers
- RT Address Control Blocks
- RT Filter Buffers
- RT Message Buffers
- Error Injection Buffers
- Interrupt Queue
- RT Default Message Buffers
- RT Default Control Buffers
- RT Broadcast Control Buffers
- RT MBUF's
- Differential I/O registers

OS Support

Core API Function

Syntax

wStatus = BusTools_GetAddr (cardnum, memtype, start, end);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

memtype (BT_UINT) memory/register segment to reference:

- GETADDR_HWREG: Hardware registers (1).
- GETADDR_RAMREG: Microcode registers (2)
- GETADDR_BCMESS: BC Message Buffers (3).
- GETADDR_BMFILTER: BM Filter Buffer (4).
- GETADDR_BMTRIGGER: BM Trigger Buffer (5).
- GETADDR_BMMESSAGE: BM Message Buffers (6).
- GETADDR_BMCONTROL: BM Control Buffers (7).
- GETADDR_BMDEFCBUF: BM Default Control Buffers (8).
- GETADDR_RTADDRESS: RT Address Control Blocks (9).
- GETADDR_RTFILTER: RT Filter Buffers (10).
- GETADDR_RTDATA: RT Message Buffers (11).
- GETADDR_EI: Error Injection Buffers (12).
- GETADDR_IQ: Interrupt Queue (13).
- GETADDR_RTMBUF_DEF: RT Default Message Buffers (14).
- GETADDR_RTCBUF_DEF: RT Default Control Buffers (15).
- GETADDR_RTCBUF_BRO: RT Broadcast Control Buffers (16)
- GETADDR_PCI_RTDATA: RT MBUF's (17)
- GETADDR_DIFF_IO (18)

start (BT_U32BIT*) reference to store the beginning offset of the specified register segment or memory block.

end (BT_U32BIT*) reference to store the ending offset of the specified register segment or memory block.

Return Value

API_SUCCESS
API_BAD_ADDR_TYPE
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

Notes

The symbol "GETADDR_COUNT" defines the number of available memory block types.

If programming a loop to get and display the offsets, begin the loop with the value "1", and end the loop with the value "GETADDR_COUNT".

4.99 BusTools_GetBoardType

Description

BusTools_GetBoardType returns the board type hosting the specified channel reference/session. Board types available include:

Mnemonic	Value	Description
QPMC1553	0x110	QPMC-1553 native PMC board
QPM1553	0x110	QPM-1553 (QPMC1553 variant)
QPCX1553	0x220	QPCX-1553 Native PCI board.
Q1041553P	0x180	PC\104 4-Ch Plus board
QVME1553	0x190	Quad Channel VME-1553
PCCD1553	0x200	Dual Channel PCCard-D1553
R15EC	0x230	Dual Channel Express Card (RoHS)
RXMC1553	0x260	Dual Channel XMC (RoHS)
QCP1553	0x210	Quad-Channel cPCI board
RPCIe1553	0x250	RPCIe-1553 (RoHS PCI-E)
R15XMC2	0x300	RXMC2-1553 (RoHS XMC)
R15-LPCIE	0x320	R15-LPCIE (low profile PCI-E)
MPCIE-1553	0x400	MPCIe-1553 (Mini PCI-E)
R15-USB	0x3000	R15-USB USB interface board
RAR15X(F)	0x360	Any Multi-Protocol XMC board

This function can be invoked after initializing a channel with a call to BusTools_API_OpenChannel or BusTools_API_InitExtended.

OS Support

Core API Function.

Syntax

wType = BusTools_GetBoardType (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

Board Type value (see Above)
API_BUSTOOLS_NOTINITED
API_BUSTOOLS_BADCARDNUM

4.100 BusTools_GetChannelStatus

Description

BusTools_GetChannelStatus returns a structure containing a summary of information about the channel specified by the card number parameter. The application passes a pointer to an API_CHANNEL_STATUS structure and BusTools_GetChannelStatus fills in the data.

Call this function periodically to assess the condition of the channel. See the description of the API_CHANNEL_STATUS structure to see the status information this function returns.

Prior to calling this function, the 1553 channel must be initialized by calling one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_GetChannelStatus (cardnum, cstat);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
cstat	(API_CHANNEL_STATUS *) pointer to a channel status structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.100.1 API_CHANNEL_STATUS Definition

Note the following definitions are not presented in structure bit order.

Error Status Bits

mf_ovfl	Minor frame overflow: 1 = overflow, 0 = no overflow
wcs_pulse	WCS heartbeat counter error: 1 = error, 0 = no error
interr	Interrupt error: 1 = error, 0 = no error
addr_err	Address error: 1 = error, 0 = no error
byte_cnt_err	Byte count error on read or write: 0 = no error, 1 = error

err_info	3-Bit error value for run time errors 0 = No Error 1 = CHAN_STAT_MF_OVFL 2 = CHAN_MEM_TST_FAIL 4 = CHAN_STAT_INT_ERR

Run Status Bits

bc_run	BC: 1 = BC is running, 0 = BC is not running
rt_run	RT: 1 = RT is running, 0 = RT is not running
bm_run	BM: 1 = BM is running, 0 = BM is not running

Operational Status

int_mode	Interrupt mode: 1 = H/W interrupts, 0 = S/W polling
run_mode	1553 mode: 1 = 1553A, 0 = 1553B
extbus	External bus: 1 = External, 0 = Internal
coupling	Bus Coupling: 1 = Transformer, 0 = Direct
SA_31	SA31 used as Mode Code: 1 = yes, 0 = no
broadcast:	RT31 used as Broadcast: 1 = yes, 0 = no
irig_on	IRIG_B time source: 1 = internal, 0 = external
int_fifo_count	Number of interrupt threads running

4.101 BusTools_GetChannelCount

Description

BusTools_GetChannelCount reads the 1553 channel configuration from the board on which the supplied channel reference resides. This function returns the channel count value of 1, 2, or 4; or in the case of an error it returns a status code value in excess of 200.

Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

OS Support

Core API Function.

Syntax

count = BusTools_GetChannelCount (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

Channel Count (1, 2, or 4)
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.102 BusTools_GetCSCRegs

Description

BusTools_GetCSCRegs returns the contents of the Control Status Configuration (CSC) registers located in the first two words on most Abaco Systems 1553 boards. These registers are not supported with native VME 1553 boards. Use BusTools_ReadVMEConfig to acquire configuration information for native VME 1553 boards. Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

There is a single CSC (and possibly AC register) for each Abaco Systems 1553 board. There are two variations of the CSC register depending on Abaco Systems 1553 board type.

For actively supported boards programmed with UCA32 firmware, the CSC register has the following format.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
acr	undefined		irig	mod	channel_num					board_type					rst0

rst0

This is to be used for board initialization only and the bit cannot be read back.

board_type:

1	QPMC-1553/QPM-1553
3	QPCI-1553/QPCX-1553
5	Q104-1553P (PCI)
7	QCP-1553
8	AMC-1553
9	R15-EC
11	RXMC-1553
12	RPCIe-1553
14	RXMC2-1553
15	R15-LPCIE
16	R15-USB
18	RAR15X(F) (RAR15-XMC-FIO, RAR15-XMC-IT)
19	R15-PMC
20	R15-MPCIE

channel_num

These five read-only bits determine the number of 1553 channels on the board. The following are possible values:

1	One 1553 channel
2	Two 1553 channels
4	Four 1553 channels

mode

This read-only bit determines if the board is single-mode/dual-mode (0) or multi-mode (1).

irig

This read-only bit determines if IRIG is enabled (1) or not (0).

acr

ACR present. This read-only bit indicates if the Advance Capabilities Register (ACR) is present (1) or not (0).

For legacy boards the CSC register has the following format.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bt[1..0]		pr2	pr1	f2	f1	m2	m1	sb2	sb1	su2	su1	rsvd	rst2	rst1	rst0

Byte 0: Reset/Power Control

rst0	7K reset (write only)
rst1	reset bus 1 LPU (write only)
rst2	reset bus 2 LPU (write only)
rsvd	reserved
su1	CH 1 WCS set-up complete
su2	CH 2 WCS set-up complete
sb1	CH 1 in standby power mode
sb2	CH 2 in standby power mode

Byte 1: Board ID bits

m1	CH 1 multiple mode (read only)
m2	CH 2 multiple mode (read only)
f1	CH 1 1553/1773 (read only)
f2	CH 2 1553/1773 (read only)
pr1	CH 1 present (read only)
pr2	CH 2 present (read only)
bt[1..0]	host bus type ID (read only)

OS Support

Core API Function

Syntax

wStatus = BusTools_GetCSCRegs (cardnum, csc, acr);

wStatus (BT_INT) status returned from this function.

cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
csc	(BT_U16BIT *) location to receive the CSC register information.
acr	(BT_U16BIT *) location to receive the AC register information.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT

4.103 BusTools_GetDevInfo

Description

BusTools_GetDevInfo returns information about an Abaco Systems 1553 device installed in Windows O/S based host.

For PCI, PCI Express, ExpressCard, and PCMCIA devices, the wDevice parameter is the board installation device ID (0 – 63).

For VME devices use the A16 address (default 0xC3C0 for VME\VXI) in place of a board installation ID. If you are checking VME devices, you must have the National Instruments VXI/VISA libraries installed and use the BTVXIMAP.DLL interface library.

Invocation of this function does not require board initialization.

OS Support

Core API function

Syntax

wStatus = BusTools_GetDevInfo (device, pInfo);

device	(BT_INT) BusTools/1553-API card number. Valid range is 0 to 63.
pInfo	(DEVICE_INFO *) location to store the device information structure content; see Device Information (DEVICE_INFO).

Return Value

API_SUCCESS
API_BAD_PRODUCT_LIST
API_INSTALL_ERROR
BTD_ERR_BADADDRMAP
BTD_ERR_LOAD_CEIINST
BTD_ERR_NOACCESS
BTD_ERR_PARAM
BTD_LL_CLOSE_ERR
BTD_LL_OPEN_ERR

4.104 BusTools_GetFWRevision

Description

BusTools_GetFWRevision returns the firmware revision of the board on which the channel referenced is installed. There are two parts to the firmware: Local Processing Unit (LPU) and Writeable Control Store (WCS). This function returns both the LPU and WCS revisions. Additionally, this function returns the Build number. For firmware V5.x and earlier the build number is the LPU build number; for firmware V6.x, this is the FPGA build number.

Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_GetFWRevision (cardnum, wrev, lrev, build);

cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
wrev	(float *) location to store the WCS revision number (floating point representation, e.g., 3.88).
lrev	(float *) location to store the LPU revision number (floating point representation, e.g., 6.15).
build	(BT_INT *) F/W Build number.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.105 BusTools_GetPulse

Description

BusTools_GetPulse returns the contents of the WCS Heartbeat Register. Because the LPU firmware continually increments the value in this register, changes in the value with successive reads indicate whether the microcode is executing. If the register contents stop changing, the WCS is no longer executing.

Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_GetPulse (cardnum, beat);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
beat	(BT_U32BIT *) location to which the current Heartbeat register value is written.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.106 BusTools_GetRevision

Description

This function returns the revision of the API and the microcode the board on which the supplied channel reference resides. Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions. When creating a string to display the revision codes, make sure to display the ucode_rev in hexadecimal format, and the api_rev in decimal format.

OS Support

Core API Function

Syntax

wStatus = BusTools_GetRevision (cardnum, ucode_rev, api_rev);

cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
ucode_rev	(BT_UINT*) location to receive the firmware revision number (hexadecimal).
api_rev	(BT_UINT*) location to receive the API revision number (decimal).

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

Notes

The *ucode_rev* firmware revision number is filled in as follows:

- Firmware version is computed as follows:
  - LPU Revision (3 digits) +
  - Single Function Flag * 1000 +
  - PROM Revision Digit * 10000.

For example, a *ucode_rev* value of 31321 shows:

  - PROM Revision – 3
  - Single Function – Yes (1 indicates a single/dual-function board)
  - Firmware (WCS and FPGA) – 3.21

- While a *ucode_rev* value of 40380 shows:
  - PROM Revision – 4
  - Single Function – No

- Firmware(fpga) – 3.80

NOTE

Only a few older products still use the PROM revision. For current boards it is set to zero. The newer function BusTools_GetFWRevision is recommended.

4.107 BusTools_GetSerialNumber

Description

Each Abaco Systems 1553 board has either a four- or five-digit serial number. Some 1553 boards have the serial number stored in flash. BusTools_GetSerialNumber provides a method to acquire the board's serial number.

Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_GetSerialNumber (cardnum, serial_number);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
serial_number	(BT_U32BIT *) location to receive the serial number.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.108 BusTools_GetTermEnable

Description

The RXMC2-1553 and R15-LPCIE have two differential discrete I/O lines supporting switchable termination. BusTools_GetTermEnable provides the method to read the setting for the 120-Ω termination on these discrete channels. See the respective 1553 board-specific chapter in the MIL-STD-1553 Hardware Installation and Reference Manual for information about board features supported and the section titled RS485 Transmit & Control within the Global Register Description chapter of the UCA32 Global Register Reference manual for descriptions of the use of RS485 the termination.

OS Support

Core API Function

Syntax

wStatus = BusTools_GetTermEnable (cardnum, tEnable);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
tEnable	(BT_U16BIT *) location to receive the current termination setting. Valid range is 0 to 3.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.109 BusTools_GetTimeTagMode

Description

BusTools_GetTimeTagMode returns the current application defined time and time-stamp features used within BusTools/1553-API, assigned via invocation of BusTools_TimeTagMode.

- Time Display Format, valid values are:

  API_TT_DEFAULT

  API_TTD_RELM

  API_TTD_IRIG

  API_TTD_DATE

  API_TTD_RELM_NS

  API_TTD_IRIG_NS

  API_TTD_DATE_NS

- Time-Tag Initialization Selection, valid values are:

  API_TTI_ZERO

  API_TTI_DAY

  API_TTI_IRIG

  API_TTI_EXT

  API_TTI_IRIG64

  API_TTI_DAY64

- Operating Mode options, valid values are:

  API_TTM_FREE

  API_TTM_RESET

  API_TTM_SYNC

  API_TTM_RELOD

  API_TTM_IRIG

  API_TTM_AUTO

  API_TTM_XCLK

- External Time-tag Synchronization Period. For Version 4/5 firmware, this value is defined as the current value of the Time Tag Counter Increment Register, having a resolution of 1µs and a range from 0 to 65535. For Version 6 firmware this value is defined as the current value of the External Increment Count field located in the UCA32 Time Tag Counter Control Register, having a resolution of 1ns and a range from 0 to 1023.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_GetTimeTagMode (cardnum, pTTDisplay, pTTinit, pTTMode, pTTPeriod);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
pTTdisplay	(BT_INT *) the current time display format.
pTTInit	(BT_INT *) the current time tag initialization selection.
pTTMode	(BT_INT *) the current time tag operation mode.
pTTPeriod	(BT_U32BIT *) the current timer increment period selection for external time sync mode.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.110 BusTools_GetValidDiscrete

Description

BusTools_GetValidDiscrete returns a 32-bit bitwise encoded value defining the discrete channels available on the board.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function (F/W version 6.01 or greater).

Syntax

wStatus = BusTools_GetValidDiscrete (cardnum, disSetting);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
disSetting	(BT_U32BIT *) location to write the value reflecting the available discrete channels.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.111 BusTools_GetValidPio

Description

BusTools_GetValidPio returns a 32-bit bitwise encoded value defining the PIO channels that are available on the board.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function (F/W version 6.01 or greater).

Syntax

wStatus = BusTools_GetValidPio (cardnum, pioSetting);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
pioSetting	(BT_U32BIT *) location to write the value reflecting the available PIO channels.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.112 BusTools_GetValidDiff

Description

These functions return a 32-bit bitwise encoded value defining the differential channels that are available on the board.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function (F/W version 6.01 or greater).

Syntax

wStatus = BusTools_GetValidDiff (cardnum, diffSetting);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
diffSetting	(BT_U32BIT *) location to write the value reflecting the available differential channels.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.113 BusTools_InterMessageGap

Description

BusTools_InterMessageGap computes the inter-message time gap between two successive MIL-STD-1553B messages recorded by the Bus Monitor. It first calculates the difference between the two time tags and then subtracts the time the first message used on the bus.

Messages are time-stamped at the mid sync of the command word, and the intermessage gap time is defined as the time between the mid parity of the last word of the previous message to the mid sync of the command word of the current message. The mid parity occurs 0.5 us before the end of the word, and the mid sync occurs 1.5 us after the command word starts, so the intermessage gap time is 2 us LESS than the bus dead time.... Response times are measured like gap times; 2 us greater than the bus dead time.

Only use BusTools_InterMessageGap for host configurations where boards with V4/5 firmware are installed.

OS Support

Core API Function

Syntax

dStatus = BusTools_InterMessageGap (first, second);

dStatus	(BT_U32BIT) Inter-message gap time in microseconds.
first	(API_BM_MBUF *) pointer to first message.
second	(API_BM_MBUF *) pointer to second message.

Notes

This function returns a calculated value since the 1553 firmware does not directly measure the inter message gap time. If certain undetected bus errors occur, the gap time returned might be inaccurate. For normal messages (and normal errors such as no response, message error, etc.), the value returned is accurate. The gap time is not accurate for MIL-STD-1553A mode codes.

4.114 BusTools_InterMessageGap2

Description

BusTools_InterMessageGap2 computes the inter-message time gap between two successive MIL-STD-1553B messages recorded by the Bus Monitor. It first calculates the difference between the two time tags and then subtracts the time the first message used on the bus.

Messages are time-stamped at the mid sync of the command word, and the intermessage gap time is defined as the time between the mid parity of the last word of the previous message to the mid sync of the command word of the current message. The mid parity occurs 0.5 us before the end of the word, and the mid sync occurs 1.5 us after the command word starts, so the intermessage gap time is 2 us LESS than the bus dead time.... Response times are measured like gap times; 2 us greater than the bus dead time.

Use the function BusTools_InterMessageGap2 when using a board with V6 firmware or a mix of boards having V4/5 and V6 firmware.

OS Support

Core API Function

Syntax

dStatus = BusTools_InterMessageGap2 (flag, first, second);

dStatus	(BT_U32BIT) Inter-message gap time in microseconds.
flag	(BT_UINT) Time-tag resolution flag; use 0 with any V4/5 firmware-based board (resolution is in μs), or 1 with any V6 firmware-based board (resolution is in ns).
first	(API_BM_MBUF *) pointer to first message.
second	(API_BM_MBUF *) pointer to second message.

Notes

This function returns a calculated value since the 1553 firmware does not directly measure the inter message gap time. If certain undetected bus errors occur, the gap time returned might be inaccurate. For normal messages (and normal errors such as no response, message error, etc.), the value returned is accurate. The gap time is not accurate for MIL-STD-1553A mode codes.

4.115 BusTools_IRIG_Calibration

Description

BusTools_IRIG_Calibration calibrates the external IRIG signal threshold on boards supporting the IRIG time function (see Table 1-2), setting the input DAC for optimal reception of IRIG AM signals. Calibration adjusts 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 external IRIG signal threshold is set to 3 volts by default, typically a valid threshold for an IRIG DC source.

IRIG-B is common to all channels on a board. Prior to invoking this function, initialize a channel using one of the BusTools/1553-API Initialization functions. Subsequently invoke BusTools_TimeTagMode to enable IRIG source timing before calibration by selecting the API_TTM_IRIG mode.

OS Support

Core API Function

Syntax

wStatus = BusTools_IRIG_Calibration (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_INT) print flag. When set to 0, calibration information is unreported; when set to 1, this function will print the calibration information to the console.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT
BTD_IRIG_NO_LOW_PEAK
BTD_IRIG_NO_HIGH_PEAK
BTD_IRIG_LEVEL_ERR

4.116 BusTools_IRIG_Config

Description

BusTools_IRIG_Config configures the IRIG function on boards supporting the IRIG time function (see Table 1-2). This function allows an application to select an internal or external IRIG time source and control the output of the IRIG generator signal.

IRIG-B is common to all channels on a board. Prior to invoking this function, initialize a channel using one of the BusTools/1553-API Initialization functions. Subsequently invoke BusTools_TimeTagMode to enable IRIG source timing before calibration by selecting the API_TTM_IRIG mode.

If an application is required to use the onboard IRIG time generator, initialize the current IRIG time via invocation of BusTools_IRIG_SetTime. If an application requires use of an external IRIG signal source, it is recommended to calibrate the IRIG receiver to that signal using BusTools_IRIG_Calibration. Invoke BusTools_IRIG_Valid to determine if the IRIG receiver is detecting a valid IRIG signal.

OS Support

Core API Function

Syntax

wStatus = BusTools_IRIG_Config (cardnum, intFlag, outFlag);

wStatus (BT_INT) status returned from this function.

cardnum (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

intFlag (BT_UINT) IRIG source flag. Options are:

IRIG_INTERNAL to internally wrap the onboard IRIG generator as the IRIG source.

IRIG_EXTERNAL select the IRIG receiver I/O pins for the IRIG source.

outFlag (BT_UINT) External IRIG generator output flag. Options are:

IRIG_OUT_ENABLE to enable external output of the IRIG generator signal.

IRIG_OUT_DISABLE to disable external IRIG.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM

API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.117 BusTools_IRIG_SetTime

Description

BusTools_IRIG_SetTime sets the IRIG encoded time output by the onboard IRIG generator. If the IRIG generator is not set using this function, time will start incrementing from 0 at board power-up.

IRIG-B is common to all channels on a board. Prior to invoking this function, initialize a channel using one of the BusTools/1553-API Initialization functions. When setting the initial IRIG time, two options for a time base are available:

1. GMTIME (0): This option converts a time in seconds since the Epoch (00:00:00 UTC, January 1, 1970) into a time expressed as Coordinated Universal Time, or UTC (e.g., the time at the GMT time zone).
2. LOCALTIME (1): This option converts a time in seconds since the Epoch (00:00:00 UTC, January 1, 1970) into a time expressed as a local system time. The function corrects for the time zone and any seasonal adjustments.

OS Support

Core API Function

Syntax

wStatus = BusTools_IRIG_SetTime (cardnum, timedate, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
timedate	(time_t) a time_t time value. Set to -1 to use system time.
flag	(BT_U32BIT) For a value of GMTIME (0), the *timedate* value is converted to UTC; for a value of LOCALTIME (1), the *timedate* value is converted to local time.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

Notes

To set the IRIG to a value other than the system time pass a *time_t* parameter containing the encoded time. The following code example shows how to set the IRIG time to noon 4 July 2005. mktime returns -1 if it cannot convert the struct tm_data into a time.

```
struct tm btime;
```

```
time_t tdate;

btime.tm_sec=1;
btime.tm_min=0;
btime.tm_hour=12;
btime.tm_mon=7-1;   // Make sure to use “Month-1”
btime.tm_mday=4;
btime.tm_year=2005-1900;
btime.tm_isdst = -1;

tdate = mktime(&btime);

if (tdate == -1) {
   printf(“time conversion error; using system time\n);
}
status = BusTools_IRIG_SetTime(cardnum, tdate, 1);
printf("BusTools_IRIG_SetTime status = %d\n", status);
```

4.118 BusTools_IRIG_Valid

Description

BusTools_IRIG_Valid determines if there is a valid external IRIG signal present. The function returns API_SUCCESS if the IRIG signal is valid and an error if it is not valid.

If the application has configured the board to use the internal IRIG clock, this invocation will always return API_SUCCESS. If an external IRIG source is used and this function returns API_IRIG_NO_SIGNAL, it is recommended to calibrate the IRIG receiver to that signal using BusTools_IRIG_Calibration.

Prior to calling this function, the channel must be initialized by calling one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_IRIG_Valid (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT
API_IRIG_NO _SIGNAL

4.119 BusTools_ListDevices

Description

This function returns a list of Abaco Systems MIL-STD-1553 Plug-n-Play devices installed in your system. This function also returns other device-related information in a "DeviceList" structure.

NOTE

The calling application must allocate storage for 16 *DeviceList* array entries.

This function may be invoked at any time, it does not require a previously opened session with a board.

OS Support

Core API Function

Syntax

wStatus = BusTools_ListDevices (list);

wStatus	(BT_INT) status returned from this function.
list	(DeviceList *) Location to store the DeviceList structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT
API_IRIG_NO _SIGNAL

4.119.1 Device List Structure Contents

Num_devices Number of MIL-STD-1553 devices found in the system.

Device_name Integer Array of device types, valid values are:

QPMC-1553	0x110
QPCI-1553	0x160
Q104-1553P	0x180
QCP-1553	0x210
QPCX-1553	0x220
R15-EC	0x230
QPM-1553	0x110
R15-AMC	0x240
RPCIE-1553	0x250
RXMC-1553	0x260
RXMC2-1553	0x300
R15-LPCIE	0x320
RAR15-XMC	0x360

	R15-PMC	0x380
	R15-MPCIE	0x400
	R15-USB	0x3000
Device_num	Integer Array of device numbers	
Name	Character Array of device names in ASCII	

4.120 BusTools_MemoryAlloc

Description

BusTools_MemoryAlloc provides a method for the application to allocate a block of memory on the Abaco Systems 1553 board when it is programmed with version 4.x or 5.x firmware. Currently, the only use for this block is for referencing one or more words by conditional BC messages.

Only invoke BusTools_MemoryAlloc after initializing the memory system via BusTools_BM_Init, and before invoking BusTools_XX_StartStop to start the execution of the board's microcode. This function will allocate space in the lower 64K for 16-bit data and ensure an even WORD boundary. The function returns an API_BUSTOOLS_NO_MEMORY error if the allocated block causes the memory allocation pointer to exceed 64K words. In this case, the function allocates no memory.

If an application requires larger amounts of memory than this function is capable of allocating, use the BC message allocation functions to allocate enough room for both the messages and the buffer. Invoke BusTools_BC_MessageGetaddr to acquire the address of the allocated buffer.

Prior to invoking this function, initialize a channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_MemoryAlloc (cardnum, segnum, bcount, addr);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
segnum	(BT_UINT) memory segment number (0 indicates SEG1).
bcount	(BT_U32BIT) number of bytes to reserve.
addr	(BT_U32BIT*) location to store the starting address of the reserved block.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BUSTOOLS_BADSEGNUM
API_BUSTOOLS_EVENBCOUNT

4.121 BusTools_MemoryAlloc32

Description

BusTools_MemoryAlloc32 provides a method for the application to allocate a block of memory on the Abaco Systems 1553 board when it is programmed with version 6.x firmware. Currently, the only use for this block is for referencing one or more words by conditional BC messages.

Only invoke BusTools_MemoryAlloc32 after initializing the memory system via BusTools_BM_Init, and before invoking BusTools_XX_StartStop to start the execution of the board's microcode. This function will allocate space in the lower 64K for 16-bit data and ensure an even WORD boundary. The function returns an API_BUSTOOLS_NO_MEMORY error if the allocated block causes the memory allocation pointer to exceed 64K words. In this case, the function allocates no memory.

If an application requires larger amounts of memory than this function is capable of allocating, use the BC message allocation functions to allocate enough room for both the messages and the buffer. Invoke BusTools_BC_MessageGetaddr to acquire the address of the allocated buffer.

Prior to invoking this function, initialize a channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_MemoryAlloc32 (cardnum, segnum, bcount, addr);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
segnum	(BT_UINT) memory segment number (0 indicates SEG1).
bcount	(BT_U32BIT) number of bytes to reserve.
addr	(BT_U32BIT*) location to store the starting address of the reserved block.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BUSTOOLS_BADSEGNUM
API_BUSTOOLS_EVENBCOUNT

4.122 BusTools_MemoryAvailable

Description

BusTools_MemoryAvailable returns the memory available for programming Bus Controller, Remote Terminal, or Bus Monitor buffers. Each channel on a 1553 interface board has 1 megabyte of memory available for programming the BC, RT, or BM functions. BusTools/1553-API allocates the Bus Controller and Bus Monitor buffers from the bottom of memory up and the Remote Terminal buffer for the top of memory down. That area between the BC and BM buffer (bottom memory) and the RT buffers (top memory) is the available memory for the channel.

Prior to invoking this function, initialize a channel using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function.

Syntax

wStatus = BusTools_MemoryAvailable (cardnum, bytes);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
bytes	(BT_U32BIT *) Pointer to variable to contain the amount of available memory, in bytes.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.123 BusTools_MemoryRead(obsoleted)

Description

BusTools_MemoryRead is obsolete and no longer supported in the API; however, it will remain supported in the API for legacy applications. BusTools_MemoryRead2 is the replacement function for BusTools_MemoryRead.

OS Support

Core API Function

Syntax

wStatus = BusTools_MemoryRead (cardnum, addr, bcount, buff);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
addr	(BT_U32BIT) 1553 board memory address/register offset to read.
bcount	(BT_U32BIT) number of bytes to read.
buff	(VOID*) location where data read will be stored.

Return Value

API_SUCCESS
API_BAD_PARAM
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.124 BusTools_MemoryRead2

Description

BusTools_MemoryRead2 provides direct read access to Abaco Systems 1553 board memory. It requires the calling application to request a specific memory region to read, accounting for the firmware version programmed on the respective board. Refer to either the *UCA32 Global Register Reference Manual* or the *MIL-STD-1553 Universal Core Architecture Manual* for the memory map applicable to the respective firmware host interface.

The calling application will supply the address within the region, the number of bytes/registers to read, and the address of a host buffer to receive the data. The address within the region is the offset from the start of the respective memory region and not the offset from the board's base address. This function will calculate the actual board offset based on the memory region.

Although it is normally not necessary to directly read or write board memory, the API provides this function to allow the application full access to board memory. This function is useful for optimizing user applications.

Prior to calling this function, the channel must be initialized by calling one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_MemoryRead2(cardnum, region, start, count, buff);

wStatus (BT_INT) status returned from this function.

cardnum (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

region (BT_INT) Memory Regions:

RAM	(0) Dual port RAM
HWREG	(1) Hardware Registers
RAMREG	(2) RAM Registers, (F/W 5.x or earlier)
HIF	(3) Host Interface
RAM32	(4) Dual port RAM

Region values supported by F/W Version 6.0 or greater only:

RELRAM32	(5) Dual port RAM
RELRAM	(6) Dual port RAM
TRIGREG	(7) Trigger registers
TTREG	(8) Time tag registers

BMRAM32	(9) Dual port RAM

start — (BT_U32BIT) memory address/register to be read

count — (BT_U32BIT) number of elements to read (must be less than 64 KBytes). The *count* parameter is interpreted as different size memory and register read operations, dependent on the board's firmware version. The size of each read access, and the meaning of *count* is defined below:

Region	V4/V5	V6
HIF	byte count	register count
HWREG	register count	register count
RAMREG	register count	N/A, error
RAM	byte count 1	byte count1
RAM32	byte count 2	byte count 2
RELRAM32	N/A, error	byte count 2
RELRAM	N/A, error	byte count 1
TRIGREG	N/A, error	N/A3
TTREG	N/A, error	N/A3
BMRAM32	N/A, error	32-bit elements2

NOTES

1: data read is a 16-bit value, so count must be even, (odd byte count will result in an error) count should be modulo 2, a non-modulo 2 value will be truncated to lower modulo 2 value

2: data read is a 32-bit value, count must be even, (odd byte count will result in an error), count should be modulo 4, a non-modulo 4 value will be truncated to lower modulo 4 value

3: data read is a single 32-bit register value, the count parameter is not used

buff — (VOID*) location to store data read. The size of each read access is defined as follows, dependent on the board's firmware version:

Region	V4/V5	V6
HIF	16-bit	32-bit
HWREG	16-bit	32-bit
RAMREG	16-bit	N/A, error
RAM	16-bit	16-bit
RAM32	32-bit	32-bit
RELRAM32	N/A, error	32-bit

RELRAM	N/A, error	16-bit
TRIGREG	N/A, error	32-bit
TTREG	N/A, error	32-bit
BMRAM32	N/A, error	32-bit

Return Value

API_SUCCESS
API_BAD_PARAM
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.125 BusTools_MemoryWrite(obsoleted)

Description

BusTools_MemoryWrite is obsolete but will remain as part of the API for legacy applications.

OS Support

Core API Function

Syntax

wStatus = BusTools_MemoryWrite(cardnum, addr, bcount, buff);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) busTools board number
addr	(BT_U32BIT) busTools memory address to be written
bcount	(BT_U32BIT) byte count.
buff	(VOID*) buffer from which data is transferred.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM

4.126 BusTools_MemoryWrite2

Description

BusTools_MemoryWrite2 provides direct write access to channel memory. It requires the caller to specify the memory region being written. All Abaco Avionics 1553 interface boards have four memory regions, Host Interface, Hardware Registers, File (RAM) Registers, and dual-port RAM. Each of these regions have a board dependent offset. Refer to the *MIL-STD-1553 Universal Core Architecture Manual* for the memory map for the different Abaco Avionics boards.

Although it is normally not necessary to directly read or write Abaco Systems 1553 board memory, this function is provided where the other API functions are either awkward to use, too slow, or do not provide the required functionality.

Prior to calling this function, the channel must be initialized by calling one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_MemoryWrite2 (cardnum, region, addr, bcount, buff);

wStatus — (BT_INT) status returned from this function.

cardnum — (BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

region — (BT_INT) Memory Regions:

RAM	(0) Dual port RAM
HWREG	(1) Hardware Registers
RAMREG	(2) RAM Registers, (F/W 5.x or earlier)
HIF	(3) Host Interface
RAM32	(4) Dual port RAM

Region values supported by F/W Version 6.0 or greater only:

RELRAM32	(5) Dual port RAM
RELRAM	(6) Dual port RAM
TRIGREG	(7) Trigger registers
TTREG	(8) Time tag registers

addr — (BT_U32BIT) memory address/register to be written

bcount — (BT_U32BIT) number of elements to write (must be less than 64K bytes). The count parameter is interpreted as different size memory and register write operations, dependent on the board's firmware version. The size of each write access, and the meaning of count is defined below:

Region	V4/V5	V6
HIF	byte count	register count
HWREG	register count	register count
RAMREG	register count	N/A, error
RAM	byte count 1	byte count1
RAM32	byte count 2	byte count 2
RELRAM32	N/A, error	byte count 2
RELRAM	N/A, error	byte count 1
TRIGREG	N/A, error	N/A3
TTREG	N/A, error	N/A3
BMRAM32	N/A, error	32-bit elements2

NOTES

1: data written is a 16-bit value, so count must be even, (odd byte count will result in an error) count should be modulo 2, a non-modulo 2 value will be truncated to lower modulo 2 value

2: data written is a 32-bit value, count must be even, (odd byte count will result in an error), count should be modulo 4, a non-modulo 4 value will be truncated to lower modulo 4 value

3: data written is a single 32-bit register value, the count parameter is not used

buff — (VOID*) location of data to be written. The size of each element is defined as follows, dependent on the board's firmware version:

Region	V4/V5	V6
HIF	16-bit	32-bit
HWREG	16-bit	32-bit
RAMREG	16-bit	N/A, error
RAM	16-bit	16-bit
RAM32	32-bit	32-bit
RELRAM32	N/A, error	32-bit
RELRAM	N/A, error	16-bit
TRIGREG	N/A, error	32-bit
TTREG	N/A, error	32-bit

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM

4.127 BusTools_PCI_Reset/BusTools_VME_Reset

Description

These functions enable or disable an Abaco Systems 1553 board's ability to respond to a PCI or VME reset request. The default condition after initialization is to ignore a PCI or VME reset signal; when in this configuration if a PCI or VME reset is initiated by the host, the board will continue operating. If you enable the PCI or VME reset with one of these functions, the board responds to a PCI or VME reset and stops operating, returning to its initial power-up state.

The board reset affects all channels on a PCI or VME device, not just the channel reference supplied in the *cardnum* parameter.

BusTools_PCI_Reset supports Abaco Systems PCI-based 1553 devices. BusToos_VME_Reset only supports the QVME-1553 and RQVME2-1553 boards.

Prior to calling this function, you must initialize the channel by calling one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function (PCI Reset valid for API version 5.90 and F/W version 4.20. VME Reset valid for API version 6.0 and F/W version 4.23)

Syntax

wStatus = BusTools_PCI_Reset (cardnum, reset_flag);
wStatus = BusTools_VME_Reset (cardnum, reset_flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
reset_flag	(BT_UINT) Reset Enable Flag: 1 - Enable Reset; 0 -Disable Reset

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.128 BusTools_Playback

Description

The Playback Function allows you to re-play previously captured MIL-STD-1553 data in real time on a standard 1553 bus. This function accepts a file recorded by the Bus Monitor and recreates the messages stored in the file as actual messages on the 1553 bus. Bus Playback can filter the records in the file by either record number or RT. When using the record number filter, the caller must define the start and stop record numbers via the respective parameters. When defining the RT's to include in the playback operation, they must be defined in a bit-encoded unsigned 32-bit integer containing the active and filtered RTs (bit 0=RT0, bit 1=RT1, etc.; 1 = playback is active for this RT, 0 = playback is inactive for this RT).

Playback uses a bmd or bmdx file recorded by the BusTools/1553-Analyzer or user application to recreate the bus traffic. Starting with BusTools/1553-API version 8.0 the output of the Bus Monitor must be formatted as a bmdx file. "bmdx" files contain a header record indicating whether the time-tag units are μs or ns. The header is followed by API_BM_MBUF records, matching the format of a bmd file; however, starting with BusTools/1553-API Version 8.0, the definition of the API_BM_MBUF datatype utilizes a 64-bit time-tag versus previous API revisions that utilize a 48-bit time-tag. Playback files recorded with API Versions before Version 8.0 cannot be used with BusTools/1553-API Version 8.0 or later. Bmdx files can only be used for playback with BusTools/1553-API Version 8.0 or later.

For message playback, the Playback function executes exclusive to the Bus Monitor; the Bus Controller and/or RT function on the respective channel should be disabled. If other functionality is active, Playback message content may be corrupted. It is an application requirement to ensure that all RT and the Bus Controller functionality is inactive during playback. An invocation of the function BusTools_Playback initiates playback processing, and that processing will continue executing as long as there are records remaining in the playback buffer(s). Playback progress can be monitored via the API_PLAYBACK_STATUS data structure provided in the API_PLAYBACK structure parameter with the BusTools_Playback invocation. To terminate the playback operations prior to completion, invoke BusTools_Playback_Stop.

Bus Playback relies on the time-tags in the Playback Input file (.bmd or .bmdx) to output the records onto the bus at the proper time. Bus Playback expects the time-tags to be monotonically increasing with record number or cycling every second. If the time-tags do not match either of these two conditions, Bus Playback will not function correctly. Bus Playback also has limited error replication ability. Bus Playback can handle No Response, Low Word Count, Parity, and Inverted Sync errors. Other message errors have an undetermined effect on Bus Playback.

Prior to calling this function, the channel must be initialized by calling one of the BusTools/1553-API Initialization functions.

OS Support

Windows and UNIX.

Syntax

wStatus = BusTools_Playback (cardnum, playbackData);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
playbackData	(API_PLAYBACK) Structure of playback data.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_PLAYBACK_INIT_ERROR
API_PLAYBACK_BAD_THREAD
API_PLAYBACK_BAD_FILE
API_PLAYBACK_BAD_EVENT
API_PLAYBACK_BUF_EMPTY
API_PLAYBACK_BAD_EXIT
API_PLAYBACK_BAD_MEMORY
API_PLAYBACK_DISK_READ

4.129 BusTools_Playback_Check

Description

BusTools_Playback_Check searches a Playback input file (file extension .bmd) for either out of sequence time-tags or large gaps in successive time-tags. Run this check on an input file prior to running playback to make sure that the file has no problems that affect how Playback will execute.

The Playback mode uses time-tags to determine when to process the next 1553 transaction. If the Playback Input file has time-tags that are out of sequence (e.g., a time-tag less than the previous), Playback will "hang" until the Playback timer rolls over. Playback will stay in this wait state for over 1 hour. Thus, to avoid this "hang-up", all time-tags in the Playback Input file must be in ascending order. The function returns API_PLAYBACK_TIME_ORDER if one or more time-tags is out of sequence in the file.

This function returns API_PLAYBACK_TIME_GAP if the time between any two successive time-tags is greater than the *wMin* parameter value as provided in the calling sequence.

The input arguments to this function are a Playback Input file name, (normally generated by the BusTools/1553 Analyzer), and a minimum time gap value. This function can be called at any time since it does not require an initialized 1553 board/session.

OS Support

Windows and UNIX.

Syntax

wStatus = BusTools_Playback_Check (bmd_file, wMin);

wStatus	(BT_INT) status returned from this function.
bmd_file	(char *) name of the bmd file to check.
wMin	(BT_UINT) number of minutes that define the minimum size of a time-tag gap.

Return Value

API_PLAYBACK_TIME_ORDER
API_PLAYBACK_TIME_GAP

4.130 BusTools_Playback_Stop

Description

BusTools_Playback_Stop terminates playback operations. It should be called to prematurely terminate playback execution or at the end of a successful playback. See BusTools_Playback for a complete description of the bus playback functions.

Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions.

OS Support

Windows and UNIX.

Syntax

wStatus = BusTools_Playback_Stop (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.131 BusTools_ReadBoardTemp

Description

BusTools_ReadBoardTemp reads the temperature sensors on boards equipped with this feature. The value returned is the internal temperature of the temperature sensor mounted on the PWB, in °C. Negative temperatures will be the 2's compliment of the value.

The RAR15-XMC-IT/RAR15XF boards have five different temperature options. Other designs have a single option. If this function is called for a board without a temperature sensor, the function returns API_HARDWARE_NOSUPPORT. If an application references a temperature location other than INTERNAL for any board other than an RAR15-XMC-IT/RAR15XF, the selected sensor will default to INTERNAL.

Prior to calling this function, initialize the channel using one of the BusTools/1553-API Initialization functions. All channels on a board read the same sensors and return the same respective values.

OS Support

Core API Function

Syntax

wStatus = BusTools_ReadBoardTemp (cardnum, location, tmp);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_INT) channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
location	(BT_UINT) Sensor location: INTERNAL (0) TFPGA (1) TZBT (2) TCHANNEL (3) TBOARD (4)
tmp	(BT_INT *) location to store the sensor temperature value, in °C.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_OVER_TEMP_ALARM
API_HARDWARE_NOSUPPORT

4.132 BusTools_ReadVMEConfig

Description

BusTools_ReadVMEConfig returns the contents of the VME configuration registers in the A16 region on a VME/VXI 1553 interface board. The region contains data on the setup and version of the board. Reading this data can help with debugging VME setup problems, such as memory map and interrupts. This function only applies to the QVME-1553 and RQVME2-1553 boards.

Prior to calling this function, the channel must be initialized by invoking BusTools_API_InitExtended.

OS Support

Core API Function

Syntax

wStatus = BusTools_ReadVMEConfig (cardnum, vdata);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_INT) channel reference to the respective 1553 board/channel session. Valid range is 0 to 15.
vdata	(BT_U16BIT *) pointer to an array of unsigned short integers used to store the contents of the configuration registers. The destination must be allocated for 12 or more locations.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_HARDWARE_NOSUPPORT

4.133 BusTools_RegisterFunction

Description

BusTools_RegisterFunction supports application-specific message processing for specific MIL-STD-1553 protocol events by creating an interrupt processing thread and an Event Object utilized by the ISR and registering a user callback function containing the application-specific processing.

1553 Events are detected using interrupt queue polling or hardware interrupts. The polling/interrupt option is selected in the BusTools/1553-API initialization via the *flag* parameter. Refer to BusTools_API_OpenChannel or BusTools_API_InitExtended for details on this parameter.

Polling/Interrupt Options compatible with BusTools_RegisterFunction:

API_SW_INTERRUPT	-	S/W Polled Interrupt
API_HW_ONLY_INT	-	H/W Only Mode
API_HW_INTERRUPT	-	H/W and S/W Mode

When calling BusTools_RegisterFunction, the application supplies an Interrupt Register/Filter/FIFO Structure (API_INT_FIFO) as an input argument. This structure supplies the API with the name of the user callback function, information about how to handle the interrupt, thread priority, and a list of desired interrupt events. This list can contain one or many events, specified by the contents of the API_INT_FIFO filter structure. Table 4-3 provides a list of available events. Both the API and the user's callback function globally access the FIFO structure. The application must ensure that Register/Filter/FIFO structure is global and that no other process overwrites this FIFO structure.

The interrupt processing thread waits on the Event Object until an interrupt event occurs matching those specified in the API_INT_FIFO filter structure. For each event in the interrupt queue matching the events in the event list, the interrupt processing thread populates an entry in the user-supplied FIFO structure. The thread then invokes the user callback function, passing it the card number and a pointer to the FIFO containing the event data. The user callback function should process all entries in the FIFO, and when complete, return control to the API. The thread continues to "block" on an enabled event until the API detects another event. The API supports up to MAX_REGISTER_FUNCTION number of these threads per board (currently 64 threads).

It is important that the user function process *all* entries in the FIFO. An Event Object activates the thread only once, and there may be multiple entries in the FIFO. It is also possible to signal the Event Object while the thread is executing. In that case, the API might call the user function to process zero entries (because it already processed the entry added during its current execution cycle).

Set the priority of the interrupt thread as appropriate for the application. Normally, you adjust the priority of the thread based on importance. For example, a display thread would have a lower priority than a thread performing 1553 message processing.

Un-register all threads created by invoking this function before invoking BusTools_API_Close.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function.

Syntax

wStatus = BusTools_RegisterFunction (cardnum, &sIntFIFO, wFlag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_INT) channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
sIntFIFO	(API_INT_FIFO *) Pointer to the Interrupt Register/Filter/FIFO Structure. This is a Thread Interrupt Filter, Control and FIFO structure.
wFlag	(BT_UINT) flag indicating what to do: UNREGISTER_FUNCTION (0) = un-register this function. REGISTER_FUNCTION (1) = register the function, structure and FIFO.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BUSTOOLS_FIFO_BAD
API_BUSTOOLS_TOO_MANY
API_BUSTOOLS_NO_OBJECT

Notes

For programming examples (BC, RT, and BM), see the TST_ALL.C example “C” program supplied with the BusTools/1553-API. For additional information, see the discussion on the API_INT_FIFO structure in Chapter 7, “Data Structures”.

The API_INT_FIFO structure definition describes the parameter list used by the API to define the user callback function. The application callback function definition

must match this parameter set. The following line of code shows this calling sequence where *FunctionName* is the name passed in the API_INT_FIFO structure.

```
BT_INT stdcall FunctionName(BT_UINT cardnum,
                            struct api_int_fifo *sIntFIFO)
```

The application can update the API_INT_FIFO filtering structure at any time, and changes take effect immediately. Since the API is interrupt driven, caution should be taken when modifying the filtering to avoid encountering excessive, unwanted events.

A feature added to BusTools_RegisterFunction, starting at API Version 4.46, allows you to refine the events causing interrupts on an RT address, Subaddress, or Transmit/Receive basis. This allows you to specify interrupts on specific events such as "message error". To use the feature, set the API_INT_FIFO member "EventInit" to "USE_INTERRUPT_MASK", then select the desired interrupt status bit using the EventMask. When you set this mask, the API augments the events in Table 4-3 with the bits specified in the interrupt status word.

For example, select EVENT_BC_MESSAGE to get interrupts on all BC messages, then set EventMask to BT1553_INT_ME_BIT. You get all Bus Controller Messages that have the message error bit set. If you set EventMask to NO_ERRORS you get all messages with only the BT1553_INT_END_OF_MESS set. This allows you to have separate interrupt functions for error-free messages from messages with errors.

The API_INT_FIFO structure is updated by this function call if *flag* is "REGISTER_FUNCTION". Do not modify any fields marked "RO", as the API uses these parameters internally.

Before exiting, the application should invoke BusTools_RegisterFunction a second time with a *flag* of UNREGISTER_FUNCTION for each API_INT_FIFO you have set up. This terminates the thread and releases the objects. Do not invoke BusTools_RegisterFunction with a *flag* of zero to terminate the thread within the thread itself. If the thread function wishes to terminate itself, it should return with any non-zero return value. This causes the API to terminate the thread and free allocated resources.

Table 4-3 Interrupt Events

Event	Event Description
EVENT_IMMEDIATE	Immediately calls user function without processing interrupt queue data.
EVENT_EXT_TRIG	External Trigger Event
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
EVENT_BM_TRIG	BM trigger event (start/stop)

Event	Event Description
EVENT_BM_START	BM started (BusTools_BM_StartStop)
EVENT_BM_STOP	BM stopped (BusTools_BM_StartStop)
EVENT_BM_OVRFLW	BM 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 extends beyond a single frame
EVENT_HP_MF_OVFL	High Priority aperiodic message list extends beyond a single frame
EVENT_BC_BSY_OVFL	BC busy overflow
EVENT_API_OVERFLO	BM API Recorder buffer overflowed
EVENT_HW_OVERFLO	BM HW Recorder buffer overflowed

Debugging Tips

The following is a list of debugging techniques:

- Do not allocate the FIFO structure on the stack. It will be overwritten when the function returns. The FIFO structure must be present throughout the execution of the registered function. Declare the structure "static" if needed.
- Link your application with the "multi-threaded" version of the library or libraries.
- Make sure that the function registered uses the proper calling sequence and declaration. It must match the function prototype given in the API_INT_FIFO structure in the Busapi.h file
- The BusTools_RegisterFunction interface uses standard threads for Windows and should be compatible with other standard Wndows products that support multiple threads.
- BusTools_RegisterFunction uses POSIX threads (pthread) for UNIX systems and is compatible with all UNIX systems supporting POSIX threads.
- VxWorks has the option of using VxWorks threads or POSIX threads. The default is VxWorks threads. Edit target_defines.h to change this setting to use POSIX threads.
- BusTools_RegisterFunction has portability to other O/S specific thread by filling in template common thread functions
- If the BusTools_RegisterFunction call succeeds (returns API_SUCCESS), the new thread and all of the control structures have been successfully created.

4.134 BusTools_RS485_TX_Enable

Description

BusTools_RS485_TX_Enable enables RS-485 differential discrete output. It allows the application to specify the respective input/output as an RS-485 differential output by applying the output enable either individually or to multiple differential outputs in a single invocation. All differential discretes are automatically set for RS-485 differential input after initialization.

The function only applies to boards that have RS-485 differential discrete I/O.

Initialize at least one channel using one of the BusTools/1553-API Initialization functions before using the RS-485 Discrete I/O.

OS Support

Core API Function

Syntax

wStatus = BusTools_RS485_TX_Enable (cardnum, enable, mask);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
enable	(BT_U16BIT) the input/output configuration of the differential discrete I/O.
mask	(BT_U16BIT) a data mask to preserve the input/output configuration of intentionally unmodified RS-485 discretes.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

NOTE

RS-485 discretes are shared by all 1553-channels on the board. If multiple processes are interacting with individual 1553-channels on a board, only one process should configure the RS-485 discretes. The remaining applications can share the discrete I/O channels, but the application must ensure the processes do not conflict when using global resources.

4.135 BusTools_RS485_Set_TX_Data

Description

BusTools_RS485_Set_TX_Data sets the state of individual or multiple RS-485 differential discrete outputs with a single invocation. The data and mask parameters are combined to affect the state of only the specified differential outputs. The application can only alter the output state of differential discretes previously configured as outputs using BusTools_RS485_TX_Enable.

The function only applies to boards that have RS-485 differential discrete I/O.

Initialize at least one channel using one of the BusTools/1553-API Initialization functions before using the RS-485 Discrete I/O.

OS Support

Core API Function

Syntax

wStatus = BusTools_RS485_Set_TX_Data (cardnum, rsdata, mask);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rsdata	(BT_U16BIT) the differential discrete state(s).
mask	(BT_U16BIT) This variable is a data mask to preserve the state of intentionally unmodified RS-485 discrete outputs.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

NOTE

RS-485 discretes are shared by all 1553-channels on the board. If multiple processes are interacting with individual 1553-channels on a board, only one process should configure the RS-485 discretes. The remaining applications can share the discrete I/O channels, but the application must ensure the processes do not conflict when using global resources.

4.136 BusTools_RS485_ReadRegs

Description

BusTools_RS485_ReadRegs reads the RS-485 component registers. This allows the application to read the differential discrete output enable setting as well as the current output and input states (respective to the output enable setting for each).

The function only applies to boards that have RS-485 differential discrete I/O.

Initialize at least one channel using one of the BusTools/1553-API Initialization functions before using the RS-485 Discrete I/O.

OS Support

Core API Function

Syntax

wStatus = BusTools_RS485_ReadRegs (cardnum, regval, rsdata);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
regval	(BT_INT) The RS-485 register to read: RS485_TXEN_REG – Transmit enable register RS485_TXDA_REG – Transmit data register RS485_RXDA_REG – Receive data register
rsdata	(BT_U32BIT *) location to write the register data.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

NOTE

The eight RS-485 discrete channels are shared by all 1553-channels on the board. If you have multiple processes running different 1553-channels, only one process should configure the RS-485 discretes. The remaining applications can share the discrete channels, but you must ensure that processes do not conflict in use these channels.

4.137 BusTools_RT_AbufRead

Description

BusTools_RT_AbufRead reads the information stored in the RT Address Buffer for the specified RT address. Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

While the RT enable bits in the hardware rt_address_buffer are active low (0 = enable), the RT enable bits in the API_RT_ABUF structure are active high (1 = enable).

All four words of the RT Address Buffer are returned by this function:

1. The RT Enables/Dynamic Bus Control/BIT Word location/Inhibit Terminal Flag
2. The RT Status Word
3. The Last Command Word
4. The Bit Word

See the function BusTools_RT_AbufWrite for more information about the RT Address Buffer parameters.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_AbufRead (cardnum, rtaddress, abuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddress	(BT_UINT) RT address. Valid range is 0 to 31.
abuf	(API_RT_ABUF*) location to be written with Address Buffer information.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_ADDR

4.138 BusTools_RT_AbufWrite

Description

BusTools_RT_AbufWrite writes the RT Address Control Block (API_RT_ABUF) structure to the RT Address Buffer for the specified RT address, controlling the operation of that RT. Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init. Information contained within the RT Address Control Block (API_RT_ABUF) structure are described below.

4.138.1 The RT Enable Bits

These bits control whether the specified RT is active on bus A and/or bus B. While the RT enable bits in the hardware rt_address_buffer are active low (0 = enable), the RT enable bits in the API_RT_ABUF structure are active high (1 = enable).

4.138.2 The Inhibit Terminal Flag

The application must initialize this multi-function Inhibit Terminal flag. This flag controls the Inhibit Terminal flag and the Dynamic Bus Control Mode, as well as other functionality.

"RT_ABUF_ITF" controls the reporting of the Terminal Flag in the message status word. When this bit is set, the RT does not set the terminal flag bit, despite the bit value in the rt_status_word (see RT Status Word Section below). This host clears the Inhibit Terminal Flag bit during setup. A remote terminal receiving the Inhibit Terminal Flag Bit mode code (00110) clears this bit. A remote terminal receiving the Override Inhibit Terminal Flag Bit mode code (00111) sets this bit. The "Override Inhibit Mode Command" does not affect the status word of the mode command itself; rather, it allows setting the terminal flag in the status word of subsequent messages.

"RT_ABUF_DBC_ENA" controls the enabling Dynamic Bus Control (DBC) mode codes. When this bit is set, the RT acts on the DBC mode code (0000) by starting the Bus Controller in accordance with MIL-STD-1553 paragraph 4.3.3.5.1.7.1. You must also legalize Mode Code 0 with a call to BusTools_RT_AbufWrite (enable word count 0, transmit, for subaddress 0 and optionally subaddress 31).

"RT_ABUF_DBC_RT_OFF" controls the operation of the RT when the RT receives and accepts a DBC mode. This bit only affects multi-function boards. On multi-function boards, this bit control whether the RT shuts down or continues when the RT receives a DBC mode code. Setting this bit causes the RT receiving the DBC mode code to stop. All other RTs continue running. Clearing this bit allows the RT to continue running. When a single-function or dual-function board receives a DBC mode code, it shuts down the RT prior to starting the BC.

"RT_ABUF_MBUF_BWD" controls whether mode code 19, Transmit BIT Word, uses the bit word stored in the Address Buffer or uses the bit word buffered in data word

zero (0) of the RT Message buffer. If bit 4 (0x10) of the inhibit terminal flag is set, the data for mode code 19 comes from the RT message buffer. If bit 4 is reset, the data is stored in the bit word of the Address buffer.

"RT_ABUF_EXT_STATUS" enables extended status mode. Under normal RT operation, the status word applies to all RT sub-addresses. Enabling extended status allows the application to set status for a specific message buffer response. Use BusTools_RT_MessageWriteStatusWord to set the extended status based on RT address, Sub address, Transmit or Receive and buffer.

"RT_ABUF_MON_ENA" Places the RT into Monitor Mode. This is normally done through a call to BusTools_RT_MonitorEnable. In Monitor Mode the RT only monitors bus traffic to the RT specified in RT address. The RT does not respond to commands. There can be another device or channel responding as the RT. This function allows the application to process all transaction to the RT address. You can manually set Monitor Mode using this macro. Note: in older versions of BusTools/1553-API this parameter was not present and the Inhibit terminal Flag was sometime set to one (1). Make sure if you have using older applications that the flag is set to zero (0) or any the parameters above. Otherwise you may inadvertently be put into monitor mode.

4.138.3 The RT Status Word

This is the status word returned by the RT. It is programmed during initialization, but it may be altered at any time by the software. The hardware may also set some of the status word bits in accordance with MIL-STD-1553B. Your software must initialize this word before running the RT by clearing all of the bits except the RT address. The API fills in the correct RT address when this function is called. You can change the RT address by using the error injection features of the API (see BusTools_EI_EbufWrite).

4.138.4 The RT Last Command Word

When the RT detects a command word on the 1553 bus, it is stored here for use in the "Transmit Last Command Word" mode command. This is an internal word used by the hardware but may be initialized by the software prior to running the RT.

4.138.5 The BIT Word

This is the 16-bit word used by the Transmit Built-In-Test (BIT) mode command. The host normally clears this word during setup and updates it during reception of an initiate self-test mode command. This capability requires application code to read the self_test bit in the message_status word or set the self_test bit in the rt_interrupt_enables word and have a self-test function to exercise the hardware.

4.138.6 Single RT Mode

Starting with F/W version 5.00, there is a channel configuration option that sets the channel to run in RT Validated (single-RT) mode. In this mode, the RT will pass the RT Validation Test Plan (1553B Handbook, Appendix A), but only a single RT can

run at a time. Select the RT address for the single RT mode by calling BusTools_RT_AbufWrite. If you call this function more than once with a different RT address, it returns API_SRT_OVERRIDE. This is informational only; to inform you that the single RT address has changed.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_AbufWrite (cardnum, rtaddress, abuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddress	(BT_UINT) RT address. Valid range is 0 to 31.
abuf	(API_RT_ABUF*) the application defined RT Address Control Block structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_SRT_OVERRIDE
API_RT_ILLEGAL_ADDR

4.139 BusTools_RT_AutoIncrMessageData

Description

BusTools_RT_AutoIncrMessageData creates an interrupt-invoked thread that automatically increments a data word in a specified RT message. This function is only applicable to operating systems supporting the function BusTools_RegisterFunction and requires a subsequent invocation to terminate the increment thread.

This function can be used with software interrupts for RT message rates slower than 10 milliseconds. If the RT message rate is faster than 10 milliseconds, the hardware interrupt option must be used. Both software and hardware interrupts are defined in the invocation of the BusTools/1553-API Initialization function.

BusTools_RT_AutoIncrMessageData only supports auto-increment with a single data word per RT/SA/TX combination using a single RT message buffer. For this reason, during setup of the RT the invocation of BusTools_RT_CbufWrite must have the buffer count set to one. If an application attempts to auto-increment more the one data word per RT/SA/TX combination, an API_RT_AUTOINC_INUSE error will be encountered. The application defines the RT address, subaddress, data word (0 – 31) to increment, an increment value, a start value, an increment rate, and a maximum value for the message. This function sets the specified data word to the start value and creates a thread that increments the data word by the increment value, each 'rate' times the message is transmitted. Setting the rate parameter to "1" causes the thread to increment on every message. The specified data word increments from the start value to the maximum value and then resets to the start value.

NOTE

This function works only with the transmit buffer for RT→BC messages.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function.

Syntax

```
wStatus = BusTools_RT_AutoIncrMessageData ( cardnum, rtaddr,  subaddr,
          data_wrd,  start,  incr, rate, max, sflag );
```

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

rtaddr	(BT_INT) The RT address for the message. Valid range is 0 to 31.
subaddr	(BT_INT) The subaddress for the message. Valid range is 0 to 31.
data_wrd	(BT_INT) The data word to increment (0-31).
start	(BT_U16BIT) Start value.
incr	(BT_U16BIT) Increment value.
rate	(BT_INT) Increment rate
max	(BT_U16BIT) Maximum increment value.
sflag	(BT_INT) 0 = Stop increment thread, 1 = Start increment thread.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED

4.140 BusTools_RT_CbufbroadRead

Description

BusTools_RT_CbufbroadRead reads the Broadcast Control Buffer for the specified subaddress, applicable to all RT's. This buffer contains the legalization bits used to enable or disable broadcast messages for the specified subaddress. The Broadcast Control Buffer contains a single bit for each possible RT address and each possible word count (for a total of 32 x 32 entries, or 1024 bits).

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_CbufbroadRead (cardnum, subaddr, tr, apicbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
subaddr	(BT_UINT) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
apicbuf	(API_RT_CBUFBROAD*) location where the RT Broadcast Control Buffer content will be written.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_SUBADDR

4.141 BusTools_RT_CbufbroadWrite

Description

BusTools_RT_CbufbroadWrite writes the Broadcast Control Buffer for the specified subaddress, applicable to all RT's. This buffer contains the legalization bits used to enable or disable broadcast messages for the specified subaddress. The Broadcast Control Buffer contains a 32-bit word for each possible RT address (0 through 30), with a single bit for each possible word count (for a total of 32 x 32 entries, or 1024 bits). These bits determine if the message is legal, and if so, the firmware sets the Broadcast Message Received bit in the associated RTs 1553 Status Word. A set bit indicates that the message is legal.

A clear bit indicates that the word count is not legal. All bits should be clear for any RTs that are not being simulated by this board.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_CbufbroadWrite (cardnum, subaddr, tr, apicbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
subaddr	(BT_UINT) RT subaddress (0 - 31).
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
apicbuf	(API_RT_CBUFBROAD*) location from where the RT Broadcast Control Buffer content will be read.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_SUBADDR

Notes

Invoke this function for each subaddress, transmit/receive combination that is being enabled for Broadcast Receive or Transmit on this channel.

Broadcast is enabled or disabled by the BusTools_SetBroadcast function.

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). The firmware then sequences through the 31 control words and checks to see if 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.

Word counts map to bit numbers as follows:

- Bit 0 (0x00000001) – Word count 32 (LSB)
- Bit 1 (0x00000002) – Word count 1
- Bit 2 (0x00000004) – Word count 2
- …
- Bit 30 (0x40000000) – Word count 30
- Bit 31 (0x80000000) – Word count 31 (MSB)

4.142 BusTools_RT_CbufRead

Description

BusTools_RT_CbufRead reads the RT Control buffer for the specified RT subunit. This structure contains a legalization bit for each possible word count for the rt address/subaddress/transmit/receive subunit. This function also returns the number of RT Message buffers originally defined for this subunit. This value is '0' for any subunit which has not yet been defined.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_CbufRead (cardnum, rtaddr, subaddr, tr, mbuf_count, apicbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address. Valid range is 0 to 31.
subaddr	(BT_UINT) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
mbuf_count	(BT_UINT*) number of RT Message buffers originally defined for this RT subunit returned by function.
apicbuf	(API_RT_CBUF *) location to write the RT Control Buffer information.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_ADDR
API_RT_ILLEGAL_SUBADDR
API_RT_ILLEGAL_TRANREC
API_RT_CBUF_BROAD

4.143 BusTools_RT_CbufWrite

Description

For each RT there are 32 subaddresses defined for transmit and 32 subaddresses defined for receive. Each combination of RT/subaddress/TR is called a subunit. A structure, API_RT_CBUF, defines an RT Control Buffer, which specifies legal word counts for a subunit.

When invoking BusTools_RT_CbufWrite, the application specifies a subunit, the number of RT Message Buffers to allocate, and the RT Control Buffer, for the specified subunit. This function writes the RT Control Buffer for the specified subunit allocates and clears (fills with zero) the RT Message Buffers. For buffers used in "transmit" messages, the application should write the desired transmit data to the RT Message Buffers using the BusTools_RT_MessageWrite function.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

If the application intends to use Dynamic Bus Control (DBC), this function must be invoked to legalize Mode Code 0 Transmit, which is the DBC mode code. BusTools_RT_AbufWrite must also be invoked to enable the DBC mode.

This function operates differently depending on how it was previously called for the specified subunit and the current arguments.

- **RT Message Buffer Allocation for SA0 and SA31**: If either subaddress 0 or 31 is specified, or SA31 as a mode code is enabled, then the call is processed as if SA0 was specified. The SA31 entry in the RT Message Buffer is modified to point to the SA0 buffer.
  This takes care of the MIL-STD-1553B option of responding to a mode code on either SA0 or SA31.
- **Using the Default RT Message Buffers (Data Wrap)**. Data wrap is the state where the subunit's data is transmitted or received via the default RT Message Buffer. Enable this state by calling this function with a buffer count of zero. Disable this state by calling this function with a buffer count that is not zero. Ensure that the RT Control Buffer structure legalizes all wordcounts. Load the default RT Message Buffers with BusTools_RT_MessageWriteDef. There is one default buffer for each RT. This default buffer is used to implement the MIL-STD-1553 Notice 2 paragraph 30.7 requirement to support "Data Wrap".
- **RT Message Buffers as a circular linked list or single-pass linked list**: If *mbuf_count* is negative, *mbuf_count* buffers are created and initialized and the buffers are linked into a one-shot list. If *mbuf_count* is positive, the buffers are linked into a circular list.

- **Update wordcount mask and/or reset the RT data buffer list pointer to a specified buffer**: If the RT Message Buffers for this subunit are NOT the default Message Buffers, MODIFY the current RT Control Buffer with the specified wordcount mask.
    - If the count parameter is less than the number of RT Message Buffers allocated, change the current RT Message Buffer pointer to the MBUF specified by the count parameter.
    - If the count parameter is equal to the number of RT Message Buffers allocated, do not modify the current MBUF pointer in the RT Control Buffer.
    - If the count parameter is greater than the number of RT Message Buffers allocated, return an error.

If none of the above conditions is true, this function allocates a new RT Control Buffer and the specified number of RT Message Buffers.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_CbufWrite (cardnum, rtaddr, subaddr, tr, mbuf_count, apicbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address. Valid range is 0 to 31.
subaddr	(BT_UINT) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
mbuf_count	(BT_INT) number of RT Message buffers allocated (see description).
apicbuf	(API_RT_CBUF*) location of the application defined RT Control Buffer.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_ADDR
API_RT_ILLEGAL_SUBADDR

API_RT_ILLEGAL_TRANREC
API_RT_CBUF_BROAD
API_RT_TOOMANY_MBUFS
API_RT_MEMORY_OFLOW

4.144 BusTools_RT_Checksum1760

Description

BusTools_RT_Checksum1760 calculates a checksum according to the algorithm described in Appendix B Section B.4.1.5.2.1 of the *Department of Defense Interface Standard for Aircraft/Store Electrical Interconnect Systems* MIL-STD-1760C Manual.

When each data word (including the checksum word) of a message is rotated right cyclically by a number of bits equal to the number of preceding data words in the message, and all the resultant rotated data words are summed using modulo 2 arithmetic to each bit (no carries), the sum shall be zero.

Two examples show messages satisfying the checksum algorithm.

Example 1

Four Word Message:

1st Word 0000-0000-0000-0001 (0001 hex.) data

2nd Word 1100-0000-0000-0000 (C000 hex.) data

3rd Word 0000-1111-0000-0000 (0F00 hex.) data

4th Word 0001-1110-0000-1011 (lE0B hex.) checksum word

Example 2

Six Word Message:

lst Word 0001-0010-0011-0100 (1234 hex.) data

2nd Word 0101-0110-0111-1000 (5678 hex.) data

3rd Word 1001-1010-1011-1100 (9ABC hex.) data

4th Word 1101-1110-1111-0000 (DEF0 hex.) data

5th Word 0000-0000-0000-0000 (0000 hex.) data

6th Word 1000-1111-0010-0000 (8F20 hex.) checksum word

Pass a pointer to the API_RT_MBUF_WRITE structure, a pointer to unsigned short (BT_U16BIT) to hold the calculated checksum and the number of data words in the RT message. Prior to calling this function, the application must define the data in the API_BC_MBUF structure. This function calculates the checksum and writes it into the last data location.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_Checksum1760 (mbuf , cksum, wdcnt);

wStatus	(BT_INT) status returned from this function.
mbuf	(API_RT_MBUF_WRITE *) pointer to RT message write structure.
cksum	(BT_U16BIT) location to write the checksum.
wdcnt	(int) word count for the RT message.

Return Value

API_SUCCESS

4.145 BusTools_RT_Init

Description

BusTools_RT_Init performs full initialization of RT functionality for the specified channel. The following actions are performed:

- Check for any illegal conditions (exit with error if these conditions aren't true).
- Initializes board memory.
- Assures available memory starts at the beginning of a segment (address = 10000).
- Allocates the filter buffer.
- Allocates the address control blocks.
- Allocates the default message buffer.
- Allocates a broadcast control buffer (if broadcast is enabled).
- Allocate a default control buffer (non broadcast).
- Setup PC data structures.
- Show no RTs running.

Prior to invoking this function, initialize the channel using one of the BusTools/1553-API initialization functions.

The *testflag* parameter *must* be set to zero. The non-zero state is used by the BusTools GUI program to perform a "wrap-around" test and is not directly applicable to a user-written program.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_Init (cardnum, testflag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
testflag	(BT_UINT) Test Flag. Must be set to zero.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_RUNNING
API_MEMORY_OFLOW

4.146 BusTools_RT_GetRTAddr

Description

BusTools_RT_GetRTAddr returns the hardwired RT address for boards supporting this feature, see Table 1-2. 1553 Board Feature Guide. The QVME-1553 and RQVME2-1553 boards support hardwired RT addressing for all channels. The QPMC-1553, QPM-1553, RPCIe-1553, and QCP-1553 have hardwired RT addressing available on channels 1 and 2. Other boards allow hardwired RT addressing on channel 1.

If the board is using hardwired RT addressing for the selected channel, this function will return a status of API_SUCCESS, with *rtaddr* containing the RT address. You must use that RT address when configuring the RT if you want that channel to operate as the hardwired RT. You can override the RT address by ignoring that RT address and using any other RT address when you configure the channel. If the hardwire RT address equals BTD_RTADDR_PARITY, that is an indication of invalid parity in the RT address detected during initialization.

This function returns API_HARDWARE_NOSUPPORT for any board or channel not supporting hardwired RT addressing, and API_NO_HARDWIRE_RT with *rtaddr* set to -1 for any supported channel not using hardwired RT addressing.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_GetRTAddr (cardnum, rtaddr);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_INT *) location to receive the hardwired RT address value.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_NO_HARDWIRE_RT
BTD_RTADDR_PARITY
API_HARDWARE_NOSUPPORT

4.147 BusTools_RT_GetRTAddr1760

Description

BusTools_RT_GetRTAddr1760 returns the hardwired RT address in the same manner as BusToolsGetRTAddr; however, an option is provided to read the latched data or the current value of the hardwired address line. On power-up the board latches the state of the hardwired RT address lines into a register. After the firmware is operational, the hardwired RT address lines can change based on external input, but the latched data remains fixed. This function provides the option to read either the latched or current hardwired RT address line values.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function (F/W V4 and V5 only)

Syntax

wStatus = BusTools_RT_GetRTAddr1760 (cardnum, aflag, rtaddr);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
aflag	(BT_UINT) flag to determine the read options LATCH_DATA (0) latched data CURRENT_DATA (1) current data
rtaddr	(BT_INT *) location to receive the hardwired RT address value.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_NO_HARDWIRE_RT
BTD_RTADDR_PARITY
API_HARDWARE_NOSUPPORT

4.148 BusTools_RT_MessageGetaddr

Description

BusTools_RT_MessageGetaddr returns the physical board offset of the specified channel's RT Message buffer. The application must specify the message buffer (MBUF) number for a specific RT subunit (combination of RT/subaddress/TR) to reference.

Typically, this function is used by an application that repeatedly updates a specific RT message using the BusTools_MemoryWrite function instead of the BusTools_RT_MessageWrite function.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_MessageGetaddr (cardnum, rtaddr, subaddr, tr, mbuf_id, mbuf_offset);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address. Valid range is 0 to 31.
subaddr	(BT_UINT) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
mbuf_id	(BT_UINT) RT Message buffer number ("0" based).
mbuf_offset	(BT_U32BIT *) location to receive the memory offset to the specified RT_MBUF structure on the board, as defined in the Hardware Reference Manual (RT_MESSAGE_BUFFER).

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_ILLEGAL_ADDR
API_RT_ILLEGAL_SUBADDR
API_RT_ILLEGAL_TRANREC
API_RT_ILLEGAL_MBUFID
API_RT_NOTINITED

4.149 BusTools_RT_MessageGetid

Description

This function converts a physical board offset of the specified channel's RT Message buffer to an RT subunit (combination of RT/subaddress/TR) and message buffer number. The specified hardware address is a byte offset from the beginning of memory on the Abaco Systems 1553 board. The address must be in the range 0x00000000 to 0x0003FFFF.

Typically, this function is used to process RT messages in the interrupt queue. The RT message buffer address supplied as a parameter to this function is converted to an RT subunit address and message number, supporting subsequent message retrieval via BusTools_RT_MessageRead.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_MessageGetid (cardnum, addr, rtaddr, subaddr, tr, mbuf_id);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
addr	(BT_U32BIT) physical board offset of the specified channel's RT Message buffer.
rtaddr	(BT_UINT *) location to receive the RT address for the referenced message. Valid range is 0 to 31.
subaddr	(BT_UINT *) location to receive the RT subaddress for the referenced message. Valid range is 0 to 31.
tr	(BT_UINT *) location to receive the RT transmit/receive flag for the referenced message; 0 = receive, 1 = transmit.
mbuf_id	(BT_UINT *) location to receive the RT Message buffer number for the specified RT subunit ("0" based).

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_MBUF_NOMATCH

4.150 BusTools_RT_MessageRead

Description

BusTools_RT_MessageRead reads the specified RT Message buffer from board memory. Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_MessageRead (cardnum, rtaddr, subaddr, tr, mbuf_id, mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address. Valid range is 0 to 31.
subaddr	(BT_UINT) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
mbuf_id	(BT_UINT) RT Message buffer number (“0” based).
mbuf	(API_RT_MBUF_READ *) address of structure to be populated by BusTools_RT_MessageRead.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_ADDR
API_RT_ILLEGAL_SUBADDR
API_RT_ILLEGAL_TRANREC
API_RT_ILLEGAL_MBUFID

4.151 BusTools_RT_MessageBufferNext

Description

BusTools_RT_MessageBufferNext returns the next RT Message Buffer number from the respective channel memory. Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_MessageBufferNext (cardnum, rtaddr, subaddr, tr, mbuf_id);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address. Valid range is 0 to 31.
subaddr	(BT_UINT) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
mbuf_id	(BT_UINT*) RT Message buffer number (0-based).

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED

4.152 BusTools_RT_MessageWrite

Description

BusTools_RT_MessageWrite writes the specified information to the specified RT Message buffer in the respective channel's memory. Prior to invoking this function, the application must allocate Message Buffers for the specified RT subunit (combination of RT/subaddress/TR) using BusTools_RT_CbufWrite.

The data for transmit messages and the Interrupt Enable Bits are the major parts of the RT Message buffer that you must initialize in channel memory when using the RT function. The Interrupt Enable bits control whether the firmware will generate interrupts when the RT processes this message. The API (and the application) can use this interrupt to trigger message processing functions. For example, you must generate an interrupt when you enable Dynamic Bus Control. This triggers the user process that starts the Bus Controller and optionally switches off the RT after receiving the DBC mode code. In that case, you must enable at least "BT1553_INT_END_OF_MESS" interrupts in the "enable" control word.

On Big Endian systems defining the WORD_SWAP macro in target_defines.h, the contents of the RT data buffer is word swapped. If you need to maintain the message data endian format you should save a local copy.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_MessageWrite (cardnum, rtaddr, subaddr, tr, mbuf_id, mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address. Valid range is 0 to 31.
subaddr	(BT_UINT) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
mbuf_id	(BT_UINT) RT Message buffer number ("0" based).
mbuf	(API_RT_MBUF_WRITE *) address of the RT Message Buffer structure.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_ADDR
API_RT_ILLEGAL_SUBADDR
API_RT_ILLEGAL_TRANREC
API_RT_ILLEGAL_MBUFID

4.153 BusTools_RT_MessageWriteDef

Description

Abaco Systems 1553 boards maintain a default (or "garbage collection") message buffer for each RT. The default message buffer is used for any messages to or from any subaddresses that do not have specific RT Message buffers allocated using the BusTools_RT_CbufWrite function. This function is used to write information into the default message buffer for the specified RT.

Typically, the data portion of this message buffer is not significant, but the API still copies it into the hardware buffer. The main reason that this function is called is to specify the Interrupt Enable Bits for the default message buffer. By setting the Interrupt Enable Bits properly, the application could get an interrupt anytime this message buffer is used, thereby receiving notice anytime an undefined subaddress is used.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_MessageWriteDef (cardnum, rtaddr, mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address. Valid range is 0 to 31.
mbuf	(API_RT_MBUF_WRITE *) address of structure to be transferred to the default message buffer.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_ADDR

4.154 BusTools_RT_MessageWriteStatusWord

Description

BusTools_RT_MessageWriteStatusWord stores a status word in the respective channel RT message buffer that is logically 'OR'd into the firmware generated status word. There are two status modes available for a channel starting with firmware version 5.00 and API version 6.44. These are Default Status and Extended Status. How this function affects a status word depends on the status mode. For firmware version 5.06 and thereafter, only the Extended Status mode is supported.

In Default Status mode, status is set based on the RT address. The RT transmits this Status Word in response to any messages sent to any RT subaddress. When in this mode the status value returned by this function is updated after a message to the RT address, subaddress, transmit/receive, buffer combination specified in the parameters occurs. Until then, the RT transmits the previous status. Once this change occurs all commands to the RT will return this same status.

In Extended Status mode, status is set based on RT address, subaddress, transmit/receive and buffer. Extended Status mode is set in BusTools_RT_AbufWrite and allows differing status based on subaddress, transmit/receive, and buffer. Call this function to set a unique status value for each RT/SA/TX/RX/buffer combination.

In Default Status mode, the following bits in the 1553 status word may be set using this function. The symbols listed are defined in the Busapi.h file:

- Terminal Flag – API_1553_STAT_TF
- Subsystem Flag – API_1553_STAT_SF
- Busy Bit – API_1553_STAT_BY
- Service Request – API_1553_STAT_SR
- Instrumentation – API_1553_STAT_IN

In the Extended Status mode, any status word bit other than the RT Address bits can be modified. If the Busy or Message Error status bits are set the data on transmit command is suppressed. An application must use the error injection feature to modify RT Address bits.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions, the Bus Monitor must be initialized via BusTools_BM_Init, and the Remote Terminal must be initialized via BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_MessageWriteStatusWord (cardnum, rtaddr, subaddr, tr, mbuf_id, wStatusWord, wFlag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT) RT address. Valid range is 0 to 31.
subaddr	(BT_UINT) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT) Transmit/receive flag; 0 = receive, 1 = transmit.
mbuf_id	(BT_UINT) RT Message buffer number (“0” based).
wStatusWord	(BT_UINT) New RT Status word
wFlag	(BT_UINT) modify flag; RT_NOCHANGE = do not modify the status word; RT_SET = modify the status word using the Default Status (for all firmware prior to v5.06, use this option); RT_EXT_STATUS = modify the status word using Extended Status (for all firmware v5.06 and after use this option only).

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_ILLEGAL_ADDR
API_RT_ILLEGAL_SUBADDR
API_RT_ILLEGAL_TRANREC
API_RT_ILLEGAL_MBUFID

4.155 BusTools_RT_MonitorEnable

Description

BusTools_RT_MonitorEnable enables or disables the specified RT to run in Monitor Mode. RT Monitor Mode is a mode that allows the RT to record all transactions to an RT address. An RT in monitor mode does not respond to any 1553 messages.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions, the Bus Monitor must be initialized via BusTools_BM_Init, and the Remote Terminal must be initialized via BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_MonitorEnable (cardnum, rtaddress, mode);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddress	(BT_UINT) RT address. Valid range is 0 to 31.
mode	(BT_UINT) requested mode for RT: 0 = Disable RT Monitor. 1 = Enable RT Monitor.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED

4.156 BusTools_RT_ReadLastMessage

Description

BusTools_RT_ReadLastMessage returns the last Remote Terminal message recorded in the interrupt queue that fits the criteria in the argument list. Specify an RT address, subaddress, and transmit or receive command. The RT address and subaddress input arguments are bit-encoded values. For example, to select RT0 or subaddress 0 use the LSB (0x0001). The function then searches the interrupt queue for a message matching the settings. If it does not find a matching message, the function returns API_RT_READ_NODATA.

On the initial call, this function searches the entire interrupt queue for messages. On later calls, it searches only the section between the current queue pointer and the queue pointer on the last call. For best results, you must call this function at a periodic rate such that the interrupt queue pointer does not wrap. There are 296 interrupt queue entries for firmware versions 3.x, 4.x and 5.x, while firmware 6.x has 512 interrupt queue entries.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions, the RT initialized via BusTools_RT_Init, and the RT activated via BusTools_RT_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_ReadLastMessage (cardnum, rt_addr, subaddress, tr, pRT_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rt_addr	(BT_INT) selects the RT address (0-31). This value is bit encoded, where RT0 is selected by the LSB (0x0001). –1 = don't care.
subaddress	(BT_INT) selects the sub address (0-31). This value is bit encoded, where subaddress 0 is selected by the LSB (0x0001). –1 = don't care.
tr	(BT_INT) select transmit or receive. 0 = receive; 1 = transmit; -1 = don't care.
pRT_mbuf	(API_RT_MBUF_READ *) pointer to an RT Message Buffer (read-only) structure populated if BusTools_RT_ReadLastMessage finds a matching message.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_READ_NODATA
API_RT_MBUF_NOMATCH

Notes

RT messages are recorded in the interrupt queue if the RT address / subaddress / transmit / receive combination was programmed to generate interrupts during RT setup. The RT message buffer contains the interrupt enable/disable word. Use BusTools_RT_MessageWrite to load the RT message buffer.

Example

This function hides the structure of the interrupt queue. The following code shows how to use this call.

```
API_RT_MBUF_READ mbuf;

BT_UINT ndflg;

status = BusTools_RT_ReadLastMessage(cardnum,
                                     0x10, -1, 1, &mbuf);
if(status==0)
{
   if(ndflag)
   {
      // This is new data
```

The above code returns the last transmit message to RT address 4.

4.157 BusTools_RT_ReadLastMessageBlock

Description

BusTools_RT_ReadLastMessageBlock returns all the Remote Terminal messages in the interrupt queue that fit the criteria in the argument list. You can specify a RT address, sub address, and transmit or receive command. The RT address and subaddress input arguments are bit-encoded values. For example, to select RT0 or subaddress 0 use the LSB (0x0001). The function then searches backwards in the interrupt queue for all messages matching the settings. The function returns API_RT_READ_NODATA if it does not find a matching message, or a count of the messages found.

On the initial call, this function searches the entire interrupt queue for messages. On later calls, it searches only the section between the current queue pointer and the queue pointer on the last call. For best results, you must call this function at a periodic rate such that the interrupt queue pointer does not wrap. There are 296 interrupt queue entries for firmware versions 3.x, 4.x and 5.x, while firmware 6.x has 512 interrupt queue entries.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions, the RT initialized via BusTools_RT_Init, and the RT activated via BusTools_RT_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_ReadLastMessageBlock (cardnum, rt_addr_mask, subaddr_mask, tr, mcount, pRT_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rt_addr_mask	(BT_INT) selects the RT address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
subaddr_mask	(BT_INT) selects the subaddress (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
tr	(BT_INT) select the transaction type as transmit or receive. 0 = Receive; 1 = Transmit; -1 = don’t care.
mcount	(BT_UINT *) pointer to that holds the count of messages found.

pRT_mbuf — (API_RT_MBUF_READ *) pointer to an array of RT Message Buffer structures populated if BusTools_RT_ReadLastMessageBlock finds a matching message.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_READ_NODATA
API_RT_MBUF_NOMATCH

Notes

BusTools_RT_ReadLastMessageBlock returns only Remote Terminal messages stored in the interrupt queue. RT messages are recorded in the interrupt queue if the RT address/subaddress/transmit/receive combination was programmed to generate interrupts during the RT setup. The RT message buffer contains the interrupt enable/disable word. Use BusTools_RT_MessageWrite to load the RT message buffer. Select the interrupt event options from the Interrupt Enable / Message Status Bits (32 bit) definition in Chapter 7, "Data Structures".

The application must allocate an array of API_RT_MBUF_READ structures large enough to hold all the messages found by this function. The worst case is that a call to this function returns the entire interrupt queue. In that case, allocation should provide enough allocation for API_RT_MBUF_READ structures having 296 interrupt queue entries for firmware versions 3.x, 4.x and 5.x, or 512 interrupt queue entries for firmware version 6.x or later.

Example

This function hides the structure of the interrupt queue. The following code shows how to use this call.

```
API_RT_MBUF_READ mbuf[296];
int i;
BT_UINT mess_cnt;

status = BusTools_RT_ReadLastMessageBlock(cardnum, 0x20, -1,
1, &mess_cnt, mbuf);
if(status==0)
{
   for(i=0;i<mess_cnt;i++)
   {
      // loop through all messages found
```

The above code returns all the transmit messages to RT address 5.

4.158 BusTools_RT_ReadNextMessage

Description

BusTools_RT_ReadNextMessage returns the next Remote Terminal message recorded in the interrupt queue that fits the criteria in the argument list. Specify an RT address, subaddress, and transmit or receive command. The RT address and subaddress input arguments are bit-encoded values.

This function continually polls the interrupt queue until the timeout period expires or it finds a matching message. If the function does not find a matching message and times out, it returns API_RT_READ_TIMEOUT. Time critical applications should use this function with caution.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions, the RT initialized via BusTools_RT_Init, and the RT activated via BusTools_RT_StartStop.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_ReadNextMessage (cardnum, timeout, rt_addr, subaddress, tr, pRT_mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(int) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
timeout	(BT_UINT) timeout value in milliseconds. Valid range is 10 to 65,535.
rt_addr	(BT_INT) selects the RT address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
subaddress	(BT_INT) selects the sub address (0-31) via bitwise encoded value, where RT0 is selected by the LSB (0x0001) and –1 indcates “don’t care”.
tr	(BT_INT) select the transaction type as transmit or receive. 0 = Receive; 1 = Transmit; -1 = don’t care.
pRT_mbuf	((API_RT_MBUF_READ *) pointer to an RT Message Buffer (read-only) structure populated if BusTools_RT_ReadNextMessage finds a matching message.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_MBUF_NOMATCH
API_RT_READ_TIMEOUT

Notes

RT messages are recorded in the interrupt queue if the RT address / subaddress / transmit / receive combination was programmed to generate interrupts during the RT setup. The RT message buffer contains the interrupt enable/disable word. Use BusTools_RT_MessageWrite to load the RT message buffer.

Timing accuracy differs between systems. Usually, most PC systems have accuracy no better than 10 milliseconds. You must consider the timing accuracy of your system when selecting a time-out value, especially if you are developing a deterministic application.

Example

This function hides the structure of the interrupt queue. The following code shows how to use this call.

```
API_BM_MBUF_READ mbuf;

BT_UINT timeout;

timeout = 100; // 100 millisecond timeout

status = BusTools_RT_ReadNextMessage(cardnum, timeout,
                                     0x10, -1, 1, &mbuf);
if(status==0)
{
   // Data returned
```

The above code returns the next transmit message to RT address 4.

4.159 BusTools_RT_StartStop

Description

BusTools_RT_StartStop is used to turn the RT on or off with the requested state specified by the *flag* parameter.

This function starts all enabled RTs. The RT Address buffer enables and disables individual RTs, see BusTools_RT_AbufWrite. The RT Control buffer enables and disables individual subaddresses, transmit/receive, and word count combinations. See the BusTools_RT_CbufWrite function.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions and the RT initialized using BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_RT_StartStop (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_UINT) requested state for RT: RT_STOP (0) = Stop the RT. RT_START (1) = Start the RT.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED
API_RT_NOTRUNNING
API_RT_RUNNING

4.160 BusTools_Set1553Mode

Description

BusTools_Set1553Mode provides the method for the application to select 1553A or 1553B mode protocol processing by a specific RT address. This setting applies to the Remote terminal, Bus Controller, and Bus Monitor. To mix 1553A and 1553B modes on the same bus you must initialize the channel in 1553B mode and then call this function to set which RT addresses run 1553A mode. If you initialize the channel to run 1553A mode, all RT address will respond with the 1553A protocol and they cannot be overwritten by this function.

This function uses a bit-encoded flag to set the RT address to 1553A mode. Each bit in this flag corresponds to an RT address, (e.g., bit 0 is RT0, bit 1 is RT1, etc.). For Example, to set RT 2 and RT 4 to run in 1553A mode you would pass the following:

0000 0000 0000 0000 0000 0000 0001 0100 = 0x00000014

This function must be called after BusTools/1553-API Initialization; however, after starting the Bus Controller, Remote Terminal or Bus Monitor, the application cannot change 1553 mode until those functions are terminated.

OS Support

Core API Function

Syntax

wStatus = BusTools_Set1553Mode (cardnum, rtmode);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtmode	(BT_U32BIT) bit-encoded flag indicating 1553 mode: 0 = 1553B. 1 = 1553A.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_RUNNING
API_BM_RUNNING
API_BC_RUNNING
API_HARDWARE_NOSUPPORT

4.161 BusTools_SetBroadcast

Description

BusTools_SetBroadcast is used to set the broadcast message operational mode with the Abaco Systems 1553 boards. There are two modes of operation with respect to broadcast messages:

- Broadcast Enabled: This mode supports broadcast messages. Any receive command sent to RT address 31 is treated as a message that is to be sent to all RTs. The BT1553_INT_BROADCAST bit is set in the Message Status word for any such messages. A transmit command sent to RT address 31 is treated as an illegal command.
- Broadcast Disabled: In this mode, broadcast messages are not used. RT address 31 is available to be defined like any other RT address.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions; however, BusTools_SetBroadcast must be invoked prior to initializing the RT via BusTools_RT_Init. If this function is invoked after BusTools_RT_Init, it clears the channel initialization and BusTools_RT_Init will have to be invoked again.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetBroadcast (cardnum, bcast);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
bcast	(BT_UINT) flag indicating broadcast mode: 0 = RT address 31 is not used as broadcast code. 1 = RT address 31 is used as broadcast code.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.162 BusTools_SetDumpPath

Description

BusTools_SetDumpPath sets a directory path for the memory dump-to-file feature provided via BusTools_DumpMemory, certain BIT tests, or when errors are encountered during certain operations specifically designed to generate memory dumps. This function sets the environment variable, CONDOR_HOME, with the path to the directory where memory dumps are stored. If CONDOR_HOME is NULL (not set), memory dump output is written to the working directory. This function is only required to direct memory dump output to a specific location. On Windows o/s versions after Windows 7, this function must be invoked if the application is executing in a protected directory.

There are two methods for setting CONDOR_HOME. You can globally set it using the environment variables in the Systems Properties of the Control Panel. You need administrator privileges do this. The other option is to use this function to set CONDOR_HOME locally. If globally set, dump files generated from all applications invoking BusTools_DumpMemory are written into the same directory. When locally set, each application can direct memory dump output to a different location.

UNIX systems can set CONDOR_HOME from the command line according the shell method for setting and exporting environment variables or using this function.

This function can be invoked at any time and is recommended before executing any application that incorporates BusTools_DumpMemory.

OS Support

Windows and UNIX (Requires file system support)

Syntax

wStatus = BusTools_SetDumpPath (dpath);

wStatus	(BT_INT) status returned from this function. API_BAD_PARAM indicates the path provided is not valid.
dpath	(char *) path name. On windows systems you must use the double slash (\\) as the directory separator. For Example (C:\\temp\\MyFiles)

Return Value

API_SUCCESS
API_BAD_PARAM

4.163 BusTools_SetExternalSync

Description

All Abaco Systems 1553 boards have a common counter to time-tag all messages recorded by the board. BusTools_SetExternalSync enables the external reset of that counter by a TTL input signal.

Since the common counter is global to all components simulated on the board, this option should be used carefully. For more information, see the BusTools_SetOptions function.

The API is not notified of the fact that the time-tag counter has been reset.

This function performs a subset of the BusTools_TimeTagMode function.

Prior to invoking this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetExternalSync (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_UINT) flag indicating operation: 0 = external sync is disabled 1 = external sync is enabled

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

Notes

This function is duplicated by the BusTools_TimeTagMode function as follows: For flag == 0 specify:

```
wStatus = BusTools_TimeTtagMode(cardnum, API_TT_DEFAULT,
API_TT_DEFAULT,  API_TTM_FREE, NULL, 0, 0, 0);

else, for flag == 1 specify:

wStatus = BusTools_TimeTagMode(cardnum, API_TT_DEFAULT,
API_TT_DEFAULT, API_TTM_RESET, NULL, 0, 0, 0);
```

4.164 BusTools_SetInternalBus

Description

Abaco Systems MIL-STD-1553 boards provide a feature to make it possible to transmit and receive messages internally, without requiring an external bus cable or termination. BusTools_SetInternalBus is used to set the bus operation mode of the board. The two modes of operation are:

- External Bus Enabled: In this mode of operation, the external 1553 bus is driven according to the loaded mode (BC, BM, and RTs). The BM monitors all activity on the external bus. Any bus messages, even between two locally defined components, are transmitted on the external bus. This is the default state after Initialization.
- Internal Operation Only: In this mode of operation, the external bus is ignored. All messages are simulated internally with no output to or input from the external bus.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetInternalBus (cardnum, busflag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
busflag	(BT_UINT) external or internal bus operation mode: EXTERNAL_BUS (0) = external bus enabled INTERNAL_BUS (1) = external bus disabled (internal operation)

Return Value

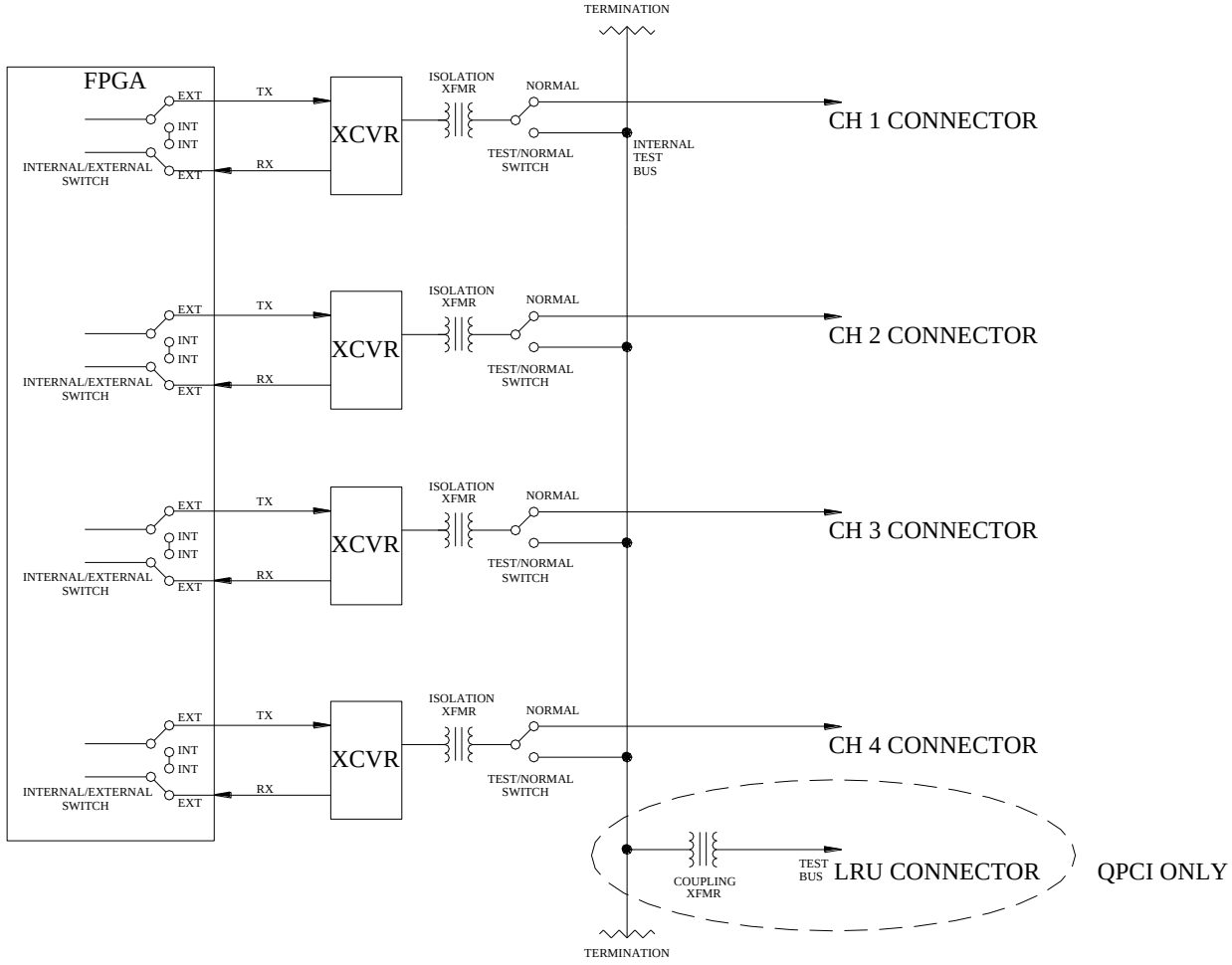
API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

Notes

The RQVME2-1553, QVME-1553, QPCI-1553, and QPCX-1553 have a Test Bus in addition to the internal and external bus options. The figure below shows how these three options operate. To use the Test Bus, select the external bus by calling BusTools_SetInternalBus and enable the Test Bus by calling BusTools_SetTestBus. The Test Bus connects the channels on multi-channel boards. You must use direct coupling with the test bus. The QPCI-1553 and QPCX-1553 has an additional LRU connector that allows you to connect to an external transformer coupled bus.

Once you select the Test Bus, the CH1 – CH4 connectors are not available. You must disable the Test Bus before you can use the connectors.

Figure 4-2 RQVME2-1553, QVME-1553, QPCI-1553, and QPCX-1553 Bus Options



4.165 BusTools_SetIntVector

Description

BusTools_SetIntVector sets the interrupt vector for VME interrupts. The vector ranges from 1 to 255. Use this function to select a vector for each channel to override the default API setting. Prior to calling this function, the channel must be initialized by invoking one of the BusTools/1553-API Initialization functions.

OS Support

VxWorks VME boards only

Syntax

wStatus = BusTools_SetIntVector (cardnum, wVector);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 15.
wVector	(BT_UINT) interrupt vector (1-255).

Return Value

API_SUCCESS
API_BAD_PARAM
API_BUSTOOLS_BADCARDNUM

4.166 BusTools_SetIRQ_Lvl

Description

BusTools_SetIRQ_Level sets the IRQ level for VME interrupts. The IRQ level ranges from 1 to 7. Use this function to select an IRQ level for each VME device to override the default API setting. There is only one IRQ per device. Each channel on that device gets the same IRQ.

This function must be called prior to initialization. This is different than most other BusTools/1553-API functions which require an initialized board. Pass the card number and IRQ value as input arguments. Call this function only once for each 1553 VME device in your system. You must be sure that the card number you use for this call matches the card number in the call to BusTools_API_InitExtended. Also, make sure that the *cardnum* you select matches a card number for the device for which you intend to set the IRQ.

OS Support

VxWorks VME boards only

Syntax

wStatus = BusTools_SetIRQ_Level (wDevice, pwIRQ);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 15.
pwIRQ	(BT_UINT) pointer to array of 16 IRQ values ranging from 1-7.

Return Value

API_SUCCES
API_BAD_PARAM
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_INITED

4.167 BusTools_SetMultipleExtTrig

Description

BusTools_SetMultipleExtTrig enables external triggers for each channel on the RXMC-1553, RXMC2-1553 and R15-LPCIE boards.

For the RXMC-1553, the possible number of trigger outputs depends on the board configuration. The table below lists the different configuration options and the output lines available for triggers.

Configuration	PIO	Discrete	EIA485
PIO_OPN_GRN	8	4	-
PIO_28V_OPN	8	4	-
DIS_OPN_GRN	-	12	-
DIS_28V_OPN	-	12	-
EIA485_OPN_GRN	-	4	4
EIA485_28V_OPN	-	4	4

The RXMC-1553 can route the output trigger to any PIO, discrete, or 485 output. The RXMC2-1553 and R15-LPCIE can route the output trigger to any discrete output. The application can configure multiple output triggers for a channel and channels can share the same output trigger.

Call this function any time after the application initializes the channel. The PIO, Discrete, and EIA-485 lines are shared resources. Your application must coordinate the use of these channels so not to interfere with application use of other channels on the same board.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetMultipleExtTrig (cardnum, trigOpt, tvalue, enableFlag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
trigOpt	(BT_INT) Select trigger type PIO, DISCRETE, or EIA485 (for R15-LPCIE and RXMC2-1553 only DISCRETE is used).
tvalue	(BT_UINT) trigger channel. Valid range is 1 to 12.
enableFlag	(BT_INT) EXT_TRIG_ENABLE; EXT_TRIG_DISABLE

Return Value

API_SUCCES
API_BAD_PARAM
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_INITED

4.168 BusTools_SetNRLRTimeout

Description

BusTools_SetNRLRTimeout sets the time-out values for late response and no-response. The late response time is the time allowed after the BC transmits a message before declaring the RT response late. The response must occur between the late response time and the no-response time. If this happens, then late response bit in the Interrupt Status Word is set. If the response exceeds the no-response time then, even if an RT does respond, the no-response bit in the Interrupt Status Word is set.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetNRLRTimeout (cardnum, wTimeout1, wTimeout2);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
wTimeout1	(BT_UINT) timeout period for the “No Response” error, in µs. Valid range is 4 to 31.
wTimeout2	(BT_UINT) time-out period for the “Late Response” error, in µs. Valid range is 4 to 31.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.169 BusTools_SetOptions

Description

BusTools_SetOptions supports the following channel configuration options:

- Suppressing the Minor-Frame overflow warning
- Memory Dump on BM stop
- BM Trigger on Message
- Monitor Invalid Commands
- Reset TT on synchronize Mode Code
- Trigger on synchronize Mode Code
- RT start on external sync
- Ignore High Word errors
- Undefined Mode Codes are Illegal

Prior to invoking this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions. The Bus Monitor must be initialized using the BusTools_BM_Init function.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetOptions (cardnum, intflag, resettimer, trig_on_sync, enable_rt);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
intflag	(BT_UINT) 0x0001 - Suppress MF OFlow message 0x0002 - Monitor invalid commands 0x0004 - Dump on BM Stop 0x0008 - BM trigger on message 0x0010 - Ignore High Word errors 0x0040 - Undefined Mode Code illegal
resettimer	(BT_UINT) “Reset Time tag on Sync” option: 0 = Disabled. 1 = Enabled.
trig_on_sync	(BT_UINT) Trigger output on Sync Mode Code 0 = Disabled 1 = Enabled

enable_rt (BT_UINT) RT start on trigger input
0 = Disabled
1 = Enabled.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_RT_NOTINITED

4.170 BusTools_SetPolling

Description

BusTools/1553-API supports a software-polling mode. This mode is set up when you initialize using either Software or Hardware interrupt mode. Hardware Only mode is the only mode that does not set up polling. The default-polling interval is 10 milliseconds. BusTools_SetPolling allows you to change the default polling rate.

BusTools_SetPolling can be invoked any time after a channel is initialized. The polling interval is set using a timer assigned to the application from the operating system. There is only one timer per application, so modifying the polling interval for one channel it will modify the interval for all channels controlled by the application.

The lowest timing interval is one millisecond. The accuracy of the timer is a function of the operating system. Setting a timing interval does not guarantee that the operating system meets that interval in all cases. This is particularly true for non-real-time operating system like Windows and UNIX. Your application may need to set operating system timing parameters to achieve the desired timing.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetPolling (polling);

wStatus	(BT_INT) status returned from this function.
polling	(BT_UINT) The polling interval in milliseconds (1 – 2000).

Return Value

API_SUCCESS
API_NO_POLLING
BTD_TIMER_FAIL

4.171 BusTools_SetSa31

Description

All Abaco Systems 1553 boards can support two RT function modes of operation with respect to subaddress 31 and mode codes. BusTools_SetSa31 provides a method to select which mode to use. The two modes of operation are:

- Subaddress 31 is not used for mode codes.
- Subaddress 31 is used for mode codes.

This function must be invoked after initializing the channel with one of the BusTools/1553-API Initialization functions but before invoking BusTools_RT_Init.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetSa31 (cardnum, sa31);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
sa31	(BT_UINT) flag indicating subaddress 31 operation mode: 0 = subaddress 31 is not used for mode codes 1 = subaddress 31 is used for mode codes

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.172 BusTools_SetTermEnable

Description

The RXMC2-1553 and R15-LPCIE have two differential discrete I/O lines supporting switchable termination. BusTools_SetTermEnable provides the method to enable or disable 120-Ω termination on these discrete channels. Setting bits 0 and 1 of the *tEnable* parameter enable termination. If those bits are cleared, termination is disabled.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetTermEnable (cardnum, tEnable);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
tEnable	(BT_U16BIT) enable or disable termination. Valid range is 0 to 3.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.173 BusTools_SetTestBus

Description

The QVME-1553, RQVME2-1553 and QPCX-1553 boards support a 1553 Test Bus. This function enables the Test Bus mode of the Abaco Systems 1553 board. The Test Bus allows communication between different channels on a multi-channel board. When you enable the Test Bus, the firmware routes the 1553 signal to an onboard bus connecting the different channels. This differs from setting the internal bus in that the internal bus setting allows only communication within a single channel. The Test Bus also differs from the internal bus in that the internal bus connects before the transformer and transceiver, while the test bus connects after these components. Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

The advantage of this design is that you test all components in the 1553 path, and you can connect two independent 1553 channels without any connectors, couplers, tees, and terminators.

When the Test Bus is enabled, the 1553 signal is switched from the external connector (front or rear panel) to the onboard Test Bus. You must select direct coupling in BusTools_BC_Init to use the Test Bus between two channels on a multi-channel board. To return the 1553 signal to the external connectors, disable the Test Bus.

The QPCI-1553 and QPCX-1553 have an additional LRU connector that allows you to connect to an external transformer coupled bus.

The two modes of operation are:

- **Test Bus Enabled**: In this mode, the 1553 output for the selected channel is switched onto the onboard Test Bus.
- **Test Bus Disabled**: In this mode, the 1553 output for the selected channel is switched to the external connector.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetTestBus (cardnum, busflag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
busflag	(BT_UINT) Test bus operation mode: TEST_BUS_ENABLE = Test bus enabled TEST_BUS_DISABLE = Test bus disabled

Return Value

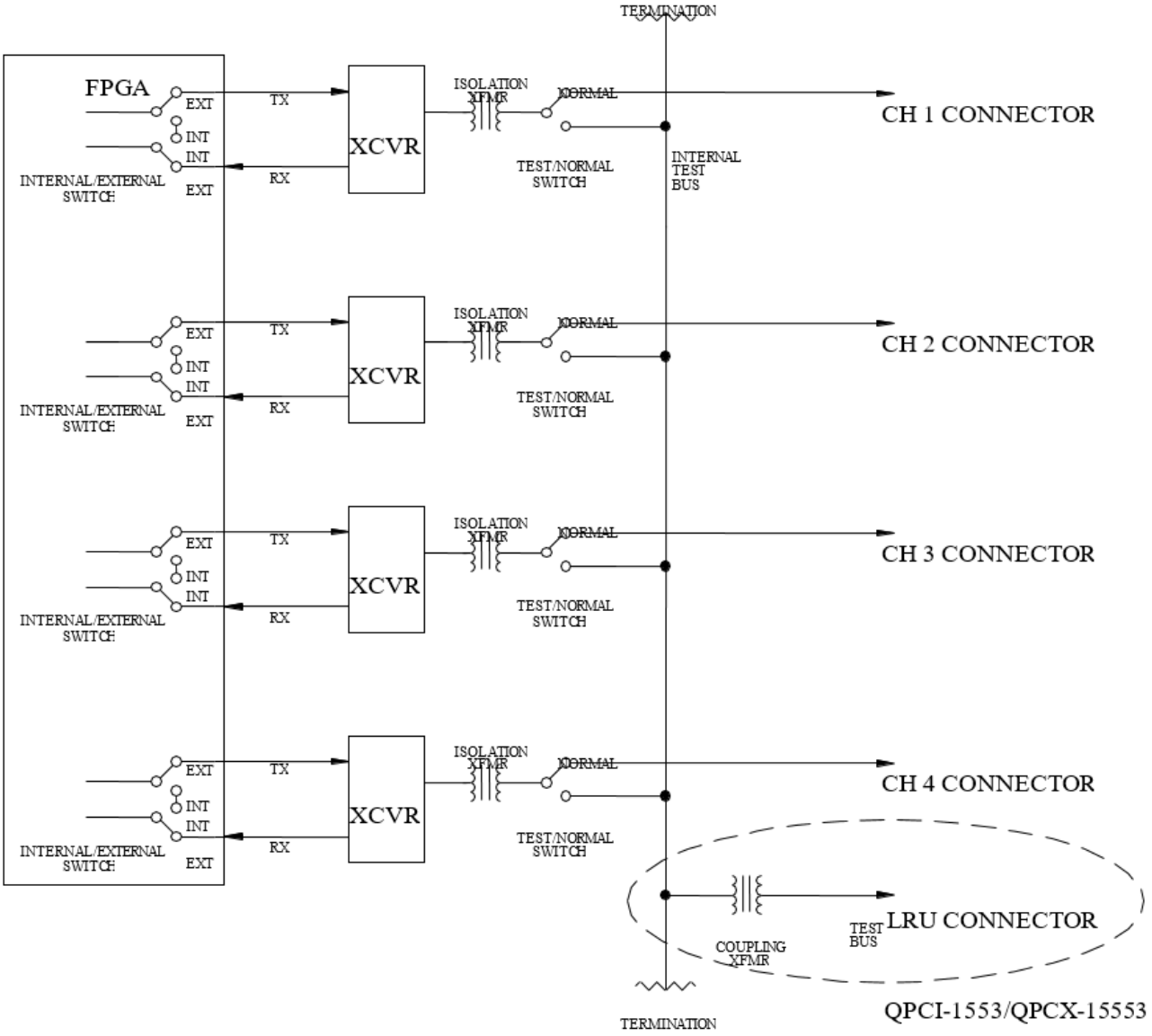
API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

Note

To use the Test Bus, select the external bus by calling BusTools_SetInternalBus and enable the Test Bus by calling BusTools_SetTestBus. The Test Bus connects the channels on multi-channel boards. You must use direct coupling with the Test Bus.

Once you select the test bus, the CH1 – CH4 connectors are not available. You must disable the test bus before you can use the connectors.

Figure 4-3 RQVME2-1553, QVME-1553, QPCI-1553 & QPCX-1553 Bus Options



4.174 BusTools_SetVoltage

Description

The Abaco Systems MIL-STD-1553 boards can drive the 1553 bus with direct coupling or transformer coupling. In addition, variable voltage boards can adjust transmit voltage levels. This function is used to set these parameters.

For direct coupling, the legal voltage range is 0.0V to 6.5V. For transformer coupling, the legal voltage range is 0.0V to 19.8V. For both types of coupling, the available resolution is approximately 0.1V.

The value of voltage passed to this function is an *integer*, which is the value of the voltage in volts multiplied by 100.

If coupling is set to DAC_VALUE the coupling is left unchanged and the voltage value set between 0 and 255 is written directly to the DAC.

This API function assumes the boards are configured to support variable voltage output. It responds to API calls for Output level, regardless of hardware. Calls made to fixed amplitude boards have no effect on the board's transmit level.

Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function

Syntax

wStatus = BusTools_SetVoltage (cardnum, voltage, voltflag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
voltage	(BT_UINT) output voltage -- in volts*100
voltflag	(BT_UINT) flag indicating coupling: DIRECT (0) = direct coupling TRANSFORMER (1) = transformer coupling DAC_VALUE (255) = DAC to wVoltage (0-255)

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BUSTOOLS_BADCOUPLING
API_BUSTOOLS_BADVOLTAGE
API_HARDWARE_NOSUPPORT

4.175 BusTools_SetV6TrigIn

Description

This function selects a discrete to use as an external input trigger for a channel on a board running with F/W Version 6.0 or greater. The logical channel referenced via *cardnum* uses the selected discrete as an input trigger.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_SetV6TrigIn (cardnum, trigOpt, tvalue);

cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
trigOpt	(BT_U16BIT) Trigger Option: DISCRETE (1) EIA485 (2) PIO (3)
tvalue	(BT_UINT) trigger channel (1-18). Select a valid trigger channel for the board in use.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.176 BusTools_SetV6TrigOut

Description

This function selects a discrete to use as an output trigger for a channel on a board running with F/W Version 6.0 or greater. The logical channel referenced via *cardnum* uses the selected discrete as an output trigger.

OS Support

Core API Function (F/W Version 6.0 or greater)

Syntax

wStatus = BusTools_SetV6TrigOut (cardnum, trigOpt, tvalue);

cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
trigOpt	(BT_U16BIT) Trigger Option: DISCRETE (1) EIA485 (2) PIO (3)
tvalue	(BT_UINT) trigger channel (1-18). select a valid trigger channel for the board in use.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.177 BusTools_StatusGetString

Description

This function is used to convert a BusTools API error number into an ASCII string suitable for display to a user. The string informs you that there was an unexpected condition with respect to the BusTools hardware and/or software. See Chapter 6, "Return Codes", for a complete listing of all error codes.

All error codes returned by the BusTools/1553-API are supported by this function. For the most recent list, see the current file "Busapi.h".

OS Support

Core API Function

Syntax

pString = BusTools_StatusGetString (status);

pString	(char *) returned address of ASCII string. This string is a "static" string in the BusTools API functions – just the address is returned – don't attempt to modify this string.
status	(BT_INT) BusTools API status code from some function.

Return Value

Pointer to a string containing the ASCII definition of the *status* parameter.

4.178 BusTools_TimeGetString

Description

BusTools_TimeGetString is used to convert a BusTools/1553-API time structure into a string suitable for display. The time structure is contained in all Bus Controller, Bus Monitor and Remote Terminal messages.

The TTDisplay parameter passed to the BusTools_TimeTagMode function defines how time tags are converted by this function as follows:

Table 4-4 TTDisplay Parameter Settings

TTDisplay	Display Type:
API_TTD_IRIG* API_TTD_IRIG_NS**	IRIG Format "(ddd)hh:mm:ss.uuuuuu". Supported by all board variants. Formatting: ddd = number of days if required; hh = number of hours if required; mm = minutes (0 – 59); ss = seconds (0 – 59); uuuuuu = microseconds (000000 – 999999). All parameters are displayed, even if zero.
API_TTD_DATE* API_TTD_DATE_NS**	Date Format "(MM/dd)hh:mm:ss.uuuuuu" where MM = month (1 - 12) dd = day of month (1 – 31)
API_TTD_RELM* API_TTD_RELM_NS**	Relative to midnight format "(ddd)hh:mm:ss.uuuuuu". Only the needed parameters are displayed (leading zero parameters are suppressed).

* String conversion based on µs lsb
**String conversion based on ns lsb

The BusTools API Time Tag Counter "BT1553_TIME" structure has a 1-µs resolution and a range of 407.23 days for V5 and earlier boards.

Starting with BusTools/1553-API version 8.0 this function uses a 64-bit time tag. In addition, the time tag can have either a micro- or nano-second resolution. If you are converting a nanosecond (ns) time tag you must set the pString parameter to "NANO" prior to calling this function or use the _NS display format when calling BusTools_TimeTagMode.

OS Support

Core API Function (Not support by VxWorks 5.4 or earlier).

Syntax

BusTools_TimeGetString (curtime, string);

curtime	(BT1553_TIME *) pointer to a time structure
string	(char*) pointer to output string (should be at least 25 characters long, depending on the selected format, and the current value of the time tag). If you pass a string set to

"NANO" the string conversion expects the time value to have a resolution of 1 nanosecond.

Return Value

None.

Notes

The format of the conversion is set by the previous call to the BusTools_TimeTagMode function. Unlike the other parameters set by the BusTools_TimeTagMode function, the conversion mode is a global parameter affecting all open boards, it is not retained on a per-board basis.

4.179 BusTools_TimeGetFmtString

Description

This function converts a BusTools time structure into a string suitable for display. The BusTools time structure is contained in all BC, BM, and RT messages. This function allows the user to pass the display format. If API_TTD_DEFAULT is used, the current global format setting is used.

Table 4-5 TTDisplay Parameter Settings

TTDisplay	Display Type:
API_TTD_IRIG API_TTD_IRIG_NS	IRIG Format "(ddd)hh:mm:ss.uuuuuu". Supported by all board variants. Formatting: ddd = number of days if required; hh = number of hours if required; mm = minutes (0 – 59); ss = seconds (0 – 59); uuuuuu = microseconds (000000 – 999999). All parameters are displayed, even if zero.
API_TTD_DATE API_TTD_DATE_NS	Date Format "(MM/dd)hh:mm:ss.uuuuuu" where MM = month (1 - 12) dd = day of month (1 – 31)
API_TTD_RELM API_TTD_RELM_NS	Relative to midnight format "(ddd)hh:mm:ss.uuuuuu". Only the needed parameters are displayed (leading zero parameters are suppressed).

The BusTools API Time Tag Counter "BT1553_TIME" structure has a 1-µs resolution and a range of 407.23 days. Starting with BusTools/1553-API version 8.0 this function takes a 64-bit time tag. In addition, the time tag can have either a micro- or nanosecond resolution. If you are converting a nanosecond time tag use the _NS display format.

OS Support

Core API Function (Not support by VxWorks 5.4 or earlier).

Syntax

BusTools_TimeGetFmtString (tFormat, curtime, string);

tFormat	(BT_INT) Display format API_TDD_RELM, API_TTD_RELM_NS, API_TTD_IRIG, API_TTD_IRIG_NS, API_TTD_DATE, API_TTD_DATE_NS, API_TTD_DEFAULT.
curtime	(BT1553_TIME*) pointer to time structure.
string	(char*) pointer to output string (should be at least 25 characters long, depending on selected format, and value of the time tag).

Return Value

None.

4.180 BusTools_TimeTagGet

Description

This function is *not* a part of the BusTools/1553-API; rather it is a user-written function in a user-provided Dynamic Link Library (DLL).

This function (and its containing DLL) is only used when the built-in time-tag initialization function supplied by the API is insufficient. This provides a mechanism for extending the API to support those cases.

The API loads the DLL containing this function when the BusTools_TimeTagMode function is called, and the parameter "TTInit" is set to "API_TTI_EXT". The API calls this function when the Time Tag Counter is to be initialized. The exact function definition is given in the function prototype contained in the Busapi.h file.

This function must compute and return the value to be loaded into the Time Tag Counter whenever it is called.

Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

OS Support

Windows.

Syntax

wStatus = BusTools_TimeTagGet (cardnum, pTime);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
pTime	(BT1553_TIME *) pointer to time structure

Return Value

API_SUCCESS if no errors were detected.

Notes

This function can return any error code desired if a failure is detected. The API returns the error code API_TIMETAG_USER_ERROR if a non-zero return code is detected from this function.

4.181 BusTools_TimeTagInit

Description

BusTools_TimeTagInit initializes both the hardware and the software time-tag counters. All board types support this function call.

The API calls this function in BusTools_TimeTagMode to initialize the Time Tag Counter. A user program can also call this function at any time to initialize the Time Tag Counter. BusTools API uses a Time Tag Counter with a 1-µs resolution. The 45-bit time tag can record up to 407.23 days.

The exact behavior of this function depends on the underlying board type and the Time Tag Counter initialization mode specified by the TTInit argument in the most recent call to BusTools_TimeTagMode.

If you set TTInit to API_TTI_ZERO or API_TT_DEFAULT, or you have not set the Time tag mode by calling BusTools_TimeTagMode, then calling this function clears the Time Tag Counter. The API also sets the base time to zero. This is the default behavior for the Time Tag Counter.

If you set TTInit to API_TTI_DAY, this function reads the current time of day, relative to midnight, from the host clock and loads the value into the Time Tag Counter.

If you set TTInit to API_TTI_IRIG, the function reads the current time of year from the host clock and loads the value into the Time Tag Counter.

If you set TTInit to API_TTI_EXT, this function calls the user-function BusTools_TimeTagGet in the specified user DLL. The function loads the returned value into the hardware Time Tag Counter.

Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function.

Syntax

wStatus = BusTools_TimeTagInit (cardnum);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED

4.182 BusTools_TimeTagMode

Description

BusTools/1553-API has many time tag modes (see tables below). These modes define the time-tag format in messages the Bus Monitor records, and the Remote Terminal function sends. It also defines the active time-tag display format.

Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

All Abaco 1553 boards programmed with version 4/5 firmware base the onboard timer on a 1-MHz clock, with the time-tag counter defined as a 45-bit counter. All Abaco 1553 boards programmed with version 6 firmware support a time-tag counter defined as 64-bit counter having a resolution of 1ns. This counter is used in all time tagging of messages recorded by the Bus Monitor or transacted by the Remote Terminal function.

The "Reset Time tag on Sync" option also controls the same time tag. Enabling this option sets the hardware Time Tag Counter to a value in the Time Tag Counter Load Register whenever the board receives a Synchronize Mode Code. Since the time tag is global to all components, use this option carefully. For more information, see the BusTools_SetOptions function.

The API is not notified on time tag reset, and when the Time Tag Counter overflows. It uses this notification information to extend, in software, the Time Tag Counter to 48 bits. A 48-bit time tag, counting a 1-MHz clock, does not wrap around for 3257.81 days.

The TTDisplay parameter defines the way time tags appear by the BusTools_TimeGetString function as follows:

Table 4-6 TTDisplay parameter for BusTools_TimeGetString

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

This parameter is global to all boards being controlled by the current instance of the API. The other mode parameters apply on a per-board basis (or a per-channel basis, for multi-channel boards). Use the _NS parameter on V6 boards that have 64-bit ns time tags.

For further information about the format of the time tag ASCII string, see the function BusTools_TimeGetString.

The table below defines how the API initializes the Time Tag Counter.

Table 4-7 TTinit Values

TTInit	Time tag Initialization Mode:
API_TT_DEFAULT	Unchanged from previous call.
API_TTI_ZERO	Time tag initialized to zero. Supported by all board variants. (Default) This happens at the call to BusTools_TimeTagInit. Time that elapses between this call and the start of traffic will be reflected in the time tag reported in the messages.
API_TTI_DAY	Time of day, relative to midnight, is loaded into the Time Tag Counter, when Bus Monitor Started (Host Clock reference)
API_TTI_IRIG	Time of year (IRIG format) (Host clock reference)
API_TTI_EXT	External time reference (provided by the user-supplied function “BusTools_TimeTagGet” in the DLL specified by “DLLname”)

TTMode specifies how the Time Tag Counter operates as follows:

Table 4-8 TTMode Values

TTMode	Time Tag Counter operating mode:
API_TT_DEFAULT	Unchanged from previous call.
API_TTM_FREE	Free running Time Tag Counter, supported by all board variants (Default).
API_TTM_RESET	Time Tag Counter reset to zero on external TTL input discrete active. Supported by all board variants.
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 (see BusTools_TimeTagWrite) by external TTL input pulse.
API_TTM_IRIG	Time Tag Counter is reset to the IRIG time from either an external or 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 1-MHz 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.

OS Support

Core API Function.

Syntax

```
wStatus = BusTools_TimeTagMode ( cardnum, TTDisplay, TTInit, TTMode,
          DLLname, TTPeriod, lParm1, lParm2 );
```

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
TTDisplay	(BT_UINT) time tag display format.
TTInit	(BT_UINT) time tag initialization mode.
TTMode	(BT_UINT) flag indicating time tag mode.
DLLname	(char *) name of the DLL containing an application-defined time-tag retrieval function.
TTPeriod	(BT_U32BIT) the duration to increment time when used in conjunction with the external TTL sync input, defined with a 1-µs resolution.
lParm1	(BT_U32BIT) spare parameter.
lParm2	(BT_U32BIT) This flag is used to enable or disable the invocation of BusTools_TimeTagInit within execution of the function BusTools_TimeTagMode. A value of zero enables invocation of BusTools_TimeTagInit. Any other value disables that invocation.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT
API_NO_OS_SUPPORT
API_TIMETAG_BAD_DISPLAY
API_TIMETAG_BAD_INIT
API_TIMETAG_BAD_MODE
API_TIMETAG_NO_DLL
API_TIMETAG_NO_FUNCTION
API_TIMETAG_USER_ERROR

Notes

If the parameter *lParm2* is non-zero, the invocation of BusTools_TimeTagInit will not occur within execution of BusTools_TimeTagMode; in this case the application must invoke BusTools_TimeTagInit before invoking BusTools_BM_StartStop or BusTools_RT_StartStop.

The firmware reads the hardware Time Tag Counter and the value saved in the message buffer at the beginning of each message recorded by the Bus Monitor, Remote Terminal, and Bus Controller.

4.183 BusTools_TimeTagRead

Description

BusTools_TimeTagRead reads the current value of the hardware Time Tag Counter on all current Abaco Systems MIL-STD-1553 boards. The definition of the Time Tag Counter differs between F/W version 4/5 and F/W version 6.0.

For F/W version 4/5 the Time Tag Counter is a 45-bit counter with a resolution of 1-µs. For F/W version 6.x the Time Tag Counter is a 44-bit counter with a resolution of 1ns.

Each channel on a 1553 board contains an independent Time Tag Counter on each channel (e.g., two Time Tag Counters on dual channel boards). The operation of the Time Tag Counter is specified via invocation of the BusTools_TimeTagMode function.

Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function.

Syntax

wStatus = BusTools_TimeTagRead (cardnum, timetag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
timetag	(BT1553_TIME *) pointer to time structure

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.184 BusTools_TimeTagReset

Description

This function enables an external reset of the time-tag counter to zero (0). This is either a 48-bit counter for V5 and earlier firmware or a 64-bit counter for V6 firmware. The Time Tag Counter operates in the mode specified by the BusTools_TimeTagMode function. This function requires firmware version 5.00 or higher.

Each board has a Time Tag Counter per channel. When enabled by this function, an external pulse resets the current contents of the Time Tag Counter. Check the MIL-STD-1553 Hardware Installation Manual for the external reset input pin.

Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function.

Syntax

wStatus = BusTools_TimeTagReset (cardnum, tflag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
tflag	(BT_UINT) reset flag: EXT_RESET_ENABLE (1) EXT_RESET_DISABLE (0)

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.185 BusTools_TimeTagWrite

Description

BusTools_TimeTagWrite writes the specified value to the 45-bit hardware Time Tag Counter on all current Abaco Systems 1553 boards. The value is written to a time-tag load register, after which the firmware loads the contents of the load register into the Time Tag Counter. The value in the load register is preserved and may be re-loaded into the Time Tag Counter again (when enabled) by a positive-going transition on the discrete input line. See BusTools_TimeTagMod for more information.

Each of these boards provides one Time Tag Counter per channel (e.g., two Time Tag Counters on dual channel boards). Each Time Tag Counter has an associated time tag load register.

The BT1553_TIME structure is different depending on the firmware version. For F/W version 5.x or before the size and resolution are 48-bits and microseconds (although only 45 bits are used). For F/W version 6.0 with BusTools/1553-API version 8.0 and later, the size and resolution are 64-bits and nanoseconds.

Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function.

Syntax

wStatus = BusTools_TimeTagWrite (cardnum, timetag, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
timetag	(BT1553_TIME *) pointer to time structure
flag	(BT_INT) if zero the value is written to the time tag holding register only. If one, the value is also transferred into the Time Tag Counter.

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_HARDWARE_NOSUPPORT

4.186 BusTools_UpdateIntFifo

Description

BusTools_UpdateIntFifo stores the contents of the local API_INT_FIFO into the global API_INT_FIFO.

This function is used for C# wrapper compatibility only and is not needed for any other application.

OS Support

Windows C# compatibility.

Syntax

void = BusTools_UpdateIntFifo (LPVOID uFifo, LPVOID nFifo);

uFiFo	(LPVOID)pointer to global API_INT_FIFO structure.
nFifo	(LPVOID)pointer to local API_INT_FIFO structure.

Return Value

None

4.187 BusTools_UpdateTailPTR

Description

BusTools_UpdateTailPTR updates the API_INT_FIFO tail pointer.

OS Support

Windows C# compatibility.

This function is used for C# wrapper compatibility only and is not needed for any other application.

Syntax

void = BusTools_UpdateTailPTR (LPVOID uFifo, BT_INT tail);

uFiFo	(LPVOID)pointer to API_INT_FIFO structure.
nFifo	(BT_INT)value for the tail pointer.

Return Value

none

4.188 BusTools_WriteVMEConfig

Description

BusTools_WriteVMEConfig writes data to a selected offset in the VME A16 configuration space on the RQVME2-1553 and QVME-1553. This allows you to write the Interrupt vectors to the vector addresses for each channel. It also allows diagnostic testing.

Use care when calling this function not to overwrite any configuration setting. See the *MIL-STD-1553 Universal Core Architecture Manual* for a full description of these Configuration Registers.

Prior to calling this function, the channel must be initialized using one of the BusTools/1553-API Initialization functions.

OS Support

Core API Function.

Syntax

wStatus = BusTools_WriteVMEConfig (cardnum, offset, vdata);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
offset	(BT_UINT) Offset to A16 base address.
vdata	(BT_U16BIT) data written to offset

Return Value

API_SUCCESS
API_HARDWARE_NOSUPPORT

5 • Extending the API

5.1 Introduction

The BusTools/1553-API is a powerful programming tool, providing access to all features of the BusTools/1553 MIL-STD-1553 interface boards. The API provides board-independent programming at a high level, freeing the programmer from many details like memory management that would consume a large portion of your programming effort.

However, there are cases when the API may not meet your exact needs. There may be cases where the API's Bus Controller memory allocation scheme does not provide the required capability, even though the underlying hardware is capable of providing the buffering. Alternatively, when using the API through the BusTools/1553 GUI interface you find that you need additional capabilities.

Before the implementation of the User DLL Interface, your only options were to:

- Rewrite the API to suit your needs or
- Write the entire application, including the complete user interface, to implement a possibly very small function that was not provided by BusTools/1553-API.

5.2 BusTools/1553-API User DLL Interface

The Abaco Systems BusTools/1553-API now includes a User DLL Interface. Use this interface to extend and enhance the API, without re-writing the current API or creating your own GUI interface. The standard BusTools/1553 GUI (available as a separate product) can now use applications that include your own custom code.

5.3 How Does it Work?

The user DLL interface executes the user-supplied functions when calls to specified API function are made. The return value from the user supplied function controls whether the API function is bypassed or executes after the user-supplied function. Enabling the user DLL feature requires the application to load the user specified DLL by calling "BusTools_API_LoadUserDLL."

Call the API function BusTools_API_LoadUserDLL to load a specified DLL. If the API finds the DLL, it is loaded, and the API looks to see if any function name within the DLL matches the user DLL entry points. If the API finds any of these functions, the API saves their addresses in a table.

There is a one-to-one relationship between API functions and User Interface DLL functions. For example, the API function BusTools_BC_MessageWrite calls the user interface function UsrBC_MessageWrite, if it is defined, at the beginning of the API function. The user interface function can perform any action it wishes, it then

returns a code that indicates whether the API function should continue normal operation or return to the calling function immediately. It can also indicate to the API that it should never be called again (useful for one-time initialization code).

The table below shows the user DLL entry and associated API functions:

Table 5-1 User DLL Entry and Associated API Functions

User DLL Entry	Associated API Function
UsrAPI_Close	BusTools_API_Close
UsrBC_MessageAlloc	BusTools_BC_MessageAlloc
UsrBC_MessageWrite	BusTools_BC_MessageWrite
UsrBC_MessageRead	BusTools_BC_MessageRead
UsrBC_MessageUpdate	BusTools_BC_MessageUpdate
UsrBC_StartStop	BusTools_BC_StartStop
UsrRT_CbufWrite	BusTools_RT_CbufWrite
UsrRT_MessageRead	BusTools_RT_MessageRead
UsrRT_StartStop	BusTools_RT_StartStop
UsrBM_MessageAlloc	BusTools_BM_MessageAlloc
UsrBM_MessageRead	BusTools_BM_MessageRead
UsrBM_StartStop	BusTools_BM_StartStop

5.3.1 Support for Multiple User Interface DLLs

The API repeats this process whenever BusTools_API_LoadUserDLL is called. This function extracts the addresses of any user DLL entry points from the DLL and adds them to the table of function addresses.

For any given user function there is only one slot in the API address table, so the table retains the address of the last function found. This means that if you define a function in two user DLLs, the API only calls the function from the last DLL you load.

When a user application calls an API function, the API then calls the corresponding user interface functions. There are now “hooks” into the application by the user interface DLLs, without needing to modify either the application or the API.

5.4 What Can I Do from a User Interface DLL Function?

The API calls User Interface DLL functions at the beginning of the corresponding API function. Here are some of the things a user interface function can do:

- It can modify the arguments to that function and return to allow the function to execute normally with the new arguments.
- It can perform the action itself, including the use of API functions, and then return to the calling function without executing the API function.
- It can perform some other function, perhaps not related to the API function, then return to the API to allow it to complete the requested action.

5.5 User Interface DLL Function Example

Suppose that you are using the BusTools/1553 GUI to control a PCI-1553 board that is acting as the Bus Controller. You are using the Bus Monitor function to record the 1553 data, and the various features of the BM to display the data while the bus is running.

The Bus Controller list consists of several messages that retrieve data from an external RT that you are trying to test. Those messages are static in nature but repetitive; you are reading out the position data from the navigational system and recording the drift over time.

However, there is one small problem. This device requires a message from the BC containing a "stale data" indicator. This message word changes every time the BC transfers the data to the Navigation system.

Therefore, what would normally be a simple BusTools/1553 GUI setup now requires programming.

Look at the actual code needed to implement this function using the User Interface DLL. To reduce the size of the listing the function headers have been removed, but the full source of this example is provided on the software distribution disks as "btuser1.c":

```
/*=============================================================================================*
 * User API ENTRY POINT:  U s r B C _ M e s s a g e W r i t e
 *=============================================================================================*
 * FUNCTION:    This User Interface function is called when BusTools_BC_MessageWrite is called.
 * DESCRIPTION: This function intercepts all BusTools_BC_MessageWrite function calls and
 *    clears the Interrupt Enable on all BC messages except the last message in the minor frame.
 * It returns API_CONTINUE       - API function should continue execution normally
 *=============================================================================================*/
NOMANGLE BT_INT CCONV UsrBC_MessageWrite
(
   BT_UINT cardnum,          // (i) card number (0 - based)
   BT_UINT *messno,          // (i) index of BC message to write to
   API_BC_MBUF *api_message  // (i) pointer to user-specified BC message
)
{
   // Determine if this message is the one that triggers the data update. If so, set the interrupt
   // enable bit, otherwise clear the interrupt enable.  We will trigger off of the first message in the
   // minor frame.  Since this is the message that we update, triggering on it gives us the maximum time
   // to update the message before it is transmitted again.
```

```
   api_message->control &= ~BC_CONTROL_INTERRUPT;    // Turn off the interrupt

   // Not a message?
   if ( (api_message->control & BC_CONTROL_TYPEMASK) != BC_CONTROL_MESSAGE )
      return API_CONTINUE;      // Have the API function continue normally

   // Not the beginning of the frame?
   if ( (api_message->control & BC_CONTROL_MFRAME_BEG) == 0 )
      return API_CONTINUE;      // Have the API function continue normally
   api_message->control |= BC_CONTROL_INTERRUPT;    // Turn on the interrupt
   return API_CONTINUE;      // Have the API function continue normally
}

/*=============================================================================================*
 * User ENTRY POINT demo_bc_watch_function
 *=============================================================================================*
 * FUNCTION:    This register function is called whenever a BC message interrupt is detected.
 *=============================================================================================*/
BT_INT _stdcall demo_bc_watch_function
(
   BT_UINT cardnum,
   struct api_int_fifo *sIntFIFO
)
{
   /*********************************************************************
   *  Local variables
   *********************************************************************/
   BT_U16BIT   data[33];       // Data buffer we update
   BT_INT      tail;           // FIFO Tail index
   BT_UINT     messno;         // Message number to be updated

   /*********************************************************************
   *  Loop on all entries in the FIFO.  Get the tail pointer and extract
   *   the FIFO entry it points to.   When head == tail, FIFO is empty
   *********************************************************************/
   tail = sIntFIFO->tail_index;
   while (tail != sIntFIFO->head_index )
   {
      // Extract the buffer ID from the FIFO and read the message from the board
      messno = sIntFIFO->fifo[tail].bufferID;
      BusTools_BC_MessageReadData(cardnum, messno, data);

      // sample data buffer update per some algorithm:
      data[0] = (BT_U16BIT)(data[0] + data[1]);
      data[2] += (short)1;

      // Now write the data back to the message buffer:
      BusTools_BC_MessageUpdate(cardnum, messno, data);

      // Now update and store the tail pointer.
      tail++;                          // Next entry
      tail &= sIntFIFO->mask_index;    // Wrap the index
      sIntFIFO->tail_index = tail;     // Save the index
   }
   return API_SUCCESS;
}

/*=============================================================================================*
 * User API ENTRY POINT U s r B C _ S t a r t S t o p
 *=============================================================================================*
 * FUNCTION:    This User Interface function is called whenever BusTools_BC_StartStop is called.
 * DESCRIPTION: This function is called by the API whenever the BC is started or stopped.
 * This function sets up a BusTools_RegisterFunction that monitors the BC for the first
 * message in the minor frame.   When that message occurs the thread runs and modifies the data
 *  in the BC message.
 * It will return API_CONTINUE       - API function should continue execution normally
 *=============================================================================================*/
static API_INT_FIFO  sIntFIFO1;     // Thread FIFO structure

NOMANGLE BT_INT CCONV UsrBC_StartStop
(
   BT_UINT cardnum,        // (i) card number (0 - based)
   BT_UINT *flag           // (i) 1 -> start BC (at message 0), 0 -> stop BC
)
{
   int    rt, tr, sa;
```

```
   /*********************************************************************
   *  If the BC is starting, register a thread for the board.
   *  If the BC is shutting down, unregister the thread.
   *********************************************************************/
   if ( *flag )
   {
      // Setup the FIFO structure for this board.
      memset(&sIntFIFO1, 0, sizeof(sIntFIFO1));
      sIntFIFO1.function     = demo_bc_watch_function;
      sIntFIFO1.iPriority    = THREAD_PRIORITY_ABOVE_NORMAL;
      sIntFIFO1.dwMilliseconds = INFINITE;
      sIntFIFO1.iNotification  = 0;       // Dont care about startup or shutdown
      sIntFIFO1.FilterType     = EVENT_BC_MESSAGE;
      for ( rt=0; rt < 32; rt++ )
         for (tr = 0; tr < 2; tr++ )
            for (sa = 0; sa < 32; sa++ )
               sIntFIFO1.FilterMask[rt][tr][sa] = 0xFFFFFFFF;  // Enable all messages

      // Call the register function to register and start the BC thread.
      BusTools_RegisterFunction(cardnum, &sIntFIFO1, 1);
   }
   else
   {
      // Call the register function to unregister and stop the BC thread.
      BusTools_RegisterFunction(cardnum, &sIntFIFO1, 0);
   }
   return API_CONTINUE;      // Have the API function continue normally
}
```

Compile the above code with the Windows DLL interface functions (provided in the file “bt-wep.c”) and link the resulting object files with the BusTools/1553-API library to create a user interface DLL.

Copy the resulting DLL (named “btuser1.dll” in the examples supplied with the API) into the folder from which you are running BusTools/1553 (or into the system directory) then run BusTools normally. From the Card Setup window, enter the name of the user DLL interface. The user interface code is automatically called, and when you monitor the BC data, you see the data being changed per the code in the “demo_bc_watch_function” above.

5.6 BusTools/1553-API User DLL Interface Functions

The current version of the BusTools/1553-API contains user interfaces for the following API functions:

- BusTools_API_Close calls **UsrAPI_Close**
- BusTools_BC_MessageAlloc calls **UsrBC_MessageAlloc**
- BusTools_BC_MessageRead calls **UsrBC_MessageRead**
- BusTools_BC_MessageUpdate calls **UsrBC_MessageUpdate**
- BusTools_BC_MessageWrite calls **UsrBC_MessageWrite**
- BusTools_BC_StartStop calls **UsrBC_StartStop**
- BusTools_BM_MessageAlloc calls **UsrBM_MessageAlloc**
- BusTools_BM_MessageRead calls **UsrBM_MessageRead**
- BusTools_BM_StartStop calls **UsrBM_StartStop**
- BusTools_RT_CbufWrite calls **UsrRT_CbufWrite**
- BusTools_RT_MessageRead calls **UsrRT_MessageRead**
- BusTools_RT_StartStop calls **UsrRT_StartStop**

These are the first functions that have been implemented; other functions will be added in the future. These interfaces provide the ability for your code, written as a stand-alone DLL, to interface with the API functions, and actually change the way these API functions operate.

The following section gives the exact interface to each of these user interface functions. You must code the interface exactly as given, so that the API can correctly call the function. The function prototypes are listed in the BUSAPI.H file included with the BusTools/1553-API software.

5.6.1 UsrAPI_Close

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_API_Close is called.

The board has not been closed when this user interface function is called. It does not matter what status code the User Interface function returns, the API closes the board once the function returns.

OS Support

Windows.

Syntax

wStatus = UsrAPI_Close (cardnum);

wStatus (BT_INT) status returned from this function.

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.2 UsrBC_MessageAlloc

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_BC_MessageAlloc is called.

The BC messages have not been allocated when this function is called.

OS Support

Windows.

Syntax

wStatus = UsrBC_MessageAlloc (cardnum, count);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
count	(BT_UINT *) number of BC messages to allocate

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.3 UsrBC_MessageRead

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_BC_MessageRead is called.

The BC message has not been yet read when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrBC_MessageRead (cardnum, messno, api_message);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messno	(BT_UINT *) index of BC message to read
api_message	(API_BC_MBUF *) buffer to read message into

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.4 UsrBC_MessageUpdate

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_BC_MessageUpdate is called.

The BC message has not been changed when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrBC_MessageUpdate (cardnum, mblock_id, buffer);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mblock_id	(BT_UINT *) index of BC message to update
buffer	(BT_U16BIT *) array of data for BC message

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.5 UsrBC_MessageWrite

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_BC_MessageWrite is called.

The BC message has not been altered when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrBC_MessageWrite (cardnum, messno, api_message);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
messno	(BT_UINT *) message number of message to alter
api_message	(API_BC_MBUF *) buffer containing new BC message

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.6 UsrBC_StartStop

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_BC_StartStop is called.

The run state of the BC has not been altered when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrBC_StartStop (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_UINT *) 1 = Start BC (at message 0), 0 = Stop BC

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.7 UsrBM_MessageAlloc

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_BM_MessageAlloc is called.

The no BM messages have been allocated when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrBM_MessageAlloc (cardnum, mbuf_count, mbuf_actual, enable);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mbuf_count	(BT_UINT *) requested number of BM messages
mbuf_actual	(BT_UINT *) number of BM messages allocated
enable	(BT_U32BIT *) BM interrupt enable mask

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.8 UsrBM_MessageRead

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_BM_MessageRead is called.

The BM message has not been read when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrBM_MessageRead (cardnum, mbuf_id, mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
mbuf_id	(BT_UINT *) ID number of the BM message to read
mbuf	(BT_U32BIT *) BM message buffer

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.9 UsrBM_StartStop

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_BM_StartStop is called.

The run state of the bus monitor had not been changed when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrBM_StartStop (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_UINT *) 1 = Start BM 0 = Stop BM

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.10 UsrRT_CbufWrite

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_RT_CbufWrite is called.

The RT Cbuf has not been altered when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrRT_CbufWrite (cardnum, rtaddr, subaddr, tr, mbuf_count, apicbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT *) RT address. Valid range is 0 to 31.
subaddr	(BT_UINT *) RT subaddress. Valid range is 0 to 31.
tr	(BT_UINT *) Transmit/Receive flag; 0 = receive, 1 = transmit.
mbuf_count	(BT_UINT *) Number of buffers to allocate. If negative, make one pass through buffers only
apicbuf	(API_RT_CBUF *) pointer to API RT control buffer

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.11 UsrRT_MessageRead

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_RT_MessageRead is called.

The RT message buffer has not been read when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrRT_MessageRead (cardnum, rtaddr, subaddr, tr, mbuf_id, mbuf);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
rtaddr	(BT_UINT *) RT address (0 - based)
subaddr	(BT_UINT *) RT subaddress (0 - based)
tr	(BT_UINT *) Transmit/Receive flag (1 = rt transmit)
mbuf_id	(BT_UINT *) RT message buffer to read
mbuf	(API_RT_CBUF *) pointer to RT message buffer

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

5.6.12 UsrRT_StartStop

Description

This function is not a part of the API; you provide this function in a User Interface DLL. The API calls this function when the API function BusTools_RT_StartStop is called.

The run status of the RT has not been altered when this user interface function is called.

OS Support

Windows.

Syntax

wStatus = UsrRT_StartStop (cardnum, flag);

wStatus	(BT_INT) status returned from this function.
cardnum	(BT_UINT) logical channel reference to the respective 1553 board/channel session. Valid range is 0 to 63.
flag	(BT_UINT *) 1 = Start RT 0 = Stop RT

Return Value

API_CONTINUE API function should continue execution normally.

API_RETURN_SUCCESS API function should return immediately with API_SUCCESS.

API_NEVER_CALL_AGAIN User function is never to be called again.

6 • Return Codes

This chapter lists the BusTools/1553-API return codes and provides a description for each code. Return codes are listed with both the numeric value and the mnemonic for that code. The manual provides the numeric as a quick reference. However, when programming an application, use the mnemonics instead of the actual numeric value. The numeric value can change between revisions.

Use BusTools_StatusGetString or BusTools_StatusGetStringLV to convert any of the BusTools/1553-API return code mnemonics to an ASCII string that describes the return condition. The include file busapi.h defines the return codes.

Most BusTools/1553-API functions have return codes. API_SUCCESS (0), indicates an error free execution of the function. A non-zero return code indicates an error or event has occurred that the user should know about. Each return code has with severity information. The return-code severity levels are:

CEI-INFO – This code type informs the user of the status of the operation and does not mean that an error occurred. The return code has information the user may need to know. An example of this type of return code is API_BM_READ_NODATA; "CEI-INFO -- No BM data in interrupt queue"; This error is returned on a call to BusTools_BM_ReadLastMessageblock. The information that there was no data in the interrupt queue is not an error condition, rather provides the the user's application with information about the status of BM messages in the interrupt queue. The application can act on this information according the it's requirements.

CEI-WARN – This code means there is a problem with parameters passed to a BusTools/1553-API function or there is problem in performing that function. The user should examine board configuration or parameters passed to make sure they are correct. An example of this type of return code is API_BUSTOOLS_NOTINITED; "CEI_WARN -- BusTools API has not been initialized". This code is return when an application attempts to call a BusTools function without first initializing the channel. The user should examine their code to make sure initialize the board prior to calling most BusTools/1553-API functions.

CEI-ERROR – This code is return when the execution of a function results in and error. This a problem where execution of the code results in an error. Bad software, hardware, or setup can cause this type of error. An example of this type of error condition is API_BC_HALTERROR; "CEI-ERROR -- BC error detected during stop, bus is probably un-terminated". This error occurs when shutting down the BC with BusTools_BC-StartStop. This error may be due to un-terminated bus or a hardware problem.

CEI-CRIT-ERR – This type of code means a critical error occurred during execution and that the 1553 operation have failed. An example of this type of error is API_HW_IQPTR_ERROR; "CEI-CRIT-ERROR -- Hardware Interrupt Pointer register

error". The interrupt point register contains the current position of the interrupt queue pointer. This value must lie within the bound of the interrupt queue. If the API detects a value outside this range, then this error is reported. This indicates a serious hardware problem. This error is also reported in the channel_status structure err_info member.

List of Return Codes

The following table lists all API and API driver return codes. The API return codes are grouped into General, BC, BM, RT, Error Injection, LabView, Playback, Time Tag and Low-level categories. The list is arranged numerically.

Table 6-1 Return Codes List

Return Code Mnemonic	Value	Return Code String
Low Level Return Codes		
BTD_OK	0	CEI-INFO – success
BTD_ERR_PARAM	1	CEI-WARN -- invalid parameter
BTD_ERR_NOACCESS	2	CEI-ERROR -- unable to map/access adapter
BTD_ERR_INUSE	3	CEI-WARN -- adapter already in use
BTD_ERR_BADADDR	4	CEI-ERROR -- invalid address
BTD_ERR_NODETECT	5	CEI-ERROR -- I/O or Config register ID invalid, board detect fail
BTD_ERR_NOTSETUP	7	CEI_ERROR -- adapter has not been setup
BTD_ERR_FPGALOAD	8	CEI-ERROR -- FPGA load failure
BTD_ERR_NOMEMORY	10	CEI-ERROR -- error allocating memory in SW version
BTD_ERR_BADADDRMAP	12	CEI-ERROR -- bad initial mapping of address
BTD_ERR_BADEXTMEM	13	CEI-ERROR -- bad extended memory mapping
BTD_ERR_BADBOARDTYPE	14	CEI-ERROR -- Unknown board type
BTD_ERR_BADWCS	15	CEI-ERROR -- Verify failure reading Writable Control Store
BTD_NO_PLATFORM	18	CEI-ERROR -- Platform specified unknown or not supported
BTD_BAD_MANUFACTURER	19	CEI-ERROR -- IP ID PROM Manufacturer code not 0x79
BTD_BAD_MODEL	20	CEI-ERROR -- IP ID PROM Model number not 0x05(MF) or 0x08(SF)
BTD_BAD_SERIAL_PROM	21	CEI-ERROR -- IP Serial PROM needs update, no support for this version
BTD_NEW_SERIAL_PROM	22	CEI-ERROR -- Serial PROM too new, not supported by this software
BTD_CHAN_NOT_PRESENT	23	CEI-WARN -- Channel not present (on multi-channel board)
BTD_NON_SUPPORT	24	CEI-WARN -- Bus/Carrier/OS combination not supported by API
BTD_BAD_HW_INTERRUPT	25	CEI-ERROR -- Hardware interrupt number bad or not defined in registry
BTD_FPGA_NOT_CLEAR	26	CEI-ERROR -- The FPGA configuration failed to clear
BTD_NEW_PCCARD_FW	27	CEI-ERROR -- PCC-1553 firmware is too new for this version of the API
BTD_OLD_PCCARD_FW	28	CEI-ERROR -- PCC-1553 firmware is too old, use the JAM Player to update it
BTD_BAD_CONF_FILE	29	CEI-ERROR -- Unable to open ceidev.conf
BTD_NO_DRV_MOD	30	CEI-ERROR -- No Driver Module found

Return Code Mnemonic	Value	Return Code String
BTD_IOCTL_DEV_ERR	31	CEI-ERROR -- Error in ioctl get device
BTD_IOCTL_SET_REG	32	CEI-ERROR -- Error in ioctl set region
BTD_IOCTL_REG_SIZE	33	CEI-ERROR -- Error in getting ioctl region size
BTD_IOCTL_GET_REG	34	CEI-ERROR -- Error in ioctl get region
BTD_BAD_SIZE	35	CEI-ERROR -- Region size is 0
BTD_BAD_PROC_ID	36	CEI-INFO -- Unable to set process ID
BTD_HASH_ERR	37	CEI-INFO -- Unable to setup hash table
BTD_NO_HASH_ENTRY	38	CEI-INFO -- No hash table entry found
BTD_WRONG_BOARD	39	CEI-INFO -- Wrong board type for command
BTD_MODE_MISMATCH	40	CEI-INFO -- IPD1553 mismatch in the mode.
BTD_IRIG_NO_LOW_PEAK	41	CEI-INFO -- No lower peak on IRIG DAC calibration
BTD_IRIG_NO_HIGH_PEAK	42	CEI-INFO -- No upper peak on IRIG DAC calibration
BTD_IRIG_LEVEL_ERR	43	CEI-WARN -- Delta between MAX and MIN DAC peak values less than required
BTD_IRIG_NO_SIGNAL	44	CEI-INFO -- No IRIG Signal Detected
BTD_RTADDR_PARITY	45	CEI-ERROR -- Parity Error on Hardwired RT address lines
BTD_BAD_BYTE_COUNT	47	CEI-ERROR -- Byte count not on 4-byte boundary
BTD_TIMER_FAIL	48	CEI-ERROR -- failed to create a timer
BTD_ERR_NOWINRT	50	CEI-ERROR -- WinRT driver not loaded/started
BTD_ERR_BADREGISTER	51	CEI-ERROR -- WinRT parameters don't match registry
BTD_ERR_BADOPEN	52	CEI-ERROR -- WinRT device open failed
BTD_UNKNOWN_BUS	53	CEI-ERROR -- Bus is not PCI, ISA or VME
BTD_BAD_LL_VERSION	54	CEI-ERROR -- Unsupported lowlevel driver installed
BTD_BAD_INT_EVENT	55	CEI-ERROR -- Unable to create interrupt event
BTD_ISR_SETUP_ERROR	56	CEI-ERROR -- Error setting up the ISR driver
BTD_CREATE_ISR_THREAD	57	CEI-ERROR -- Error creating the ISR thread
BTD_NO_REGIONS_TO_MAP	58	CEI-ERROR -- No regions requested in call to vbtMapBoardAddresses
BTD_RESOURCE_ERR	60	CEI-ERROR -- Integrity Resource Error
BTD_READ_IODEV_ERR	61	CEI-ERROR -- Integrity I/O Device Read Error
BTD_MEMREG_ERR	62	CEI-ERROR -- Integrity error getting memory region
BTD_MEM_MAP_ERR	63	CEI-ERROR -- Integrity Memory Mapping error
BTD_CLK_RATE_NOT_SET	64	CEI-ERROR -- Error setting clock rate
BTD_VIOPEN_FAIL	70	CEI-ERROR -- viOpen Error
BTD_VIMAPADDRESS_FAIL	71	CEI-ERROR -- viMapAddress Error
BTD_VIOPENDEFAULTRM	72	CEI-ERROR -- viOpenDefaultRM Error
BTD_VIUNMAP_ERR	73	CEI-ERROR -- viUnMapAddress Error
BTD_SEM_CREATE	80	Error Creating semaphore
BTD_TASK_CREATE	81	Error spawning task

Return Code Mnemonic	Value	Return Code String
BTD_EVENT_WAIT_FAILED	-82	CEI-ERROR -- Event Wait Failure
BTD_EVENT_WAIT_ABANDONED	-83	CEI-ERROR -- Event Wait Abandoned
BTD_EVENT_WAIT_TIMEOUT	84	CEI-INFO -- Timeout on Event Wait
BTD_EVENT_WAIT_UNKNOWN	-85	CEI-ERROR -- Unknown Event Error
BTD_EVENT_SIGNAL_ERR	86	CEI-ERROR -- Error Occurred During Event Signal
BTD_SET_PRIORITY_ERR	87	CEI-ERROR -- Error Setting Thread Priority
BTD_THRD_CREATE_FAIL	88	CEI-ERROR -- Thread Create Failure
BTD_CLOSE_ERR	90	CEI-ERROR -- Failed to close 1553 device
BTD_OPEN_ERR	91	CEI-ERROR -- Failed to open 1553 device
BTD_VBT_OPEN_ERR	92	CEI-ERROR -- Failure in vbtOpen1553Channel
BTD_FIND_DEV_ERR	93	CEI_ERROR -- Failure in BusTools_FindDevice
BTD_LIST_DEV_ERR	94	CEI_ERROR -- Failure in BusTools_ListDevices
General BusTools/1553-API Return Codes		
API_SUCCESS	0	CEI-INFO -- No error detected
PI_FEATURE_SUPPORT	120	CEI-INFO -- Feature supported by board
API_CONTINUE	121	CEI-INFO -- API function should continue execution normally
API_RETURN_SUCCESS	122	CEI-INFO -- API function should return immediately with API_SUCCESS
API_NEVER_CALL_AGAIN	123	CEI-INFO -- User function is never to be called again
API_INIT_NO_SUPPORT	124	CEI-WARN -- Cannot initialize board type with this function
API_NO_CHANNEL_MAP	125	CEI_WARN -- Channel mapping not supported for current card type
API_BUSTOOLS_INT_USED	170	CEI-WARN -- Interrupt on card already in use
API_NULL_PTR	171	CEI-WARN -- NULL Pointer passed to function
API_MAX_CHANNELS_INUSE	180	CEI-WARN -- Maximum 1553 channels already in use
API_CARDNUM_INUSE	181	CEI-WARN -- cardnum in already in use
API_BAD_PRODUCT_LIST	182	CEI-WARN -- Unable to build the Abaco Systems Product list
API_BAD_DEVICE_ID	183	CEI-WARN -- Bad device ID
API_INSTALL_INIT_FAIL	184	CEI-ERROR -- CEI_INSTALL init failure
API_NO_POLLING	190	CEI-WARN -- Polling is not enabled
API_TIMER_ERR	191	CEI-ERROR -- Error setting up polling timer
API_BUSTOOLS_INITED	201	CEI-WARN -- This card has already been init'ed
API_BUSTOOLS_NOTINITED	202	CEI-WARN -- BusTools API not initialized
API_BUSTOOLS_BADCARDNUM	203	CEI-WARN -- Bad card number specified
API_BUSTOOLS_BADCOUPLING	206	CEI-WARN -- Bad coupling specified in BusTools_SetVoltage
API_BUSTOOLS_BADVOLTAGE	207	CEI-WARN -- Bad voltage specified in BusTools_SetVoltage
API_BUSTOOLS_EVENBCOUNT	209	CEI_ERROR -- Even byte count required for this function
API_BUSTOOLS_BADMEMORY	210	CEI_ERROR -- BusTools Board Dual-Port Memory Self-Test Failed
API_BUSTOOLS_TOO_MANY	211	CEI-WARN -- Too many user interrupt functions registered

Return Code Mnemonic	Value	Return Code String
API_BUSTOOLS_FIFO_BAD	212	CEI_ERROR -- User API_INT_FIFO structure corrupted or bad entry
API_BUSTOOLS_NO_OBJECT	213	CEI_ERROR -- Error creating event object or thread
API_BUSTOOLS_NO_FILE	215	CEI-WARN -- Could not open the specified file
API_BUSTOOLS_NO_MEMORY	216	CEI-WARN -- BusTools_MemoryAlloc request overflows first 64 Kw of board memory
API_HW_IQPTR_ERROR	217	CEI-CRIT-ERR -- Hardware Interrupt Pointer register error.
API_BIT_BC_RT_FAIL_PRI	218	CEI-ERROR -- BIT failure/data error detected on BC-RT primary bus
API_BIT_BC_RT_FAIL_SEC	219	CEI-ERROR -- BIT failure/data error detected on BC-RT secondary bus
API_BUSTOOLS_FIFO_DUP	220	CEI-ERROR -- Specified API_INT_FIFO structure is already in use.V4.35.ajh
API_BIT_BM_RT_FAIL_PRI	221	CEI-ERROR -- BIT failure/data error detected on BM-RT primary bus
API_BIT_BM_RT_FAIL_SEC	222	CEI-ERROR -- BIT failure/data error detected on BM-RT secondary bus
API_HARDWARE_NOSUPPORT	225	CEI-WARN -- Function not supported by current hardware
API_OUTDATED_FIRMWARE	226	CEI-WARN -- Firmware version no longer supported, contact factory for upgrade
API_NO_OS_SUPPORT	227	CEI-WARN -- Function not supported by underlying Operating System
API_NO_BUILD_SUPPORT	228	CEI-WARN -- Function not supported by API as built
API_CHANNEL_OPEN_OTHER	229	CEI-WARN -- Board or channel already opened as another cardnum
API_SINGLE_FUNCTION_ERR	231	CEI-WARN -- Attempted to start multiple functions on a single function board
API_CANT_LOAD_USER_DLL	232	CEI-WARN -- Cannot load specified user DLL
API_REGISTERFUNCTION_OFF	233	CEI-WARN -- RegisterFunction operations not enabled
API_BAD_PARAM	240	CEI-WARN -- Bad parameter for the function call
Error Injection Return Codes		
API_EI_BADMSGTYPE	252	CEI-WARN -- Bad message type specified in EbufWrite
API_EI_ILLERRORNO	253	CEI-WARN -- Error injection buffer num > number of buffers avail
API_EI_ILLERRORADDR	254	CEI-WARN -- Illegal error buffer address
Discrete and Data Conversion Return Codes		
API_BAD_ADDR_TYPE	271	CEI-WARN -- Bad address type for BusTools_GetAddr
API_BAD_DISCRETE	280	CEI-WARN -- Attempting to configure invalid discrete
API_OUTPUT_DISCRETE	281	CEI-WARN -- Attempting to read from an output
API_INPUT_DISCRETE	282	CEI-WARN -- Attempting to write to an input
API_MEM_ALLOC_ERR	283	CEI-ERROR -- Error allocating memory
API_BADDATATYPE	284	CEI-WARN -- Bad data type for EU conversion.
API_BADBCDDATA	285	CEI-WARN -- Bad data for BCD EU conversion.
API_BADTRANSLATE	286	CEI-WARN -- Bad translation table data, for translate EU conversion.
API_BADFACTORTYPE	287	CEI-WARN -- Bad factor type for scaled EU conversion.
Bus Controller Return Codes		
API_BC_NOTINITED	301	CEI-WARN – BusTools_BC_Init not yet called
API_BC_INITED	302	CEI-WARN – BusTools_BC_Init already called

Return Code Mnemonic	Value	Return Code String
API_BC_RUNNING	303	CEI-WARN -- BC currently running
API_BC_NOTRUNNING	304	CEI-WARN -- BC not currently running
API_BC_MEMORY_OFLOW	305	CEI-ERROR -- BC memory overflow
API_BC_ILLEGAL_MBLOCK	306	CEI-WARN -- BC illegal memory block number specified
API_BC_MBLOCK_NOMATCH	307	CEI-WARN -- BC specified address is not a BC message block
API_BC_MBUF_NOT_ALLOC	308	CEI-WARN -- BC message buffers have not been allocated
API_BC_MBUF_ALLOCD	309	CEI-WARN -- BC message buffers already allocated
API_BC_ILLEGAL_NEXT	310	CEI-WARN -- BC illegal next message number
API_BC_ILLEGAL_PREV	311	CEI-WARN -- BC illegal previous message number
API_BC_ILLEGAL_BRANCH	312	CEI-WARN -- BC illegal branch message number
API_BC_MESS1_COND	313	CEI-WARN -- BC first message in buffer is conditional
API_BC_BAD_COND_ADDR	314	CEI-WARN -- BC bad address value in conditional message
API_BC_BADTIMEOUT1	315	CEI-WARN -- BC illegal "No Response" timeout
API_BC_BADTIMEOUT2	316	CEI-WARN -- BC illegal "Late Response" timeout
API_BC_BADFREQUENCY	317	CEI-WARN -- BC illegal minor frame frequency
API_BC_HALTERROR	318	CEI-ERROR -- BC error detected during stop, bus is probably un-terminated
API_BC_BOTHBUFFERS	323	CEI-WARN -- BC cannot specify both buffers
API_BC_BOTHBUSES	324	CEI-WARN -- BC cannot specify both buses
API_BC_UPDATEMESSTYPE	326	CEI-WARN -- BC message update must operate on a message (not branch)
API_BC_ILLEGALMESSAGE	327	CEI-ERROR -- BC message in memory is not legal
API_BC_ILLEGALTARGET	328	CEI-WARN -- BC branch data message number not legal
API_BC_NOTMESSAGE	329	CEI-WARN -- BC message is not a proper 1553-type message
API_BC_NOTNOOP	330	CEI-WARN -- BC message is not a proper noop-type message
API_BC_APERIODIC_RUNNING	331	CEI-WARN -- BC Aperiodics still running, cannot start new msg list
API_BC_APERIODIC_TIMEOUT	332	CEI-ERROR -- BC Aperiodic messages did not complete in time
API_BC_CANT_NOOP	333	CEI-WARN -- BC cannot noop or un-noop a noop message
API_BC_READ_TIMEOUT	335	CEI-INFO -- RT timeout when attempting to read data.
API_BC_READ_NODATA	336	CEI-INFO -- No RT data in int queue
API_BC_AUTOINC_INUSE	337	CEI-WARN -- Auto-Increment in use for message
Bus Monitor Return Codes		
API_BM_NOTINITED	401	CEI-WARN – BusTools_BM_Init or BusTools_BM_MessageAlloc not called
API_BM_INITED	402	CEI-WARN – BusTools_BM_Init already called
API_BM_RUNNING	403	CEI-WARN -- BM currently running
API_BM_NOTRUNNING	404	CEI-WARN -- BM not currently running
API_BM_MEMORY_OFLOW	405	CEI-ERROR – BM memory overflow
API_BM_ILLEGAL_ADDR	408	CEI-WARN -- BM illegal RT address specified
API_BM_ILLEGAL_SUBADDR	409	CEI-WARN -- BM illegal subaddress specified

Return Code Mnemonic	Value	Return Code String
API_BM_ILLEGAL_TRANREC	410	CEI-WARN -- BM illegal transmit/receive flag specified
API_BM_ILLEGAL_MBUFID	411	CEI-WARN -- BM illegal mbuf_id for specified subunit
API_BM_MBUF_NOMATCH	412	CEI-ERROR -- BM no match for specified address
API_BM_WRAP_AROUND	413	CEI-ERROR -- BM API message buffer has overflowed, data has been lost
API_BM_MSG_ALLOC_CALLED	415	CEI-WARN – BusTools_BM_MessageAlloc has already been called
API_BM_HW_WRAP_AROUND	416	CEI-ERROR -- BM HW message buffer has overflowed, data has been lost
API_BM_POINTER_REG_BAD	417	CEI-ERROR -- BM HW pointer register contents invalid
API_BM_READ_TIMEOUT	418	CEI-INFO -- BM timeout when attempting to read data.
API_BM_READ_NODATA	419	CEI-INFO -- No BM data in int queue
API_BM_1760_ERROR	420	CEI-INFO -- Checksum error on MIL-STD-1760 message.
Remote Terminal Return Codes		
API_RT_NOTINITED	501	CEI-WARN – BusTools_RT_Init not yet called
API_RT_INITED	502	CEI-WARN -- BusTools_RT_Init already called
API_RT_RUNNING	503	CEI-WARN -- RT currently running
API_RT_NOTRUNNING	504	CEI-WARN -- RT not currently running
API_RT_MEMORY_OFLOW	505	CEI-ERROR -- RT memory overflow
API_RT_CBUF_EXISTS	506	CEI-ERROR -- RT subunit MBUFs already allocated
API_RT_ILLEGAL_ADDR	508	CEI-WARN -- RT illegal address specified
API_RT_ILLEGAL_SUBADDR	509	CEI-WARN -- RT illegal subaddress specified
API_RT_ILLEGAL_TRANREC	510	CEI-WARN -- RT illegal transmit/receive flag specified
API_RT_ILLEGAL_MBUFID	511	CEI-WARN -- RT illegal mbuf_id for specified subunit
API_RT_CBUF_BROAD	513	CEI-WARN -- RT 31 is broadcast only
API_RT_CBUF_NOTBROAD	514	CEI-WARN -- specified rt address is non-bro only
API_RT_MBUF_NOMATCH	515	CEI-WARN -- RT message buffer not found at specified address
API_RT_BROADCAST_DISABLE	516	CEI-WARN -- RT 31 Broadcast is disabled
API_RT_SELF_TEST_MODE	517	CEI-WARN -- RT Self Test Wrap-Around Mode selected, normal operation inhibited
API_RT_READ_TIMEOUT	518	CEI-INFO -- RT timeout when attempting to read data.
API_RT_READ_NODATA	519	CEI-INFO -- No RT data in interrupt queue
API_NO_HARDWIRE_RT	520	CEI-INFO -- RT hardwired address not enabled.
LabView Return Codes		
API_LV_BADARRAY	700	CEI-WARN -- LabView array structure not correctly setup
API_NO_LV_SUPPORT	701	CEI-WARN -- Function not supported in LabView
Playback Return Codes		
API_PLAYBACK_INIT_ERROR	801	CEI-ERROR -- Error initializing Playback
API_PLAYBACK_BAD_THREAD	802	CEI-ERROR -- Attempt to create thread failed
API_PLAYBACK_BAD_FILE	803	CEI-ERROR -- File open failed

Return Code Mnemonic	Value	Return Code String
API_PLAYBACK_BAD_EVENT	804	CEI-ERROR -- Event creation error
API_PLAYBACK_BUF_EMPTY	805	CEI-ERROR -- Playback Buffer empty
API_PLAYBACK_BAD_EXIT	806	CEI-ERROR -- Unexpected Exit from Playback
API_PLAYBACK_BAD_MEMORY	807	CEI-ERROR -- Unable to allocate memory on Host
API_PLAYBACK_DISK_READ	808	CEI-ERROR -- Disk read Error during playback
API_PLAYBACK_RUNNING	809	CEI-WARN -- Playback is already running
API_PLAYBACK_BAD_ALLOC	810	CEI-ERROR -- Failure to allocate enough BusTools Memory for PB
API_PLAYBACK_TIME_GAP	811	CEI-INFO -- There larger gaps in time tags in playback file.
API_PLAYBACK_TIME_ORDER	812	CEI-ERROR -- Time tags in playback file out of sequence.
Time Tag and IRIG Return Codes		
API_TIMETAG_BAD_DISPLAY	901	CEI-WARN -- Unknown or unsupported Time Tag display format
API_TIMETAG_BAD_INIT	902	CEI-WARN -- Unknown Time Tag Initialization method
API_TIMETAG_BAD_MODE	903	CEI-WARN -- Unknown Time Tag Operating Mode
API_TIMETAG_NO_DLL	904	CEI-WARN -- DLL containing BusTools_TimeTagGet could not be loaded
API_TIMETAG_NO_FUNCTION	905	CEI-WARN -- Could not get the address of the BusTools_TimeTagGet function
API_TIMETAG_USER_ERROR	906	CEI-WARN -- User function BusTools_TimeTagGet returned an error
API_TIMETAG_WRITE_ERROR	907	CEI-ERROR -- Cannot write to time tag load register when in API_TM_IRIG mode
API_IRIG_NO_SIGNAL	908	CEI-INFO -- No external IRIG signal present

7 • Data Structures

This section describes the data structures and other definitions used throughout the BusTools API. For each structure or topic, a high-level description and a detailed definition of each element is provided.

The topics discussed in this section are:

- 1553 Command Word (BT1553_COMMAND)
- 1553 Status Word (BT1553_STATUS)
- BC Retry Parameters (BusTools_BC_Init argument)
- BC Message Buffer (API_BC_MBUF)
- BM Filter Buffer (API_BM_CBUF)
- BM Message Buffer (API_BM_MBUF)
- BM Trigger Buffer (API_BM_TBUF)
- BM Word Status Bits (8/16 bit)
- Error Injection Definition (API_EIBUF)
- Interrupt Enable / Message Status Bits (32 bit)
- Interrupt Queue Message Block Structure (iq_mblock)
- Interrupt Register/Filter/FIFO Structure
- Playback Data (API_PLAYBACK)
- Playback Status (API_PLAYBACK_STATUS)
- RT Address Control Block (API_RT_ABUF)
- RT Control Buffer (API_RT_CBUF)
- RT Control Buffer for Broadcast (API_RT_CBUFBROAD)
- RT Message Buffer (read-only) (API_RT_MBUF_READ)
- RT Message Buffer (write-only) (API_RT_MBUF_WRITE)
- Time Structure (BT1553_TIME)
- Device Mapping (DEVMAP_T)
- Device Information (DEVICE_INFO)

7.1 1553 Command Word (BT1553_COMMAND)

Code Definition

```
typedef struct bt1553_command
       {
       #ifdef NON_INTEL_BIT_FIELDS
          BT_U16BIT rtaddr:5;     // rt address field              (MSB)
          BT_U16BIT tran_rec:1;   // transmit/receive bit
          BT_U16BIT subaddr:5;    // subaddress field
          BT_U16BIT wcount:5;     // word count or mode code field (LSB)
       #else  /* INTEL-Compatible bit field ordering */
          BT_U16BIT wcount:5;     // word count or mode code field (LSB)
          BT_U16BIT subaddr:5;    // subaddress field
          BT_U16BIT tran_rec:1;   // transmit/receive bit
          BT_U16BIT rtaddr:5;     // rt address field              (MSB)
       #endif
       }
BT1553_COMMAND;
```

Description

This structure is used to define a 1553 Command Word as it would appear on the bus. The fields of the Command Word are defined as individual elements of this structure. When the structure is completed, the resulting 16-bit value can be used directly on the 1553 bus.

Data Elements

wcount: This is the word count (or mode code) field of the Command Word. It should be a value from 0 to 31. Per the 1553 specification, a value of 0 implies a word count of 32 words.

subaddr: This is the RT subaddress field of the Command Word. It should be a value from 0 to 31. **Note** that a value of 0 (or 31, if subaddress 31 as a mode code is enabled) means that the *wcount* field should be interpreted as a mode code.

tran_rec: This is the transmit / receive bit of the Command Word. It should be a value of 1 or 0. A 0 indicates that the Command Word is for a "receive" command. A 1 indicates that the Command Word is for a "transmit" command.

rtaddr: This is the RT address field of the Command Word. It should be a value from 0 to 31. A value of 31 typically means that the command is a broadcast command.

7.2 1553 Status Word (BT1553_STATUS)

Code Definition

```
typedef struct bt1553_status
       {
       #ifdef NON_INTEL_BIT_FIELDS
          BT_U16BIT rtaddr:5;         // rt address field    (MSB)
          BT_U16BIT me:1;             // message error
          BT_U16BIT inst:1;           // instrumentation bit
          BT_U16BIT sr:1;             // service request
          BT_U16BIT res:3;            // unused bits
          BT_U16BIT bcr:1;            // broadcast received bit
          BT_U16BIT busy:1;           // busy flag bit
          BT_U16BIT sf:1;             // subsystem flag bit
          BT_U16BIT dba:1;            // dynamic bus acceptance flag bit
          BT_U16BIT tf:1;             // terminal flag bit   (LSB)
       #else  /* INTEL-Compatible bit field ordering */
          BT_U16BIT tf:1;             // terminal flag bit   (LSB)
          BT_U16BIT dba:1;            // dynamic bus acceptance flag bit
          BT_U16BIT sf:1;             // subsystem flag bit
          BT_U16BIT busy:1;           // busy flag bit
          BT_U16BIT bcr:1;            // broadcast received bit
          BT_U16BIT res:3;            // unused bits
          BT_U16BIT sr:1;             // service request
          BT_U16BIT inst:1;           // instrumentation bit
          BT_U16BIT me:1;             // message error
          BT_U16BIT rtaddr:5;         // rt address field    (MSB)
       #endif
       }
BT1553_STATUS;
```

Description

This structure is used to define a 1553 Status Word as it would appear on the bus. The fields of the Status Word are defined as individual elements of this structure. When the structure is completed, the resulting 16-bit value can be used directly on the 1553 bus.

Data Elements

tf: Terminal Flag Bit

dba: Dynamic Bus Acceptance Flag Bit

sf: Subsystem Flag Bit

busy: Busy Flag Bit

bcr: Broadcast Received Bit

res: Unused Bits

sr: Service Request

inst: Instrumentation Bit

me: Message Error

rtaddr: RT Address Field

7.3 BC Retry Parameters (BusTools_BC_Init argument)

Code Definition

```
#define BC_RETRY_ALTB       0x00001
#define BC_RETRY_NRSP       0x00002
#define BC_RETRY_ME         0x00004
#define BC_RETRY_BUSY       0x00008
#define BC_RETRY_TF         0x00010
#define BC_RETRY_SSF        0x00020
#define BC_RETRY_INSTR      0x00040
#define BC_RETRY_SRQ        0x00080
#define BC_RETRY_INV_WRD    0x00100
#define BC_RETRY_INV_SYNC   0x00200
#define BC_RETRY_MID_BIT    0x00400
#define BC_RETRY_TWO_BUS    0x00800
#define BC_RETRY_PARITY     0x01000
#define BC_RETRY_CONT_DATA  0x02000
#define BC_RETRY_EARLY_RSP  0x04000
#define BC_RETRY_LATE_RSP   0x08000
#define BC_RETRY_BAD_ADDR   0x10000
#define BC_RETRY_WRONG_BUS  0x20000
#define BC_RETRY_BIT_COUNT  0x40000
#define BC_RETRY_NO_GAP     0x80000
```

Description

This constants are used in the call to BusTools_BC_Init to enable the various Bus Controller message retry conditions.

Data Elements

BC_RETRY_ALTB: Retry on Alternate Bus.

BC_RETRY_NRSP: Retry on No Response.

BC_RETRY_ME: Retry on Message Error

BC_RETRY_BUSY: Retry on Busy Bit Set.

BC_RETRY_TF: Retry on Terminal Bit Set.

BC_RETRY_SSF: Retry on Subsystem Flag Set.

BC_RETRY_INSTR: Retry on Instrumentation Bit Set.

BC_RETRY_SRQ: Retry on Service Request Bit Set.

BC_RETRY_INV_WRD Retry on Invalid Word Error*

RETRY_INV_SYNC Retry on Inverted Sync**

BC_RETRY_MID_BIT Retry on Mid Bit Zero Crossing Error**

BC_RETRY_TWO_BUS Retry on Two-Bus error**

BC_RETRY_PARITY Retry on Parity Error**

BC_RETRY_CONT_DATA Retry on Non-Contiguous Data**

BC_RETRY_EARLY_RSP Retry on Early Response

BC_RETRY_LATE_RSP Retry on Late Response

BC_RETRY_BAD_ADDR Retry on Bad RT Address

BC_RETRY_WRONG_BUS Retry on Wrong Bus Error***

BC_RETRY_NO_GAPRetry on no Inter-Message Gap

* Use Invalid word retry in place of sync, parity, non-contiguous data, and mid-bit for F/W 6.10 and greater.
** Not supported by F/W v6.10 and greater
*** Not supported in f/W v5.0 and greater

7.4 BC Message Buffer (API_BC_MBUF)

Code Definition

```
typedef struct api_bc_mbuf
{
        BT_U16BIT messno; // Message number (0-based) 'FFFF' indicates end of aperiodic
        list
        BT_U16BIT control;           // Bus Controller Control Word.  Defines this
                                     // message to be either a 1553 data transfer
                                     // msg (BC_CONTROL_MESSAGE) or a list management
                                     // msg (conditional branch or noop).
        BT_U16BIT messno_next;       // Next message number
        BT_U16BIT messno_prev;       // Previous message number (Not used, V2.51 and up)

        // This group is for standard bc messages (not conditional messages)
        // Any given message may NOT be both a 1553 message and a conditional message
        // at the same time.  Data in this section is ignored for conditional msgs,
        // and data in the following section is ignored for 1553 messages.
        BT1553_COMMAND mess_command1;    // 1553 command word (Receive for RT→RT msgs)
        BT1553_COMMAND mess_command2;    // 1553 cmd word #2 (Transmit, RT→RT msgs)
        BT_U16BIT errorid;               // error injection buffer id number
        BT_U16BIT gap_time;              // time to delay after end of current msg
        BT1553_STATUS mess_status1;      // 1553 status word
        BT1553_STATUS mess_status2;      // 1553 status word #2 (for RT→RT msgs)
        BT_U32BIT status;                // interrupt status from h/w
        BT_U16BIT data[2][BT1553_BUFCOUNT]; // data buffers A and B

        // This group is for BC_CONTROL_CONDITION, #2 and #3 list management messages

        BT_U16BIT data_value;       // Data value to compare
        BT_U16BIT data_mask;        // Bit mask for compare
        BT_U16BIT address;          // Word number of the previous or specified msg
                                    // to compare for BC_CONTROL_CONDITION or
                                    // BC_CONTROL_CONDITION3 Conditional branch msgs.
        BT_U16BIT messno_branch;    // If '==' branch to this message
        BT_U16BIT messno_compare;   // Msg # containing data word in “address”

        // These variables are only used by the extended BC message function calls.

        BT_U16BIT start_frame;      // Start frame for message scheduling
        BT_U32BIT test_address;     // Byte address of word to be tested by
                                    // BC_CONTROL_CONDITION2 conditional branch msg.
        BT_U16BIT cond_count_val;   // Conditional counter reload value
        BT_U16BIT cond_counter;     // Conditional counter initial value

        BT_U16BIT data_buf_num1;    // Buffer number of the first BC data buffer
        BT_U16BIT data_buf_num2;    // Buffer number of the second BC data buffer
        BT1553_TIME time_tag;       // Tag Time for F/W version 3.97 and above
        BT_U16BIT   rep_rate;       // repeat rate for message scheduling.
}
API_BC_MBUF;
```

Description

This structure contains all information required to define a BC message. The specified information is used to create a BC message on the board. Six types of BC message can be defined by this structure, only one of which may be specified for any given message:

- **Conventional 1553 message**: This message includes the standard command word, and one or two data buffers for transmit/receive data. Mode codes, RT to BC, BC to RT, and broadcast messages may be defined using this format. To specify this type of message, you set BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP in the "control" word.
- **Branch message**: This message transfers control to another location in the BC message list. To specify this type of message, the BC_CONTROL_BRANCH bit of the "control" word must be set.
- **Conditional branch message (BC_CONTROL_CONDITION)**: This message transfers control to another message in the BC message list if a specified condition in the previous message is met. To specify this type of message, the BC_CONTROL_CONDITION bit of the "control" word must be set.
- **Conditional branch message (BC_CONTROL_CONDITION2)**: This message transfers control to another message in the BC message list if the specified location if the conditions of the contents at that location meet the specified conditions. To specify this type of message, the BC_CONTROL_CONDITION2 bit of the "control" word must be set.
- **Conditional branch message (BC_CONTROL_CONDITION3)**: This message transfers control to another message in the BC message list if a specified word in a specified message meets the specified conditions. To specify this type of message the BC_CONTROL_CONDITION3 bit of the "control" word must be set. See above when specifying the previous message as the source of the test word.
- **No-op message**: This message performs no function; it just jumps to the next message. To specify this type of message, the BC_CONTROL_NOP bit of the control word must be set.

Data Elements

messno: This is the message number of this BC message in the message list. The first message in the list is message #0.

control: This is the control word used to determine the contents of the BC message. It has two parts: (1) a message type field and (2) message control bits. The first seven entries below are message type definitions. The control word can only specify a single message type. The remaining entries are message control bits.

Message Types:

BC_CONTROL_BRANCH: This code indicates that this BC message is a branch message. When the microcode "executes" this message, it jumps to another position within the BC message list.

BC_CONTROL_CONDITION: This code indicates that this BC message structure defines a conditional branch message. This message transfers control to another position within the BC message list based on the contents of a specified word in the previous message.

BC_CONTROL_CONDITION2: This code indicates that this BC message structure defines a condition branch message. This message transfers control to another position within the BC message list based on the contents of a *user-specified location in memory*, as specified by the test_address parameter.

BC_CONTROL_CONDITION3: This code indicates that this BC message structure defines a conditional branch message. This message transfers control to another position within the BC message list based on the contents of a specified word in the message in the messno_compare parameter, rather than the previous message.

BC_CONTROL_MESSAGE: This code indicates that this BC message structure defines a standard BC message. It instructs the BC to generate a message on the bus (either transmit or receive). You can noop this message by calling BusTools_BC_MessageNoop. This results in a message type that is different from the BC_CONTROL_NOP.

BC_CONTROL_MSG_NOP: This code indicates that this BC message structure is a standard BC message. The message is put into the Noop state when it is written to the BC data buffer. You can activate this message by calling BusTools_BC_MessageNoop.

BC_CONTROL_LAST: This code indicates the last message to be executed in the BC message list. When the BC executes this message, it turns off the BC Run and Busy bits and stops. No further BC messages are output until the BusTools_BC_StartStop function is called to start the BC.

BC_CONTROL_HALT. This code indicates the last message to be executed in the BC message list. When the BC executes this message, it turns off the BC Run and clears the BC Enable External Sync Bit. This stops the BC and ensures that an external trigger pulse will not restart the BC. No further BC messages are output until the BusTools_BC_StartStop function is called to start the BC.

BC_CONTROL_NOP: This code indicates that this is a noop message (this message does nothing). It can be used to branch back to the beginning of the bus list when repeating a bus list.

BC_CONTROL_TIMED_NOP: This code indicates that this is a timed no-op message. It will execute inter-message gap timing but does not transmit data on the bus.

Message Control Bits:

BC_CONTROL_BUFFERA: This bit is set to indicate that Buffer A is to be used for data transfers. This bit is valid only for microcode revision 1.04 or later (all currently shipping hardware is above revision 1.04) (see the BusTools_GetRevision function. This bit is valid only for messages of type BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

BC_CONTROL_BUFFERB: This bit is set to indicate that Buffer B is to be used for data transfers. This bit is valid only for microcode revision 1.04 or later (see the BusTools_GetRevision function. This bit is valid only for messages of type BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

BC_CONTROL_CHANNELA or BC_CONTROL_BUSA: This bit is set to indicate that this message should be transmitted on the primary bus (Bus A). This bit is valid only for microcode revision 1.04 or later (see BusTools_GetRevision function). This bit is valid only for messages of type BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

BC_CONTROL_CHANNELB or BC_CONTROL_BUSB: This bit is set to indicate that this message should be transmitted on the secondary bus (Bus B). This bit is valid only for microcode revision 1.04 or later (see BusTools_GetRevision function). This bit is valid only for messages of type BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

BC_CONTROL_INTERRUPT: This bit is set, if the BusTools hardware needs to check the results of this message against the enabled BC interrupt conditions. If the bit is set and one of the enabled interrupt conditions is met, an interrupt record is added to the interrupt queue.

BC_CONTROL_MFRAME_BEG: This bit must be set for the first message in each minor frame. **Note** that the first message of a minor frame cannot be a conditional or branch message.

BC_CONTROL_MFRAME_END: This bit must be set for the last message in each minor frame. The last message of a minor frame cannot be a conditional or branch message.

BC_CONTROL_RETRY: This bit is set to indicate that retries are enabled for this message. This bit is valid only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

BC_CONTROL_RTRTFORMAT: This bit is set to indicate that this BC message is an RT to RT format message. In this case, both mess_command1 and mess_command2 must be defined. The first command must be a “receive”

and the second command must be a “transmit”. On reading a BC message from the BusTools hardware, both mess_status1 and mess_status2 are valid. This bit is valid only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

BC_CONTROL_USEBUFFA: If this bit is set, Buffer A should be used for data transfers for this message. If this bit is clear, Buffer B should be used for data transfers for this message. This bit is valid only for microcode revision 1.03 or earlier.

messno_next: This is the message number of the next message in the BC message list to be executed after this message. Using this entry, a conventional message can actually be combined with a branch message in a single entry. **Note** that message numbers are “0” based (the first message in the list is message #0).

messno_prev:: Was the message number of the previous message in the BC message list. Currently this parameter is unused.

mess_command1: This is the first MIL-STD-1553 command word to be placed onto the 1553 bus by this message.

mess_command1.rtaddr: This is the RT address to be placed in the first command word. This entry should be in the range 0 to 31 (inclusive). This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP. If Broadcast is enabled (see the BusTools_SetBroadcast function), an RT address of 31 is automatically a broadcast message.

mess_command1.subaddr: This is the RT subaddress to be placed in the first command word. This entry should be in the range 0 to 31 (inclusive). This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP. A RT subaddress of 0 is automatically a mode code command. In addition, if SA31 is enabled (see the BusTools_SetSa31 function), an RT subaddress of 31 is also a mode code command.

mess_command1.tran_rec: This is the transmit/receive bit to be placed in the first command word. This entry should be either 0, for receive, or 1, for transmit. This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

mess_command1.wcount: This is the word count (or mode code, if applicable) to be placed in the first command word. This entry should be in the range 0 to 31 (inclusive).

mess_command2: This is the second MIL-STD-1553 command word to be placed onto the 1553 bus by this message. It is used only by RT-to-RT messages. It is ignored for all other 1553 messages.

mess_command2.rtaddr: This is the RT address to be placed in the second command word. This entry should be in the range 0 to 31 (inclusive). This entry is used

only if the BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP. If Broadcast is enabled (see the BusTools_SetBroadcast function), an RT address of 31 is automatically a broadcast message.

mess_command2.subaddr: This is the RT subaddress to be placed in the second command word. This entry should be in the range 0 to 31 (inclusive). This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP. A RT subaddress of 0 is automatically a mode code command. In addition, if SA31 is enabled (see the BusTools_SetSa31 function), a RT subaddress of 31 is also a mode code command.

mess_command2.tran_rec: This is the transmit/receive bit to be placed in the second command word. This entry should be either 0, for receive, or 1, for transmit. This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

mess_command2.wcount: This is the word count (or mode code, if applicable) to be placed in the second command word. This entry should be in the range 0 to 31 (inclusive).

errorid: This is the error injection buffer id to be used with this message. If no errors are to be injected with this message, the id of a no-error error injection buffer must be supplied. Typically, the first error injection buffer (buffer 0) is defined to be the no-error buffer. This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

gap_time: This is the time from the mid-parity bit of the last word of the current message to the mid-sync bit of the next message. The delay is in microseconds, with a range of 5 – 65535. This entry is only for

start_frame: This the starting frame number for the message when message scheduling is enabled.

rep_rate: This the repeat rate for the message when message scheduling is enabled.

BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP. Gap time is not applied to BC_CONTROL_MSG_NOP messages until they are operational. Use BusTools_BC_MessageNoop to activate noop messages.

mess_status1: This is the 1553 status word returned by the RT because of executing this message. It is only valid after reading the BC message, using the BusTools_BC_MessageRead function. This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

mess_status2: This is the 1553 status word returned by the receiving RT as part of an "RT to RT" message. For non-"RT to RT" messages, this parameter is not valid and its value is indeterminate. It is only valid after reading the BC message, using the BusTools_BC_MessageRead function. This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP

status: This is the 32-bit message status from the Abaco Systems 1553 board. The definition of the individual bits in this parameter can be found in Section 7.11, "Interrupt Enable / Message Status Bits (32 bit)". This entry is used only for BC_CONTROL_MESSAGE or BC_CONTROL_MSG_NOP.

data: This is a data buffer portion of the message. There are two data buffers, each with 33 "BT1553_BUFCOUNT" words. There are 33 words to allow for the possibility of a 32-word message with a high word count error. The two buffers are provided to support the "Ping-Pong" buffering for BC messages. On writing a message, both buffers are stored in channel memory. On reading a message, both buffers are read from channel memory.

data_value: This is the 16-bit data value used for comparison in conditional branch instructions (BRANCH_CONDITION, BRANCH_CONDITION2 or BRANCH_CONDITION3 bits set)

data_mask: This is the 16-bit mask applied to the bus data prior to the comparison for conditional message. This entry is used only if BRANCH_CONDITION, BRANCH_CONDITION2 or BRANCH_CONDITION3 bits set bit are set.

address: This parameter indicates what data word should be used for the conditional message comparison. For type #1 or #3 conditional messages, this parameter specifies a word from the previous message. For type #2 conditional messages, this parameter is not used (see "test_address"). For type #1 messages, possible values are as follows:

0	Command word.
1	Command word #2 (for "RT to RT" messages only).
2	Status word.
3	Status word #2 (for "RT to RT" messages only).
4-35	Data words #1 through #32.

test_address: This parameter specifies the byte address of the word to be tested by a BC_CONTROL_CONDITION2 type #2 conditional message.

messno_branch: This is the destination message number in the BC message list executed if the specified comparison is true. **Note** that message numbers are "0" based (the first message in the list is message #0). If the comparison is false, the message specified in "messno_next" is executed next. This entry is used only if BRANCH_CONDITION, BRANCH_CONDITION2 or BRANCH_CONDITION3 bits set bit are set.

messno_compare: If BC_CONTROL_CONDITION3 is set, this word contains the message number of the message containing the "address" data word to be used for the conditional message comparison.

Time_tag: This is the BT1553_TIME structure for recording the 45-bit time tag for each BC message. Only BC message blocks and stop block report time tag. This feature is only available for F/W version 3.97 or higher.

7.5 BM Filter Buffer (API_BM_CBUF)

Code Definition

```
typedef struct api_bm_cbuf
{
   union
   {
      DWORD wcount;    // enabled word counts (bit field)
      DWORD modecode;  // enabled mode codes
   };
   WORD pass_count;    // number of passes before interrupt
}
API_BM_CBUF;

/*********************************************************
*    BM control buffer "mode_code" bits
*********************************************************/

#define BM_FILTER_MC_DBC     0x00000001L // dynamic bus control
#define BM_FILTER_MC_SYNC    0x00000002L // synchronize
#define BM_FILTER_MC_TSWS    0x00000004L // transmit status word
#define BM_FILTER_MC_STST    0x00000008L // initiate self-test
#define BM_FILTER_MC_XSD     0x00000010L // transmitter shutdown
#define BM_FILTER_MC_OXSD    0x00000020L // override transmitter shutdown
#define BM_FILTER_MC_ITF     0x00000040L // inhibit terminal flag bit
#define BM_FILTER_MC_OITF    0x00000080L // Override inhibit terminal flag bit
#define BM_FILTER_MC_RRT     0x00000100L // Reset remote terminal

#define BM_FILTER_MC_TVWC    0x00010000L // Transmit vector word
#define BM_FILTER_MC_SWO     0x00020000L // Synchronize with data word
#define BM_FILTER_MC_TLC     0x00040000L // Transmit last command
#define BM_FILTER_MC_TBW     0x00080000L // Transmit bit word
#define BM_FILTER_MC_SXSD    0x00100000L // Selected transmitter shutdown
#define BM_FILTER_MC_OSXS    0x00200000L // Override selected tx shutdown
```

Description

The BM microcode records messages as they appear on the 1553 bus. When a message is recorded, information about the message (including content and error status) is placed in one of the BM Message buffers allocated in the channel's Bus Monitor memory.

The BM filter provides the ability to specify that only certain messages be recorded. The filter operates based on the command word (CW) of the message (which is a combination of the RT address, RT subaddress, transmit / receive flag and word count or modecode for the message). When the CW is received, the BM filter determines if messages with that specific CW are to be recorded or ignored by the BM microcode. If the BM filter bit for the CW in a specific message is reset (0), then the message is ignored by the BM microcode and the message is not stored in one of the BM Message buffers.

If the BM filter bit for a specific CW is set (1), the filter checks a filter counter for that CW. The counter is initialized by the application to indicate the fraction of BM

messages with a specific CW to be recorded. A counter set to n indicates that every nth message should be recorded; the other n-1 messages should be ignored.

This structure contains the BM filter enable / disable bits for various word count and modecode possibilities for a specific RT subunit (RT address, subaddress, and transmit / receive flag combination). If the specified bit is not set, the BM ignores messages with the specified CW. If the specified bit is set, messages are recorded subject to the specified counter. The application can independently specify the BM filter bits for each RT subunit using the BusTools_BM_FilterWrite function.

If the RT subaddress is a non-modecode subaddress, then the 32-bit *wcount* parameter is used. In this case, each bit in the parameter is used to enable or disable BM recording for a specific word-count for the specified RT subunit. If the specified bit is 0, the BM ignores messages with the specified RT address, subaddress and word count. If the specified bit is "1", a message with the specified RT address, subaddress and word count is stored in the BM Message buffers by the BM. The bit assignments are as follows:

0x00000001L: control word count = 0 (32 words)

0x00000002L: control word count = 1

0x00000004L: control word count = 2

• • •

0x80000000L: control word count = 31

If the RT subaddress is a modecode subaddress, then the 32-bit modecode parameter is used. Bits are set in this parameter to enable / disable BM operations with the associated mode codes. The bit assignments for each modecode are provided in the BM_FILTER_MC_XXX parameters listed above.

Data Elements

wcount: This is the 32-bit enable / disable word for non-modecode subaddresses. The bit assignments are described above. **Note** that this parameter is a union with modecode. The BusTools BM hardware does not differentiate between modecode subaddresses and normal subaddresses with respect to filter operations.

modecode: This is the 32-bit enable / disable word for modecode subaddresses. These bit assignments are specified in the BM_FILTER_MC_XXX parameters. **Note** that this parameter is a union with "wcount". The BusTools BM hardware does not differentiate between modecode subaddresses and normal subaddresses with respect to filter operations.

pass_count: The counter parameter used by the BM filter. If this value is set to n, then every nth message with the specified command word is recorded. If the application wants every message recorded, set this parameter to 1.

7.6 BM Message Buffer (API_BM_MBUF)

Code Definition

```
typedef struct api_bm_mbuf
{
   BT_U32BIT      messno;  // Message number (generated by API, 1-based)
   BT_U32BIT      int_status;     // Interrupt status from board
   BT1553_TIME    time;           // Time of message (48-bits, 1 us LSB)
   BT1553_COMMAND command1;       // 1553 command word #1 (Rx for RT→RT)
   BT_U16BIT      status_c1;      // 1553 command word #1 error status
   BT1553_COMMAND command2;       // 1553 command word #2 (Tx for RT→RT)
   BT_U16BIT      status_c2;      // 1553 command word #2 error status
   BT1553_BMRESPONSE response1; // 1553 response time #1 (byte)
   BT1553_BMRESPONSE response2; // 1553 response time #2 (byte)
   BT1553_STATUS status1;  // 1553 status word #1 (Transmit for
                                // RT→RT or Broadcast RT→RT)
   BT_U16BIT     status_s1;       // 1553 status word #1 error status
   BT1553_STATUS status2;  // 1553 status word #2 (Receive for
                              // RT→RT, NULL for Broadcast RT→RT)
   BT_U16BIT     status_s2;       // 1553 status word #2 error status
   BT_U16BIT     value[BT1553_MBUFCOUNT];  // 1553 data words
   BT_U8BIT      status[BT1553_MBUFCOUNT]; // 1553 status for data words
}
API_BM_MBUF;
```

Description

This structure contains all information provided by the Abaco Systems 1553 board when a message is detected on the 1553 bus. This information includes all command words, data words, and status words as they appeared on the bus. The RT response time (the time required for the RT to provide its status word) is provided. In the case of an “RT to RT” message, the second RT response time is also provided. Finally, a message number and time stamp are provided.

Errors information detected for the message is provided in two ways:

- for each word appearing on the bus, a 16-bit error status is provided. See “BM Word Status Bits (8/16 bit)”.
- for the entire message, a 32-bit error status is provided. See “Interrupt Enable / Message Status Bits (32 bit)”.

The low order 16 bits of the 32-bit error status is a logical “OR” of the 16 bit error status words from all words of the message.

The structure contains value and status information for up to 32 data words, however, only the number of data words specified in the command word (and actually present on the bus) are valid. The remaining value and status information is indeterminate.

Certain elements of this structure (command2, status_c2, response2, status2, and status_s2) are valid only in the case of an RT to RT message. You can determine

whether this message is RT to RT by checking the BT1553_INT_RT_RT_FORMAT bit of the int_status parameter of this structure.

Data Elements

messno: This is the buffer number for this message. Buffer number corresponds to the buffers created on the call to BusTools_BM_MessageAlloc.

int_status: This is the 32-bit Interrupt Status Word from the BusTools channel for this message. See Section 7.11, "Interrupt Enable / Message Status Bits (32 bit)".

time: The time at which this message was detected on the bus. It is measured from when the BM was started using the BusTools_BM_StartStop function. Each time the BM is restarted, the time counters are reset.

command1: This is the command word as it appeared on the bus. The components of the command word (RT address, subaddress, word count, and transmit/receive flag) can be accessed using the elements of this structure.

status_c1: This is the 16-bit error (or quality) word associated with the command word. For the bit definitions in this word, see "BM Word Status Bits (8/16 bit)".

command2: (valid only for "RT to RT" messages): This is the second command word as it appeared on the bus. The components of the command word (RT address, subaddress, word count and transmit/receive flag) can be accessed using the elements of this structure.

status_c2 (valid only for "RT to RT" messages): This is the 16-bit error (or quality) word associated with the second command word. For the bit definitions in this word, see Section 7.8, "BM Word Status Bits (8/16 bit)".

response1: The time delay that was measured prior to the RT providing its status word.

status1: This is the 1553 status word provided by the RT in response to this message. The components of the status word (individual status bits) can be accessed using the elements of this structure.

status_s1: This is the 16-bit error (or quality) word associated with the status word. For the bit definitions in this word, see Section 7.8, "BM Word Status Bits (8/16 bit)".

response2 (valid only for "RT to RT" messages): The time delay that was measured prior to the second RT providing its status word.

status2: (valid only for "RT to RT" messages): This is the 1553 status word provided by the second RT in this message.

status_s2: (valid only for “RT to RT” messages): This is the 16-bit error (or quality) word associated with the second status word. For the bit definitions in this word, see Section 7.8, “BM Word Status Bits (8/16 bit)”.

value[i]: This is the “i”th data word as it appeared on the 1553 bus. Only command1.wcount data entries are valid in this message.

status[i]: This is the 8-bit error (or quality) word associated with the “i”th data word. For the bit definitions in this word, see Section 7.8, “BM Word Status Bits (8/16 bit)”.

user1/user2: These two conditionally-defined data elements are for the Bus Monitor User data option. They are set by calling BusToos_BM_SetUserData to write data into onboard registers. The firmware updates the BM message buffer with the contents of these two registers. Define the macro _BM_USER_DATA_ to use these two values. Using BM User Data is not compatible with playback or BusTools/1553 User Interface Software.

7.7 BM Trigger Buffer (API_BM_TBUF)

Code Definition

```
/*****************************************************************
*  BM Trigger Buffer definition
*
*   4 - Start Trigger Events (only 3 Active) A - B - C - D(Not Used)
*   4 - Stop Trigger Events (only 3 Active) E - F - G - H(Not Used)
*
*   control: dialog box entries for order of events:
*                        START SIDE          STOP SIDE
*                0 -> unconditionally            never
*                1 -> if A                       if E
*                2 -> if A AND B                 if E AND F
*                3 -> if A OR B                  if E OR F
*                4 -> if A with B (same message) if E with F (same msg)
*                5 -> if B armed by A            if F armed by E
*                6 -> if A with B and C          if E with F and G
*                7 -> if A with B or C           if E with F or G
*                8 -> if C armed by A with B     if G armed by E with F
*                9 -> if A with B armed by C     if E with F armed by G
*               10 -> if A and B and C           if E and F and G
*               11 -> if A or B or C             if E or E or G
*
*   type:    trigger type:
*                    0 -> none
*                    1 -> command
*                    2 -> status
*                    3 -> data
*                    4 -> Not used
*                    5 -> Interrupt status lsb
*                    6 -> Interrupt status msb
*
*   mask:   16 bit mask (applied prior to compare)
*   value:  16 bit value to compare
*   word:   word number (data trigger - 3) or offset (user offset trigger - 5)
*   count:  number of times event must occur before event_occurred bit is set
*
*****************************************************************/
typedef struct api_bm_tbuf
{
   BT_U16BIT   trig_ext;         // 1 -> external trigger input enabled
                                 // 2 -> external output on every BM interrupt
                                 // (rest of structure is ignored unless zero)
   BT_U16BIT   trig_err;         // 1 -> trigger on errors
   BT_U16BIT   trig_ext_output;  // 1 -> external output on trigger
                                 // 2 -> repetitive external output on trigger
   BT_U16BIT   control1;         // control of trigger start
   BT_U16BIT   control2;         // control of trigger stop
   struct
   {                    // Start events programmed here
      BT_U16BIT type;   // Event type which causes trigger
      BT_U16BIT mask;   // Specified 1553 word compared with "word" under this mask
      BT_U16BIT value;  // Value of the command, status or data word for trigger
      BT_U16BIT word;   // Word number within message to test
      BT_U16BIT count;  // Number of times event occurs before Start declared
   }
   capture[4];          // Two Start events are supported by the old firmware.
   struct
      {                 // Stop events programmed here
      BT_U16BIT type;   // Event type which causes trigger
      BT_U16BIT mask;   // Specified 1553 word compared with "word" w/this mask
      BT_U16BIT value;  // Value of the cmnd, status or data word for trigger
      BT_U16BIT word;   // Word number within message to test
      BT_U16BIT count;  // Number of times event occurs before Stop declared
   }
   stop[4];             // Two Stop events are supported by the old firmware.
}  API_BM_TBUF;
```

Description

Different versions of the hardware support different combinations of triggering: external inputs, external outputs, one-shot, and repetitive modes. For a complete description, see the hardware reference manual for the particular board of interest.

Data Elements

See the description of the BM Trigger buffer definition above.

7.8 BM Word Status Bits (8/16 bit)

Code Definition

```
#define BT1553_INT_HIGH_WORD      0x0001
#define BT1553_INT_INVALID_WORD   0x0002
#define BT1553_INT_LOW_WORD       0x0004
#define BT1553_INT_INVERTED_SYNC  0x0008
#define BT1553_INT_MID_BIT        0x0010
#define BT1553_INT_TWO_BUS        0x0020
#define BT1553_INT_PARITY         0x0040
#define BT1553_INT_NON_CONT_DATA  0x0080
#define BT1553_INT_EARLY_RESP     0x0100
#define BT1553_INT_LATE_RESP      0x0200
#define BT1553_INT_BAD_RTADDR     0x0400
#define BT1553_INT_CHANNEL        0x0800
#define BT1553_INT_WRONG_BUS      0x2000
#define BT1553_INT_BIT_COUNT      0x4000
#define BT1553_INT_NO_IMSG_GAP    0x8000
```

Description

The Bus Monitor Word Status is a 16-bit parameter, with each bit indicating a different condition that can be detected and reported by the Abaco Systems 1553 board for an individual word (either command, data or status) of a 1553 message. The Word Status is reported as part of the BM Message buffer for every word detected on the bus. The bit definitions for the BM Word Status match the first 16 bits of the Interrupt Enable / Message Status parameter.

The Bus Monitor records 8 bits of error status information for each data word logged. These 8 bits are the first 8 bits in the above list.

Data Elements (in alphabetical order)

- BT1553_INT_BAD_RTADDR: This bit indicates that the RT address in the status word response is not identical to the RT addressed in the command word.
- BT1553_INT_BIT_COUNT: This bit indicates that the bit count of one or more words in the message was not the expected value (a value of 16). This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_BIT_COUNT_DATA: This bit indicates that the bit count of the associated data word in the message was not the expected value (a value of 16). This bit is only defined for data words within a API_BM_MBUF structure. This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_CHANNEL: This bit indicates the bus on which the message was detected. If this bit is set, the message was detected on bus B. If this bit is not set, the message was detected on bus A.
- BT1553_INT_EARLY_RESP: This bit indicates that the 1553 decoder detected a command sync with < 2μs of bus dead time when a status word is expected.

- BT1553_INT_HIGH_WORD: This bit indicates that the message contained more data words than was indicated in the word count field of the command word.
- BT1553_INT_INVALID_WORD: This bit indicates that any of the following was detected on one or more words of the message: inverted sync, invalid Manchester II encoding (including zero and crossing errors), bit count error or parity error.
- BT1553_INT_INVERTED_SYNC: This bit indicates that the sync field was inverted from what was expected by the message transfer format on one or more of the words in the message. This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_LATE_RESP: This bit indicates that the 1553 decoder did not detect a command sync within the specified "Late Response Timeout" time when a status word is expected. This time is set with the wTimeout2 parameter to the BusTools_BC_Init function.
- BT1553_INT_LOW_WORD: This bit indicates that the message contained less data words than were indicated in the word count field of the command word.
- BT1553_INT_MID_BIT: This bit indicates that successive mid-zero crossings were not within 150ns of the expected time for any successive bits in any word of the message (except the sync bit – see below). This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_MID_SYNC: This bit indicates that successive mid-zero crossings were not within 150ns of the expected time for any mid-sync or end-sync bit time. This condition also sets the BT1553_INT_INVALID_WORD bit. Not currently in use.
- BT1553_INT_NO_IMSG_GAP: This bit indicates that the mid-sync zero crossing of the next command sync was detected prior to 4μs preceding the mid-zero crossing of the parity bit of the last word of the current message. The next command sync may or may not produce a valid command word.
- BT1553_INT_NON_CONT_DATA: This bit indicates that a gap was detected between successive data words in the message. The hardware allows a 4-μs gap before declaring a Low Word error.
- BT1553_INT_PARITY: This bit indicates that a parity error was detected in one or more words of the message. Odd parity is used (per the MIL-STD-1553 Specification). This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_TWO_BUS: This bit indicates that both buses (bus A and bus B) were active sometime during the message.
- BT1553_INT_WRONG_BUS: This bit indicates that the RT responded on a different bus than the one on which the command word was transmitted.

7.9 Device List Structure (DeviceList)

Code Definition

```
typedef struct devicelist
{
   BT_INT  num_devices;
   BT_UINT device_name[MAX_BTA];
   BT_INT  device_num[MAX_BTA];
   char    name[MAX_BTA][256];
}
DeviceList;
```

Description

This structure holds information about the Abaco Systems MIL-STD-1553 devices installed in the system. A pointer to this structure is passed for BusTools_ListDevices to fill in the information.

Data Elements

nm_devices: The number of MIL-STD-1553 devices found in the system.

Device_name: An array of unsigned integers containing the device name value.

```
#define PMC1553     0x20/* This is the PMC-1553 native PCI board                */
#define PCI1553         0x40    /* This is the PCI-1553 native PCI board        */
#define CPCI1553    0x50
#define ISA1553         0x80    /* This is the ISA-1553 native ISA board        */
#define DT1553      0x90
#define PCC1553         0xA0    /* This is the PCC-1553 PCMCIA board            */
#define VME1553     0x100       /* This is the VME-1553 Native VME Board        */
#define QPMC1553    0x110        /* This is the QPMC-1553 native PMC board      */
#define IPD1553     0x120       /* This is the Dual Channel single wide IP      */
#define VXI1553     0x140       /* This is the Native Plug-n-Play VXI-1553      */
#define QPCI1553    0x160       /* This ia the QPCI-1553 Native PCI board       */
#define Q1041553    0x170       /* This is a PC\104 4-Ch ISA board              */
#define Q1041553P   0x180       /* This is a PC\104 4-Ch PCI board              */
#define QVME1553    0x190       /* This is a Quad Channel VME-1553 New Arch     */
#define PCCD1553    0x200       /* This is a Dual Channel PCCard-D1553          */
#define QCP1553     0x210       /* This is the Quar-Channel cPCI-1553 board     */
#define QPCX1553    0x220       /* This is the QPCX-1553 Native PCI board       */
#define R15EC           0x230   /* This is the R15-EC express card board        */
#define RXMC1553    0x260        /* This is the RXMC-1553                       */
#define R15AMC      0x240       /* This is the R15-AMC (QPM1553 variant)        */
#define QPM1553     0x110       /* This is the QPM-1553 (QPMC1553 variant)      */
#define AMC1553     0x110       /* This is the AMC-1553 (QPMC1553 variant)      */
#define R15AMC          0x240   /* This is the RoHS AMC board                   */
#define RPCIe1553       0x250   /* This is the RoHS PCI-E board                 */
```

```
#define RXMC1553          0x260    /* This is the RoHS XMC board                    */
#define AR15VPX           0x280    /* This is the AR15-VPX (1553/429)               */
#define R15XMC2           0x300     /* Same as above                                */
#define LPCIE1553         0x320    /* This is the low profile PCI express           */
#define RAR15XMCXT        0x360    /* This is the RAR15XMC extended Temp XMC        */
#define R15USB            0x3000   /* USB Board                                     */
#define MPCIE1553         0x400    /* This is the Mini PCI express                  */
```

These are legacy boards that are no longer available.

device_num: Array containing the installation device number for each board found.

Name: An array of strings holding the name of the board type.

7.10 Error Injection Definitions (API_EIBUF and API_ ENH_EIBUF)

Code Definition

```
typedef struct api_eibuf
{
   BT_U16BIT buftype;  // error injection buffer type struct
   {
      BT_U8BIT etype;  // error code (EI_NONE, EI_PARITY …
      BT_U8BIT edata;  // error data (if req'd)
   }
   error[EI_COUNT];
}
API_EIBUF;

typedef struct api_enh_eibuf
{
   BT_U16BIT buftype;  // error injection buffer type struct
   {
      BT_U8BIT etype;  // error code (EI_NONE, EI_PARITY, etc)
      BT_UINT  edata;  // error data (if req'd)
   }
   error[EI_COUNT];
}
API_ENH_EIBUF;
```

Description

This structure is used to control Error Injection by the BusTools microcode. Error Injection is defined on a word-by-word basis for a message. The device putting words out to the bus injects the errors. Thus, in typical operation, the Bus Controller would inject some errors (e.g., errors on the control word and any data transmitted by the BC). The Remote Terminal would inject other errors (e.g., errors on the status word and any data transmitted by the RT).

There are five types of Error Injection buffers defined. The different types are used in different situations. The types are:

- EI_BC_REC (**BC: Receive**): Error Injection buffer type used to define the errors to be injected by the BC into a Receive Message. This buffer contains error specifications for 34 words that can be transmitted by the BC in a Receive Message. The first error spec is for errors to be injected in the command word. The next 33 error specs are for errors to be injected in the data words. An extra word is reserved in case a Word Count Error is requested.
- EI_BC_TRANS (**BC: Transmit**): Error Injection buffer type used to define the errors to be injected by the BC into a Transmit Message. Since the BC generates only one word in a Transmit Message (the control word), there is only one error specification in this type of buffer.
- EI_RT_REC (**RT: Receive**): Error Injection buffer type used to define the errors to be injected by a RT into a Receive Message. Since the RT only generates one

word in a Receive Message (the status word sent to the BC after the data has been received), there is only one error specification in this type of buffer.

- EI_RT_TRANS (**RT: Transmit**): Error Injection buffer type used to define the errors to be injected by a RT into a Transmit Message. This buffer contains error specifications for 34 words that can be transmitted by the RT in a Transmit Message. The first error spec is for errors to be injected in the status word. The next 33 error specs are for errors to be injected in the data words. An extra word is reserved in case a Word Count Error is requested.
- EI_BC_RTTORT (**BC control of a RT-to-RT message**): Error Injection buffer type used to define the errors to be injected by a BC into a RT-to-RT Message. This buffer contains error specifications for two words that are transmitted by the BC in a RT-to-RT Message (the two command words).

Data Elements

buftype: The Error Injection buffer type (one of EI_BC_REC, EI_BC_TRANS, EI_RT_REC, EI_RT_TRANS, or EI_BC_RTTORT as defined above).

error[i].etype: Error Injection specification for a single word transmitted onto the bus. The following shows possible errors.

- EI_NONE- (**no error**): This entry is used when there is to be no error injected for this particular word of the message. The error[i].edata element is ignored.
- EI_BADADDR- (**Respond with Wrong Address: RT**): Normally, the RT status word contains the address of the RT. This error forces the status word to contain a different address. The address you use is specified in the error[i].edata element of the structure. You can use address 0 to 31. This error can be specified only for RT: Receive and RT: Transmit EI buffer types.
- EI_BITCOUNT- (**Bit Count Error: RT, BC**): Use this entry to inject a bit count error into a specific word of a message. The hardware normally transmits a sync pulse followed by 16 data bits ending with a parity bit for each word sent out on the 1553 bus. When you select this error, the actual number of data bits to be transmitted in the word must be specified in the error[i].edata entry. While the low-level hardware format allows values from 1 to 32, it is recommended that values in the range 14 to 19 be used (as specified in the MIL_STD_1553 RT validation test plan). Values outside of this range may cause other parts of the hardware to function improperly. This error can be specified for any entry in any EI buffer type. When bit count is injected, the parity bit is counted as a bit. Thus, injecting 17 bits may produce a parity error, but is not a bit count error.
- EI_PARITY- (**Parity Error: RT, BC**): This entry injects a parity error into a particular word of the message. This error does not use error[i].edata. You can specify this error for any entry in any EI buffer type.
- EI_SYNC- (**Inverted Sync Error: RT, BC**): This entry injects an inverted sync error into a particular word of the message. This error does not use error[i].edata. You can specify this error for any entry in any EI buffer type.

- EI_DATAWORDGAP- (**Data word Gap Error: RT, BC**): This inject a data word gap from .5 to 2.5 μs depending on the programming.
- EI_WORDCOUNT- (**Word Count Error: RT, BC**): Normally, the command word specifies the number of words in a message. This error forces the actual number of words in the message to differ from that in the command word. Use any word count from 1 to 33. Specify this error condition on the command word of a BC message (error[0].edata) for BC Receive messages. Specify this error on the status word of an RT message (error[0].edata), for RT Transmit messages.
- EI_MIDBIT, EI_MIDPARITY and EI_MIDSYNC (**Mid-Bit and Mid-Sync Errors: RT, BC**) - These error injection codes delay the zero crossing point by 300ns from where it is expected. For mid-bit zero-crossing errors, you use error[i].edata to select the bit, 0 through 15. Bit 0 is the LSB of the 16-bit word, however it is the last bit to be transmitted in the serial stream. Similarly, bit 15, the MSB is the first bit transmitted. The Mid-Sync and Mid-Parity errors do not use error[i].edata.
- EI_RESPWRONGBUS (**Response on Wrong Bus: RT**) - This bit set causes the RT to respond with the status word using the address programmed in the five LSBs of the error[0].edata.
- EI_BIPHASE – (**Bi-Phase Error: RT, BC**) - A bi-phase bit error is when there is no zero crossing for the entire bit time. Use error[i].edata to select the bit. It is not predictable how a 1553 decoder interprets a word when this error is injected on one of the first two bits, as it depends on timing and the states of the first two bits. When the bit doesn't cross zero, it may stay in one state for 1½ μs 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.
- EI_LATERESPONSE (**Programmable Response: RT**) – This RT selection allows the application to enter a programmable response time of up to 31½μs, programmed as a binary number in error[0].edata, with the LSB equal to 500ns. An entry less than 4μs constitutes an *early response*. An entry greater than 14μs constitutes a *late response* according to the MIL-STD-1553B Specification. Values of less than 8μs are *not guaranteed* to be attainable by the hardware. This error only applies to the RT status word.

- EI_ENH_ZEROXNG – EI_TCENH_ZEROXNG (**Enhanced Zero Crossing: BC-RT**) – This selection is only available when using firmware version 5.x or greater and requires using the function BusTools_EI _EbufWriteENH. This error offsets the zero crossing transition by the amount set in edata of the API_ENH_EIBUF structure. Setting the edata's three MSB's to either 111 or 101 determines whether use transmitter compensated zero-crossing.

Bit	Field	Definition
5-0	ZC_HalfBit	Half bit time of the selected word
7-6	reserved	
11-8	ZC_Offset	Offset of the transition
12	ZC_Early	Transition is early or late
13	1	Must be binary "1"
14	TC	Transmitter compensated (active low)
15	1	Must be binary "1"

Applying Zero-Crossing to a word offsets the selected transition by the amount determined in the ZC_Offset field. The four-bit ZC_Offset resolution (LSB) is 25ns, which allows for a range of 400ns.

The transition may be programmed to occur before the expected transition by setting the ZC_Early bit, or after the expected transition by clearing the ZC_Early bit.

Each 1553 word consists of forty half-bit times of 500ns. The ZC_HalfBit field determines which half-bit time contains the injected zero crossing error..

Early and late transitions may be programmed in the half-bit time indicated:

ZC_HalfBit = 4; // mid-sync

ZC_HalfBit = 7; // end-sync

ZC_HalfBit = ((BitCount x 2) + 6); // mid-bit*

ZC_HalfBit = ((BitCount x 2) + 7); // end-bit*

ZC_HalfBit = 40; // mid-parity

ZC_HalfBit = 41; // end-parity/start data sync

*The first bit transmitted on the bus is BitCount = 1 up to the last bit before parity is BitCount = 16.

Only one transition per message may be injected with a zero-crossing error.

- When using zero-crossing for RT validation, the affected half bit might require adjustment for differences of the MIL-STD-1553 transmitter outputs when the inputs are stable for greater than 500ns (1.0, 1.5 or 2.0 us). Programming the TC bit to a logic zero determines *Transmitter Compensated Zero-Crossing* which adjusts

the early transition or the transition following a late transition by moving it 25ns later so that plus or minus 150ns is met at the UUT. Generally, compensation is required if testing an RT near the receiver margins of plus or minus 150ns, Setting the TC bit to a logic one does not apply transmitter compensation

error[i].edata: If the error injection spec requires data, this is the data value. See above for the definition of this value. If the specified error does not require this data than set this word to zero.

7.11 Interrupt Enable / Message Status Bits (32 bit)

Code Definition

```
/*****************************************************************
*  Bit definitions for BusTools Interrupt Enable
*  and Interrupt Status entries (8-/16-/32-bit interrupt/status masks)
*****************************************************************/

#define BT1553_INT_HIGH_WORD       0x00000001L // high word error
#define BT1553_INT_INVALID_WORD    0x00000002L // invalid word error    **
#define BT1553_INT_LOW_WORD        0x00000004L // low word error
#define BT1553_INT_INVERTED_SYNC   0x00000008L // inverted sync         **
#define BT1553_INT_MID_BIT         0x00000010L // mid bit error         **
#define BT1553_INT_TWO_BUS         0x00000020L // Two bus error V4.25
#define BT1553_INT_PARITY          0x00000040L // parity error          **
#define BT1553_INT_NON_CONT_DATA   0x00000080L // non-contiguous data   **
#define BT1553_INT_EARLY_RESP      0x00000100L // early response
#define BT1553_INT_LATE_RESP       0x00000200L // late response
#define BT1553_INT_BAD_RTADDR      0x00000400L // incorrect rt address
#define BT1553_INT_CHANNEL         0x00000800L // Bus (0=A, 1=B)
#define BT1553_INT_WRONG_BUS       0x00002000L // response on wrong bus
#define BT1553_INT_BIT_COUNT       0x00004000L // bit count error       **
#define BT1553_INT_NO_IMSG_GAP     0x00008000L // no intermessage gap
//      Note: In the above list, "**" means the bit can be set for a
//            data word (vs. a command or status word, or a message),
#define BT1553_INT_END_OF_MESS     0x00010000L // end of message
#define BT1553_INT_BROADCAST       0x00020000L // broadcast message
#define BT1553_INT_RT_RT_FORMAT    0x00040000L // rt-to-rt message format
#define BT1553_INT_RESET_RT        0x00080000L // reset rt
#define BT1553_INT_SELF_TEST       0x00100000L // self test
#define BT1553_INT_MODE_CODE       0x00200000L // message is a Mode Code
                                                // (wcs > 3.07, was BIT FAIL)
#define BT1553_INT_NOCMD           0x00400000L // No command seen by decoder
#define BT1553_INV_RTRT_TX         0x00800000L // Invalid RTRT TX CMD2
#define BT1553_RTRT_RCV_NRSP       0x01000000L //
#define BT1553_INT_RETRY           0x02000000L // retry N/A for BM
#define BT1553_INT_NO_RESP         0x04000000L // no response (for RT→RT, set
                                                // if EITHER was no response)
#define BT1553_INT_ME_BIT          0x08000000L // 1553 status wd msg error bit
#define BT1553_INT_ALT_BUS         0x80000000L // retry on alternate bus.

// *****  SOFTWARE ONLY BITS ***** (not set by the hardware)

#define BT1553_INT_TRIG_BEGIN      0x10000000L // message with trigger begin
#define BT1553_INT_TRIG_END        0x20000000L // message with trigger end
#define BT1553_INT_BM_OVERFLOW     0x40000000L // message at buffer overflow
```

Description

The Interrupt Enable / Message Status are each 32-bit parameters, with each bit indicating a different condition which can be detected and/or reported by the Abaco Systems 1553 board. In the case of the Interrupt Enable parameter, each bit tells the Abaco Systems 1553 board that if the specified condition is detected, an interrupt record should be added to the interrupt queue on the board. In the case of the

Message Status parameter, each bit indicates a condition that was detected by the Abaco Systems 1553 board.

There are three situations where the Interrupt Enable parameter is used:

- BC Interrupt Enable (specified in the BusTools_BC_Init function and stored globally in one of the RAM registers).
- BM Interrupt Enable (specified as one of the parameters in each BM message buffer).
- RT Interrupt Enable (specified as one of the parameters in each RT message buffer).

While the same Interrupt Enable format is used for each of these situations, there may be certain conditions that can't be detected or do not make sense for a particular situation. See the table below for where each defined condition is applicable.

The Message Status parameter appears in three message buffers:

- BC Message buffer
- BM Message buffer
- RT Message buffer

In each case, the BusTools microcode fills in the value based on the conditions detected for that particular message. It should be noted that the Interrupt Enable bits set separately do not affect the status bits set in the status word. The status bits are set for all conditions detected by the microcode for that message.

Data Elements (in alphabetical order)

- BT1553_INT_BAD_RTADDR: This bit indicates that the RT address in the status word response is not identical to the RT addressed in the command word.
- BT1553_INT_BIT_COUNT: This bit indicates that the bit count of one or more words in the message was not the expected value (16). This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_BM_OVERFLOW: This bit indicates that the BM overflowed at this point and some messages were lost. The number lost can be determined by examining the message numbers.
- BT1553_INT_BROADCAST: This bit indicates that the message was a broadcast message. This is set in BC, BM and RT messages.
- BT1553_INT_CHANNEL: This bit indicates the channel on which the message was detected. If this bit is set, the message was detected on Bus B; otherwise, the message was detected on Bus A.
- BT1553_INT_EARLY_RESP: This bit indicates that the 1553 decoder detected a command sync with less than 2μs of bus dead time when a status word is expected.

- BT1553_INT_END_OF_MESS: This bit indicates that the parity bit of the last word of the message has been transmitted or received on the 1553 bus. This bit is set on every legal message with a valid command word, regardless of any other detected error conditions.
- BT1553_INT_HIGH_WORD: This bit indicates that the message contained more data words than was defined in the word count field of the command word.
- BT1553_INT_INVALID_WORD: This bit indicates that any of the following was detected on one or more words of the message: inverted sync, invalid Manchester II encoding (including zero and crossing errors), bit count error, or parity error.
- BT1553_INT_INVERTED_SYNC: This bit indicates that the sync field was inverted from what was expected by the message transfer format on one or more of the words in the message. This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_LATE_RESP: This bit indicates that the 1553 decoder did not detect a command sync within the specified Late Response Timeout when a status word is expected. This time is set with the "wTimeout2" parameter of the BusTools_BC_Init function.
- BT1553_INT_LOW_WORD: This bit indicates that the message contained fewer data words than were indicated in the word count field of the command word.
- BT1553_INT_ME_BIT: This bit indicates that the 1553 Status Word response had the Message Error bit set.
- BT1553_INT_MID_BIT: This bit indicates that successive mid-zero crossings were not within 150ns of the expected time for any successive bits in any word of the message (except the sync bit – see below). This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_MODE_CODE: This bit is set to indicate a mode code message.
- BT1553_INT_NOCMD: This bit is set when the decoder does not see any message on the bus after a BC command is sent. This is probably the result of an improperly terminated bus. The BT1553_INT_NO_RESP bit is set as well.
- BT1553_INT_NO_IMSG_GAP: This bit indicates that the mid-sync zero crossing of the next command sync was detected prior to 4μs preceding the mid-zero crossing of the parity bit of the last word of the current message. The next command sync may or may not produce a valid command word.
- BT1553_INT_NO_RESP: This bit indicates that the 1553 decoder did not detect a command sync within the specified No Response Timeout time when a status word is expected. This time is set with the wTimeout1 parameter to the BusTools_BC_Init function.
- BT1553_INT_NON_CONT_DATA: This bit indicates that a gap was detected between successive data words in the message. The hardware allows a 4-μs gap before declaring a Low Word error and beginning a search for the next command word.

- BT1553_INT_PARITY: This bit indicates that a parity error was detected in one or more words of the message. Odd parity is used (per the MIL-STD-1553 Specification). This condition also sets the BT1553_INT_INVALID_WORD bit.
- BT1553_INT_RESET_RT: This bit indicates that a valid Reset Terminal mode code command was received. The application must reset this RT to an initialized state.
- BT1553_INT_RETRY: This bit indicates that an automatic retry was executed by the BC. This bit doesn't indicate whether the retry was successful. Failure of the retry results in the No Response error bit set or the Message Error bit in the RTs status word being set.
- BT1553_INT_RT_RT_FORMAT: This bit indicates that the message is an RT¬ to RT message. It is detected in hardware by two consecutive words with a command sync.
- BT1553_RTRT_RCV_NRSP: This bit indicates which message on a RT→RT message is not responding. If BT1553_INT_NO_RESP is set for a RT→RT message check this bit. If set the receive command did not respond. If reset the transmit command did not respond.
- BT1553_INT_SELF_TEST: This bit indicates the reception of a Built-In-Test modecode command.
- BT1553_INT_TRIG_BEGIN: This bit indicates that the BM trigger enable condition was met in this message. BM message gathering begins at this time. This bit is generated by the API interrupt service function.
- BT1553_INT_TRIG_END: This bit indicates that the BM trigger disable condition was met in this message. BM message gathering terminates at this time. This bit is generated by the API interrupt service function.
- BT1553_INT_TWO_BUS: This bit indicates that both buses (Bus A and Bus B) were active sometime during the message. This is a 1553 protocol error.
- BT1553_INT_WRONG_BUS: This bit indicates that the RT responded on a different bus than the one on which the command word was transmitted. This is a 1553 protocol error.

7.12 Interrupt Queue Message Block Structure (F/W 5.x or earlier)

Code Definition

```
typedef struct iq_mblock
{
   union
   {
       BT_U16BIT      modeword;
       BT1553_INTMODE mode;
   } t;               // Interrupt mode/type bits
   BT_U16BIT msg_ptr; // msg that caused the interrupt
   BT_U16BIT nxt_int; // next interrupt in the queue
}
IQ_MBLOCK;
```

Description

This structure has all the data for a single entry in the interrupt queue. Read the interrupt queue data into this structure to determine the type and address of the message in the queue.

The interrupt queue is 296 records long. The firmware writes the data into the queue. The user application can process this data to read the messages. There are three entries in this structure, the interrupt mode, the message pointer, and pointer to the next interrupt queue entry.

Data Elements

Modeword/mode: This a union of the BT1553_INTMODE structure, see below and a 16 bit unsigned integer. The BT1553_INTMODE structure shows the source of the interrupt.

```
typedef struct bt1553_intmode
   {
#ifdef NON_INTEL_BIT_FIELDS
   BT_U16BIT unused:7;   // unused                    (MSB)
   BT_U16BIT bc_ctl:1;   // BC control interrupt     0x0100
   BT_U16BIT bm_swap:1;  // BM-Only Buffer Swap*     0x0080
   BT_U16BIT ext_trig:1; // External Trigger         0x0040
   BT_U16BIT bmtrig:1;   // bm trigger has occurred  0x0020
   BT_U16BIT bc:1;       // bc interrupt             0x0010
   BT_U16BIT bm:1;       // bm interrupt             0x0008
   BT_U16BIT rt:1;       // rt interrupt             0x0004
   BT_U16BIT timer:1;    // timer overflow or load   0x0002
   BT_U16BIT iack:1;     // interrupt acknowledge bit (LSB)
#else  /* INTEL-Compatable bit field ordering */
   BT_U16BIT iack:1;     // interrupt acknowledge bit (LSB)
   BT_U16BIT timer:1;    // timer overflow or load   0x0002
   BT_U16BIT rt:1;       // rt interrupt             0x0004
   BT_U16BIT bm:1;       // bm interrupt             0x0008
   BT_U16BIT bc:1;       // bc interrupt             0x0010
   BT_U16BIT bmtrig:1;   // bm trigger has occurred  0x0020
   BT_U16BIT ext_trig:1; // External Trigger         0x0040
   BT_U16BIT bm_swap:1;  // BM-Only Buffer Swap*     0x0080
   BT_U16BIT bc_ctl:1;   // BC control interrupt     0x0100
   BT_U16BIT unused:7;   // unused                    (MSB)
```

```
#endif   }
BT1553_INTMODE;
/* *BM-Only Buffer Swap for AR15-VPX only */
```

msg_ptr: Message pointer is the message that generated the interrupt. This pointer is to a BC, BM, or RT message block. The interrupt type in the BT1553_INTMODE structure determines what type of message block you read.

nxt_ptr: pointer to the next entry in the interrupt queue. The interrupt queue is a circular linked list, 296 entries long. Interrupt records are placed sequentially in to the queue, starting at the beginning of the queue, until the last queue entry is reached where it wraps around to the start. The nxt_ptr allows the user application to read the next queue position without having to check for the end-of-queue.

7.13 Interrupt Queue Message Block Structure (F/W 6.0)

Code Definition

```
typedef struct iq_mblock_v6
   {
      BT_U32BIT mode;          // interrupt mode
      BT_U32BIT msg_ptr;       // points interrupt message
 }IQ_MBLOCK_V6;
```

Description

This structure has data for a single entry in the interrupt queue. Read the interrupt queue data into this structure to determine the type and address of the message in the queue.

The F/W 6.0 interrupt queue is 512 records long. The firmware writes the data into the queue. The user application can process this data to read the messages. There are three entries in this structure, the interrupt mode, the message pointer, and pointer to the next interrupt queue entry.

Data Elements

mode: Contain the type of interrupt.

```
NO_INTERRUPT             0  No interrupt present
TTIMER_LOAD_INTERRUPT    1  Tag timer load interrupt (BC,RT,BM)
TRIGGER_IN_INTERRUPT     2  Trigger input interrupt (BC,RT)
BC_MESSAGE_INTERRUPT     3  BC Message interrupt (BC)
BC_CNTRLWD_INTERRUPT     4  BC control word interrupt (BC)
RT_MESSAGE_INTERRUPT     5  RT Message interrupt (RT)
BM_MESSAGE_INTERRUPT     6  BM Message interrupt (BM)
BM_TRIGGER_INTERRUPT     7  BM Message and trigger interrupt (BM)
MF_OVFL_INTERRUPT        8  Minor Frame overflow interrupt (BC)
BC_BSY_MFOVFL_INTERRUPT  9  BC busy MF overflow interrupt. (BC)
LP_MF_OVFL_INTERRUPT     10 BC LP aperiodic minor frame OVFL(BC)
HP_MF_OVFL_INTERRUPT     11 BC HP aperiodic minor frame OVFL(BC)
```

msg_ptr: Message pointer is the message that generated the interrupt. This pointer is to a BC, BM, or RT message block. The interrupt type in the mode determines what type of message block you read.

7.14 Interrupt Register/Filter/FIFO Structure (API_INT_FIFO)

Code Definition

```
#define MAX_FIFO_LEN  256 /* Size of the event FIFO (power of 2!) XXXXXX 64?*/
#if (MAX_FIFO_LEN - 1) & MAX_FIFO_LEN
#error The MAX_FIFO_LEN parameter is not a power of 2!
#endif

typedef struct api_int_fifo
{
       //*************************************************************************
       //  Parameters setup by the user before calling BusTools_RegisterFunction
       //*************************************************************************
   // Pointer to user interrupt thread function:
   BT_INT (_stdcall *function)(BT_UINT cardnum, struct api_int_fifo *pFIFO);
                  //  Function should return API_SUCCESS if thread is to
                  //  continue operation, any other value will cause
                  //  the thread and event object to be destroyed.
   int   iPriority;    // User-requested thread priority
   DWORD dwMilliseconds;  // Thread time-out interval in msec or INFINITE
   // Mask to request startup and shutdown notification:
   BT_INT  iNotification; // CALL_STARTUP if function to be called at creation
                          // of thread, CALL_SHUTDOWN if function is to be
                          // called upon destruction of thread.  "OR" both
                          // together to enable notification on both events.
                          // See "bForceShutdown" and "bForceStartup" below.

   // User variables; not referenced by the API:
   int     nUser[8];      // Spare variables for use by the user.
   void   *pUser[8];      // Spare variables for use by the user.

   //*************************************************************************
   //  Event filter structure.  A one "1' enables the specified event; when
   //   detected the API will place it in the FIFO and call the user function.
   //*************************************************************************
   // Top Level Notification Event Mask:
   BT_INT  FilterType;    // One or more EVENT_ definitions, "or'ed" together.
   // Event filter mask array.
   //                 rt tr  sa
   BT_INT  FilterMask[32][2][32];  // The bits in the word form the word count
                                   // mask; bit 0 is 32 words, bit 1 is 1
                                   // word, bit 2 is 2 words, etc.  Bit set
                                   // enables combination.

   //*************************************************************************
   //  Parameters setup by BusTools_RegisterFunction
   //*************************************************************************
   // Reason codes explain why user function is being called.
   //  If both bForceShutdown and bForceStartup are zero,
   //   function is being called to process events.  user:
   int     bForceShutdown;// 1 - Thread is being shutdown, -1 complete     (RO) $
   int     bForceStartup; // 1 - Thread is being started, 0 complete       (RO) $
   int     nPtrIndex;     // Index into API pointer table                  (RO) $
   BT_UINT cardnum;       // Card number associated with this thread.      (RO) $
   BT_UINT numEvents;   // Total number of events, including overflows     (RW) $
   BT_UINT queue_oflow; // Count incremented by API when FIFO overflows    (RW) $
```

```
   HANDLE  hEvent;      // Handle to event object                          (RO) $
   HANDLE  hkEvent;     // Kernel mode handle to event object              (RO) $
   HANDLE  hThread;     // Handle to thread                                (RO) $
   DWORD   lThreadId;   // ID of new user interrupt thread.                (RO) $

   //  Note that a "$" indicates that the API initializes this parameter.
   //*************************************************************************
   //  FIFO of events, to be processed by user function.  The API enters events
   //  at the head_index, then increments head_index.  The user function should
   //  compare head_index with tail_index.  If equal, the FIFO is empty and the
   //  user function should return to wait for more events (return API_SUCCESS).
   //
   //  If head_index != tail_index, then the user function should process the
   //  FIFO entry indexed by tail_index, increment tail_index by one and
   //  logically "AND" it with mask_index, saving the resulting value back into
   //  tail_index.
   //  The user function should then compare head_index with the updated value
   //  of tail_index, and if not equal, process the next entry and update the
   //  value of tail_index, etc., otherwise return "API_SUCCESS" to wait for
   //  more events.
   //*************************************************************************
   BT_INT  head_index;  // Index of element being added to queue (0->63)  (RO) $
   BT_INT  tail_index;  // Index of element to be removed from queue      (RW) $
   BT_INT  mask_index;  // Mask for wrapping head and tail pointers       (RO) $
   struct  BT_FIFO        // FIFO structure: events for user to process.       $
   {
      BT_INT event_type;  // EVENT_ definitions below.
      BT_INT buffer_off;  // Byte offset of message buffer which caused event
      BT_INT rtaddress;   // Terminal address of message
      BT_INT transrec;    // Non-zero if transmit message, zero for receive
      BT_INT subaddress;  // Subaddress of message
      BT_INT wordcount;   // Word count of message; 0-31; 0 indicates 32 words
                          //  unless mode code (then indicates mode code number)
      BT_INT bufferID;    // Buffer ID number or message ID number.
      BT_INT reserved2;   // Reserved for API.
   }
   fifo[MAX_FIFO_LEN];             // FIFO has exactly 64 entries.

   BT_U32BIT EventMask[32][2][32]; // The bits correspond to the int status words
   BT_U32BIT EventInit;
   CEI_MUTEX mutex;
   BT_UINT timeout;  //timeout ona timed wait 0=event - 1=timeout;
}
API_INT_FIFO;

#define CALL_STARTUP      0x0001   /* Thread created and initialized        */
#define CALL_SHUTDOWN     0x0004   /* Thread shutdown has been requested    */
#define USE_INTERRUPT_MASK 0x12345678
#define MAX_REGISTER_FUNCTION  64  /* Max number of registered functions    */

/*************************************************************************
*  Event filter specification values.  When specified event is detected
*  the API places it in the FIFO and calls the user function.
*************************************************************************/
#define EVENT_IMMEDIATE 0x000f
#define EVENT_EXT_TRIG  0x0001
#define EVENT_TIMER_WRAP 0x0002 /* Tag Timer overflow or discrete input    */
```

```
#define EVENT_RT_MESSAGE  0x0004 /* RT message transacted                 */
#define EVENT_BM_MESSAGE  0x0008 /* BM message transacted                 */
#define EVENT_BC_MESSAGE  0x0010 /* BC message transacted                 */
#define EVENT_BM_TRIG     0x0020 /* BM trigger event (start/stop)         */
#define EVENT_BM_START    0x0040 /* BM started (BusTools_BM_StartStop)V4.01*/
#define EVENT_BM_STOP     0x0080 /* BM stopped (BusTools_BM_StartStop)    */
#define EVENT_BC_START    0x0100 /* BC started (BusTools_BC_StartStop)    */
#define EVENT_BC_STOP     0x0200 /* BC stopped (BusTools_BC_StartStop)    */
#define EVENT_RT_START    0x0400 /* BC started (BusTools_RT_StartStop)    */
#define EVENT_RT_STOP     0x0800 /* BC stopped (BusTools_RT_StartStop)    */

#define EVENT_RECORDER    0x1000 /* BM recorder buffer has 64K or timeout  */
#define EVENT_MF_OVERFLO  0x2000 /* Minor frame timing overflow            */
#define EVENT_API_OVERFLO 0x4000 /* BM API Recorder buffer overflowed      */
#define EVENT_HW_OVERFLO  0x8000 /* BM HW Recorder buffer overflowed       */
#define EVENT_BC_CONTROL 0x10000 /* BC Control block (NOOP condition stop) */
#define EVENT_LP_MF_OVFL 0x020000 /* Low priority overflow interrupt       */
#define EVENT_HP_MF_OVFL 0x040000 /* High priority overflow interrupt      */
#define EVENT_BC_BSY_OVFL 0x080000 /* BC_busy minor frame overflow interrupt */
#define EVENT_BM_OVRFLW   0x100000 /* Bus Monitor overflow interrupt       */
```

Description

This structure contains all of the information needed by the API to filter and deliver interrupt notifications to a user-specified function. Be sure to look at the current version of this structure in the Busapi.h file for possible additions or changes.

Memory for this structure is user-allocated, initialized, and passed to the BusTools_RegisterFunction API function. The API creates a new thread and an event object, and links the new structure into its internal list of such structures.

When an enabled event occurs, the API interrupt function scans each registered structure to determine if the associated thread should be signaled. If the filter enables the event, an entry is added to the structure FIFO and the event object is signaled. This causes the thread to wake up and call the user function, passing it the card number and a pointer to the registered structure. The user function can then perform the processing that is required, being sure to process all of the entries in the FIFO (there might be more than one). When all of the entries in the FIFO have been processed (and the FIFO tail pointer updated) the user thread performs a return; it then waits in the API for the next event notification.

If the API attempts to write an event into the FIFO and finds it full, the event is discarded and the queue_oflow counter is incremented. The user thread may do what it wants with this counter/flag. The numEvents counter is always incremented, even if the event is discarded.

The event FIFO is implemented as a circular buffer, when the head and tail pointers are equal the FIFO is empty. When updating the tail pointer, the application should use the mask_index to wrap the tail pointer. The head and tail pointers are actually indexes into the FIFO array, with values from 0 - 63. The API only reads and does not write the tail pointer. Likewise, the application should only read and not write the head pointer to avoid race conditions when updating the pointers.

For more information on this structure, see the BusTools_RegisterFunction function.

Data Elements

See Busapi.h for the most current description of the data elements in this structure.

7.15 Playback Data (API_PLAYBACK)

Code Definition

```
typedef struct api_playback
{
  BT1553_TIME timeStart;      // Start Tag Time
  BT1553_TIME timeStop;       // Stop Tag Time
  BT_U32BIT messageStart;     // Starting message number
  BT_U32BIT messageStop;      // Ending message number
  BT_INT filterFlag;   // 0 no filter, 1 = time filter, 2 = message fil22ter
  BT_U32BIT activeRT;  // Active RTs 0 = do not playback; 1 = use in playback
  BT_INT Subadress31Flag; // 0 = SA 31 is not a mode code, 1 = SA31 is mode code
  BT_INT Rt31Flag;     // 0 = RT 31 is not broadcast, 1 = RT 31 is broadcast
  API_PLAYBACK_STATUS *status; // Status structure to monitor playback.
  Char *fileName;      // Log file name
} API_PLAYBACK;
```

Description

This structure contains all information required by BusTools_Playback to execute the bus playback function. The caller to BusTools_Playback fills in the data and passes this structure as an argument in the calling sequence to BusTools_Playback. If the filter flag is set to zero, no data is required for the messageStart, messageStop, timeStart, and timeStop. The fileName pointer should include the path and filename of the Bus Monitor recorded .bmd file. BusTools_Playback does not validate the messageStart, messageStop, timeStart, and timeStop data. It is up to the calling function to ensure that these values are consistent and within the range of values of the Bus Monitor record (.bmd) file specified by *fileName*.

Data Elements

timeStart: The start time tag that BusTools_Playback uses when the filterFlag is set to 1. It is up to the calling function to ensure that timeStart is within the range of time tag in the .bmd file.

timeStop: The stop time tag that BusTools_Playback uses when the filterFlag is set to 1. It is up to the calling function to ensure that timeStop is within the range of time tag in the .bmd file and that the timeStop is greater than timeStart.

messageStart: This is the message start value used when filterFlag is set to 2. The Bus Monitor numbers each bus message starting at 1 and incrementing for each new message recorded. It is up to the calling function to ensure the messageStart is within the range of message numbers in the .bmd file.

messageStop: This is the message stop value used when filterFlag is set to 2. The Bus Monitor numbers each bus message starting at 1 and incrementing for each new message recorded. It is up to the calling function to ensure the messageStop is within the range of message numbers in the .bmd file and is greater than messageStart. To playback a single bus message, messageStart and messageStop should be equal.

filterFlag: This flag indicates the type of message filtering required. Either message time tag or message number can be used as a filter criterion. When set to 0, no filter is requested and the messageStart, messageStop, timeStart, and timeStop values are ignored. When filterFlag is set to 1, the timeStart and timeStop values are used by BusTools_Playback to filter using message tag time to filter the .bmd file messages. When filterFlag is set to 2, the messageStart, and messageStop values are used by BusTools_Playback to filter using message number.

activeRT: This is a 32-bit encoded value, with each bit corresponding to a remote terminal address. If the corresponding bit is set (1), that RT is included in the bus playback. If the bit is reset (0), that RTs responses is filtered out of the bus playback.

subadress31Flag: Indicates to BusTools_Playback whether Subaddress 31 is a mode code. 0 = SA 31 is not a mode code, 1 = SA31 is mode code.

rt31Flag: Not used. RT Address 31 is always a broadcast message.

status: Structure address passed to BusTools_Playback so the calling function can monitor the status of the bus playback thread. See Playback Status (API_PLAYBACK_STATUS) for a complete structure definition. A call to BusTools_Playback causes a separate thread of execution to start for bus playback. The BusTools_Playback function returns with an indication of whether or not bus playback started successfully. API_SUCCESS means that a bus playback thread is executing. The status structure allows the calling function to determine the status of bus playback during execution and when bus playback is done.

fileName: The name of the Bus Monitor record file (.bmd or .bmdx) used for bus Playback. File name must include the path if the .bmd file is not located in the default directory.

7.16 Playback Status (API_PLAYBACK_STATUS)

Code Definition

```
typedef struct api_playback_status
{
  BT_U16BIT    tailPointer;      // Playback BufferTail Pointer
  BT_U16BIT    playbackStatus;   // Playback status
  BT_INT        playbackError;   // Playback Error
  BT_U32BIT    recordsProcessed; // Number of record processed
                                 // by BusTools_Playback
} API_PLAYBACK_STATUS;
```

Description

This structure contains bus playback status information. Since bus playback runs a separate thread of execution, the calling function needs to monitor bus playback through the data in this structure. The bus playback thread updates the values periodically.

Data Elements

tailPointer: Playback uses a circular buffer to store messages. The playback function moves the tail pointer as it processes the data in the buffers. The tail pointer should change periodically. If the tail pointer value remains constant for a period, this may indicate that the bus playback has been abnormally halted. It is up to the function monitoring the bus playback status to determine what constitutes an excessive time for the tail pointer to remain constant. Normal 1553 traffic would have several messages every second, so periods greater than a few seconds would indicate a problem.

playbackStatus: The 1553 playback function provides a status register. BusTools_Playback reads this status register periodically and reports the results in playbackStatus. The table below describes the contents:

Table 7-1 Playback Status Bits

Bit(s)	Function	Description
5	Halt Bit	Set to halt playback.
4	Buffer empty bit	Set when the playback tail pointer and playback head pointer are equal.
3	Error bit	Set by the firmware if an error is encountered (i.e., invalid message code).
2	Run bit	Set by the firmware once playback begins transmitting. Cleared at the end of playback or by calling BusTools_Playback_Stop.
1	Not used	Not used
0	Start bit	Set only by host. This bit is cleared at the end of playback or by calling BusTools_Playback_Stop.

PlaybackError: This variable reports any error detected by the playback thread during playback execution.

recordsProcessed: This variable shows the number of messages processed by BusTools_Playback. This number represents the processing done by the host system not the 1553 device, so it can't be used to indicate the number of records actually played over the 1553 bus. However, at the successful end of playback, this value should equal the number of messages in the bus monitor file or the subset of the file played back.

7.17 RT Address Control Block (API_RT_ABUF)

Code Definition

```
typedef struct api_rt_abuf
{
   WORD enable_a;      // enable channel a
   WORD enable_b;      // enable channel b
   WORD inhibit_term_flag;  // inhibit terminal flag
   WORD status;        // latest status word
   WORD command;       // latest command word (read only)
   WORD bit_word;      // latest built in test word
}
API_RT_ABUF;

// Define the bits in the inhibit terminal flag control word:
#define RT_ABUF_ITF        0x0004 // Set this bit to inhibit terminal flag
#define RT_ABUF_DBC_ENA    0x4000 // Set this bit to enable Dynamic Bus Acceptance
#define RT_ABUF_DBC_RT_OFF 0x8000 // Set this bit to shut down RT on valid DBA
```

Description

This structure contains control information that must be specified and can be changed for each RT defined on the Abaco Systems 1553 board. The most important elements of this structure are the enable flags. These flags control whether or not the BusTools microcode processes any messages for this RT. In fact, if both enable flags are set to "0", then the specified RT is "off".

Data Elements

enable_a: This flag indicates whether the RT should respond to messages on the primary bus (Bus A). If this flag is off, the specified RT ignores messages on Bus A.

enable_b: This flag indicates whether the RT should respond to messages on the secondary bus (Bus B). If this flag is off, the specified RT ignores messages on Bus B.

inhibit_term_flag: This is a multi-function flag word. The bit defined by the constant "RT_ABUF_ITF" controls the reporting of the Terminal Flag within the message status word. If this flag is set, the Terminal Flag isn't set in the message status word. If this flag is not set, the actual state of the Terminal Flag (in the status word defined below) is reported. This flag should be initialized by the application. Once the RT is operating, this flag is modified as required by the microcode in response to the Inhibit Terminal Flag Bit and Override Inhibit Terminal Flag Bit modecode messages.

The bit defined by the constant "RT_ABUF_DBC_ENA" controls enabling Dynamic Bus Control (DBC) mode codes. When this bit is set, receipt of a DBC by the RT causes the Bus Controller to start in accordance with MIL-STD-1553 paragraph 4.3.3.5.1.7.1. Mode Code 0 must have been legalized by a call to BusTools_RT_AbufWrite (enable word count 0, transmit, for subaddress 0

and optionally subaddress 31).

The bit defined by the constant “RT_ABUF_DBC_RT_OFF” controls the operation of the RT when a DBC mode code is received and accepted by the RT. If the board is a single-function or dual-function board, acceptance of a DBC mode code causes the RT to be turned off before the BC is turned on (the application must have previously setup but not started the bus controller). If the board is a multi-function board, the RT is NOT turned off; however, if this bit is set then this one RT is turned off, if this bit is clear all RTs continue to run. “RT_ABUF_EXT_STATUS” enable extended status mode, allowing differnet status response on RT/SA/TX/RX combinations.

status: This is the latest message status word. It should be initialized by the application. Once the RT is operating, this entry is modified as required by the microcode. The microcode modifies the Message Error, Broadcast Command Received, Dynamic Bus Acceptance and Terminal Flag bits as required by 1553 message traffic and the 1553 specification.

command: This is the command word contained in the last message targeted at this RT. It is maintained by the microcode in order to respond to the Transmit Last Command Word modecode message. Typically, the application need not initialize this entry, nor does it care what is contained in this entry.

bit_word: This is the Built-In-Test word. It can be initialized by the application to be any value. It is used by the microcode in order to respond to the Transmit BIT Word modecode message. The microcode does not modify this word at any time.

7.18 RT Control Buffer (API_RT_CBUF)

Code Definition

```
typedef struct api_rt_cbuf
{
   DWORD legal_wordcount;   // legal word count bits
}
API_RT_CBUF;
```

Description

The RT Control buffer is a structure that controls the operation of a single RT subunit (a single RT address /subaddress combination for transmit or receive operations). It is a 32-bit value where each bit enables (= 1) or disables (= 0) a wordcount for that subunit. See the table below for sample wordcounts.

This version of the RT Control buffer controls both normal RT subunit messages and RT modecode operations. Modecodes are subunits with a subaddress of 0 and optionally 31. The structure is identical for Modecodes except that the bit assignments for wordcount are replaced with bit assignments based on the modecode value.

This version of the RT Control buffer is *not* used for broadcast RT addresses (address 31 if broadcast operations are enabled). There is a different structure used for broadcast addresses (see RT Control Buffer for Broadcast (API_RT_CBUFBROAD)).

Data Elements

legal_wordcount: This is a 32-bit value containing one bit for each possible wordcount for messages to this RT `subunit`. If the bit is set, messages with the corresponding wordcount are allowed for this RT `subunit`. If the bit is not set, messages with the corresponding wordcount are not allowed for this RT `subunit`. The bits in this value are assigned as follows:

Bit number	Enabled Word Count	legal_wordcount (hexadecimal)
0	32	0x00000001
1	1	0x00000002
2	2	0x00000004
3	3	0x00000008
…	…	…
29	29	0x20000000
30	30	0x40000000
31	31	0x80000000

7.19 RT Control Buffer for Broadcast (API_RT_CBUFBROAD)

Code Definition

```
typedef struct api_rt_cbufbroad
{
   DWORD legal_wordcount[31]; // legal word count bits for each sub address
   BT_U32BIT mbuf_count;
}
API_RT_CBUFBROAD;
```

Description

The RT Control buffer for Broadcast is a structure controlling the operation for all RT address and word count combinations in a broadcast command to a specified subaddress. There are two components of an RT Control buffer for Broadcast:

- an array of bit fields which enable and/or disable broadcast operations to the specified RT subaddress. The array of bit fields must be supplied by the application.
- a pointer to the first RT Message buffer used for broadcasting to this RT subaddress.

The RT Message buffer address is managed by the BusTools API functions (thus, it is not included in the RT Control buffer for Broadcast structure definition).

To fully control all possible broadcast message combinations, 32 copies of this structure must be stored in channel memory. However, RT subaddresses, which share a common set of enable/disable parameters, can share a common Control buffer. The BusTools API functions define a default Control buffer with all enable bits set. Initially, all subaddresses use this default structure to control broadcast commands. If this setting is acceptable, then the application need not supply any additional RT Control buffer for Broadcast structures.

Data Elements

legal_wordcount: This is an array of 32-bit values, with each value controlling broadcast operations to a specific RT address. Each bit within the value enables or disables broadcast messages with a specific word to the RT address. Bit values are assigned as follows:

Bit number	Enabled Word Count	legal_wordcount (hexadecimal)
0	32	0x00000001
1	1	0x00000002
2	2	0x00000004
…	…	…
30	30	0x40000000
31	31	0x80000000

mbuf_count: Set the number of broadcast buffers for that subaddress.

7.20 RT Message Buffer (read-only) (API_RT_MBUF_READ)

Code Definition

```
typedef struct api_rt_mbuf_read
{
   DWORD           status;           // interrupt status
   BT_U32BIT       reserved;         //
   BT1553_TIME     time;             // time

   // contents of real message to/from bus
   BT1553_COMMAND  mess_command;     // command word
   BT1553_STATUS   mess_status;      // status
   BT_U16BIT       mess_data[32];    // data
   BT_U16BIT       spare;            // spare data word for hi-word errors
}
API_RT_MBUF_READ;
```

Description

This structure contains all of the information read from a RT Message buffer. All elements of this structure and, except for data values, contain valid information only after a message has been processed for this RT subunit (RT address, subaddress, and transmit / receive flag combination). For Message buffers associated with transmit RT subunits, the data values are those originally supplied by the application. For Message buffers associated with receive RT subunits, the data values are those received from the bus and are only valid after a message has been received.

Data Elements

status: This is the Message Status bits associated with this particular message. The bit assignments are shown in Section 7.11, "Interrupt Enable / Message Status Bits (32 bit)".

time: This is the time at which this message was transmitted / received. The time is measured from the start of the BusTools (typically when the Bus Monitor function was first started). The time is accurate to microseconds.

mess_command: This is the message command word as received from the Bus Controller.

mess_status: This is the message status word as supplied by the Remote Terminal for this command.

mess_data: This is the actual data transmitted / received during the command. Use the wcount field of the message command word to determine the number of valid data words contained in the message.

7.21 RT Message Buffer (write-only) (API_RT_MBUF_WRITE)

Code Definition

```
typedef struct api_rt_mbuf_write
{
   DWORD enable;       // interrupt enable bits
   WORD error_inj_id;  // id of error injection buffer
   WORD mess_data[32]; // data
}
API_RT_MBUF_WRITE;
```

Description

This structure is used to initialize a RT Message buffer. You must supply all the contents this structure for transmit messages. For receive messages you only need to supply the interrupt enable bits and the error injection pointer.

Data Elements

enable: This word contains the Interrupt Enable Bits for this message. If any of the conditions specified by the bits in this parameter are met when the message is received / transmitted, an interrupt block is added to the Interrupt Queue. For bit definitions, see Section 7.11, "Interrupt Enable / Message Status Bits (32 bit)".

error_inj_id: The Error Injection buffer number used with this message. For transmit messages, this should be a buffer that contains an entry for the status word and each possible data word for this message. For receive messages, this should be a buffer that contains an entry for only the status word.

mess_data (required only for transmit messages): This is the actual data transmitted. The application is responsible for ensuring the number of data elements supplied here matches and the word count requested by the BC.

7.22 Time Structure (BT1553_TIME)

Code Definition

```
typedef struct bt1553_time
{
   BT_U32BITµs;           // µs since start
   BT_U16BITtopuseconds; // Most significant part of µs
}
BT1553_TIME;   // Note that this is a 48-bit value for F/W 5.x or earlier

typedef struct bt1553_time
{
   BT_U32BITµs;           // µs since start
   BT_U32BITtopuseconds; // Most significant part of µs
}
BT1553_TIME;   // Note that this is a 64-bit value for F/W 6.0
```

Description

This structure describes time within the BusTools API. Time is measured from the beginning of the operation. For F/W 5.x and earlier it's in microseconds. For F/W 6.0 it's in nanoseconds.

Data Elements

microseconds: Number of µs/ns since the start.

topuseconds: MSB of µs/ns since the start.

7.23 Device Mapping(DEVMAP_T)

Code Definition

```
typedef struct _DEVMAP_T {
   int        busType;                    // One of BUS_TYPE.
   int        interruptNumber;            // Interrupt number.
   int        memSections;                // Number of memory regions defined.
   int        flagMapToHost[MAX_MEMORY]; // Set to map region into host space.
   char       * memHostBase[MAX_MEMORY]; // Base address of region in host space.
   unsigned   memStartPhy[MAX_MEMORY];   // Physical base address of region.
   unsigned   memLengthBytes[MAX_MEMORY]; // Length of region in bytes.
   int        portSections;               // Number of I/O port regions.
   unsigned   portStart[MAX_PORTS];       // I/O Address of first byte.
   unsigned   portLength[MAX_PORTS];      // Number of bytes in region.
   int        llDriverVersion;            // Low Level driver version.
   int        KernelDriverVersion;        // Kernel driver version.
   HANDLE     hKernelDriver;              // Handle to the kernel driver.
   unsigned int VendorID;                 // Vendor ID if PCI card
   unsigned int DeviceID;                 // Device ID if PCI card
   int          device;                  // This the device
   int          use_count;               // Number of user channels
#ifdef _USE_BM_DMA
   unsigned *   vaddr;                   //
   CEI_UINT64   laddr;                   // local DMA addres
#endif
   int          use_channel_map;         //
   int          mapping;                 // Mapping option 0 - physical 1 - virtual
} DEVMAP_T, *PDEVMAP_T;
```

Description

This structure is used to get device mapping information from the low level mapping function vbtGetBoardAddresses. This structure is available only through lowlevel.h. This description is to assist porting the API to currently non-supported operating systems.

This structure is used by all operating systems supported by BusTools/1553-API. This structure is used by all Abaco Systems PCI, ISA, and PCMCIA boards, but not VME boards. Not all elements in the structure are used by each O/S. The Data element description outlines the O/S use. Not all elements are currently in use but are maintained for compatibility.

Data Elements

busType: (All) This is the bus value based on the following enum:

```
typedef enum _BUS_TYPE     /* Host bus types supported
      */
{
    BUS_INTERNAL,
    BUS_ISA,
    BUS_PCI,
    BUS_VME,
    BUS_PCMCIA,
```

```
        BUS_OTHER
    } BUS_TYPE;
```

interruptNumber: (Windows) interrupt number returned form the low-level mapping

memSections: (All) The number of memory sections (PCI BAR regions) mapped. For ISA and PCMCIA this is always equal to 1.

flagMapToHost (All) Set by the mapping function to specify which memory regions are mapped.

memHostBase: (All) The mapped base address of the device. For flat mapped system this is the same as the physical address, otherwise the virtual memory address for the device.

memStartPhys: (All) The starting physical address of the device. For ISA devices this is passed. For PCI devices this is the address stored in the BAR register for each bar region mapped.

memLengthBytes: (All) The length in bytes mapped for the device. This has a minimum of a page size.

portSections: Not used

portStart: Not Used.

portLength: Not Used.

llDrvierVersion: Not Used

KernelDriverVersion: Windows only driver version.

hKernelDriver: Windows only.

VendorID: (All) The PCI vendor ID 0x13c6 for all Abaco Systems PCI boards. Not used for ISA and PCMCIA.

DeviceID: (All) The PCI Device ID. The value varies depending on the device. Not used for ISA and PCMCIA

device: The device number. 0 – MAX_BTA.

use_count: The number of channels currently open on the device. This value is incremented and decremented each time a channel on the device is opened or closed. This value keeps track of the device used. When all channels are closed (use_count=0) the device is un-mapped.

7.23.1 Device Information (DEVICE_INFO)

Code Definition

```
typedef struct   DEVICE_INFO {
   int          busType;         // One of BUS_TYPE.
   int          nchan;           // number of channeld.
   int          irig;            // IRIG option 0=no 1=yes.
   int          mode;           // mode 0=single 1=multi.
   int          memSections;     //Number of memory section
   unsigned int VendorID;        // Vendor ID if PCI card
   unsigned int DeviceID;        // Device ID if PCI card
} DEVICE_INFO;
```

Description

This structure contains information about the type of device and the features and channels available.

Data Elements

busType: This is the bus value based on the following enum:

```
typedef enum _BUS_TYPE     /* Host bus types supported */
{
    BUS_INTERNAL,
    BUS_ISA,
    BUS_PCI,
    BUS_VME,
    BUS_PCMCIA,
    BUS_OTHER
} BUS_TYPE;
```

nchan: The number of channel available on the device (1 - 4).

IRIG: 1 = IRIG capable; 0 = No IRIG support.

mode: Operation mode of the board. 0 = single - function; 1= multi-function.

memSections: The number of memory sections (PCI BAR regions) mapped. For ISA and PCMCIA this is always 1.

VendorID: The PCI vendor ID 0x13c6 for all Abaco Systems PCI boards. Not used for ISA and PCMCIA.

DeviceID: The PCI Device ID. Varies depending on the device. Not used for ISA and PCMCIA.

A • Sample Programs

A.1 List

This section lists the example programs and their descriptions. It also provides the BusTools/1553-API functions used in each sample program. Be sure to check in this manual for the applicability of each function to your operating system. Not all API calls are available for all operating systems. Each function description lists Operating System Support. The default device ID for the sample program is set to 0.

Table A-1 Sample Programs

Example Name	Descriptions	BusTools/1553-API
example_auto_sync_mode.c	This example shows how to setup the time tag auto sync mode. This mode synchronizes the time tag to an external pulse. The example also sets up a dead man timer to detect if the external 1PPS pulse does not occur on time. There is optional code to have the 1553 interface board generate the 1PPS pulse instead of external 1PPS.	BusTools_API_OpenChannel, BusTools_API_Close, BusTools_SetInternalBus, BusTools_BoardIsV6, BusTools_SetV6TrigIn, BusTools_SetV6TrigOut, BusTools_DiscreteSetIO, BusTools_DiscreteTriggerOut, BusTools_DiscreteTriggerIn, BusTools_TimeTagMode, BusTools_ExtTrigIntEnable, BusTools_BM_StartStop, BusTools_BC_StartStop, BusTools_RT_StartStop, BusTools_RegisterFunction, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_MessageRead, BusTools_TimeGetString, BusTools_DiscreteWrite, BusTools_RT_Init, BusTools_RT_CbufWrite, BusTools_RT_MessageWrite, BusTools_RT_AbufWrite, BusTools_BM_Init, BusTools_BM_MessageAlloc

Example Name	Descriptions	BusTools/1553-API
example_bcrt_2ch.c	This is a 2 CHANNEL example program that sets up one channel as a Remote Terminal and another channel as Bus Controller on a multi-channel board. The first channel is setup as RT1 with two subaddresses, SA1 RECEIVE and SA2 TRANSMIT. BusTools_RegisterFunction is called to implement a user callback function to process the RT transmit message data. The second channel is setup as BC with a list of two messages, 1-R-1-32 and 1-T-2-32, in a 500ms minor frame. This BC list runs until stopped by user input. User can either display the data for the 1-T-2-32 message or quit. A interrupt callback function is setup by calling BusTools_RegisterFunction. The rt_intFunction displays the RT data. The user can also switch between Transformer or direct couple or dump the channels memory contents through command-line input. This test requires a multi-channel board and either a physical connection between channel 1 and channel 2 or if you are using a QPCX-1553 or QPCI-1553 you can use the test bus.	BusTools_API_OpenChannel, BusTools_API_Close, BusTools_SetInternalBus, BusTools_TimeTagMode, BusTools_GetBoardType, BusTools_SetTestBus, BusTools_SetVoltage, BusTools_RT_StartStop, BusTools_BC_StartStop, BusTools_RegisterFunction, BusTools_RT_MessageRead, BusTools_TimeGetString, BusTools_RT_Init, BusTools_StatusGetString, BusTools_RT_AbufWrite, BusTools_RT_CbufWrite, BusTools_RT_MessageWrite, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite,
example_bc_1Shot.c	This console application shows how to setup BC messages that transact just once and then stop the Bus Controller. This setup is referred to as a 1Shot message frame. Three one shot frames are created and BusTools_BC_Start is used to start the BC at the beginning one of the selected bus list.	BusTools_API_OpenChannel, BusTools_API_Close, BusTools_SetInternalBus, BusTools_TimeTagMode, BusTools_RT_StartStop, BusTools_BC_Start, BusTools_RegisterFunction, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_MessageRead, BusTools_RT_Init, BusTools_RT_CbufWrite, BusTools_RT_MessageWrite, BusTools_RT_AbufWrite, BusTools_RegisterFunction
example_bc_aperiodic.c	This example shows how to setup and run aperiodic messages. This example sets up a periodic bus list running at 1hz and two aperiodic bust lists. The user can command either of the aperiodic list to run at low or priority or high priority.	BusTools_API_OpenChannel, BusTools_API_Close, BusTools_SetInternalBus, BusTools_BC_AperiodicRun, BusTools_BC_StartStop, BusTools_RegisterFunction, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_MessageRead

Example Name	Descriptions	BusTools/1553-API
example_bc_auto_incr.c	This example sets up the Bus Controller, Remote Terminal and Bus Monitor and then enables auto-increment on the Bus Controller message buffer 3. BusTools_BC_AutoIncrMessageData is used setup the increment process. Message buffer 3 contains a 32-word receive command to RT4 SA4. The last word in the command, data word word 32 increments each time the message transacts.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BM_Init BusTools_BM_MessageAlloc BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_AutoIncrMessageData BusTools_BM_StartStop BusTools_RT_StartStop BusTools_BC_StartStop BusTools_DumpMemory BusTools_RegisterFunction BusTools_BC_MessageRead BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_branch_on_data.c	This application shows how to setup Message Scheduling, Frame-Start-Timing and Conditional Branch 2. Conditional Branch 2 branches based on the value in an onboard memory location allocated using BusTools_MemoryAlloc. The application writes values to this location to controlwhich conditional messages transactions. There are 10 conditional messages. Each is run based on a different bitwise (1,2,4,8...) value written to the allocated memory address. By using the data_mask and data_value the application can send any combination of conditional messages by entering a value between 1 and 0x3ff. That value is written to the allocated memory location. The Conditional Branch message evaluate the data at that test address to determine whether to branch. This is an alternate approach to apperiodic messaging, The application can select any combination of messages to run after the periodic messages transact. Unlike aperiodic messages you need to ensure there is enough time in the frame after the periodic messages for the selected conditional messages to run. All messages run in the current frame. After each conditional message runs the corresponding bit at the test address is cleared so the message only runs once per data update. **Note**: This example run on both the new V6 firmware and older V4/V5 firmware. In the new V6 firmware allocate 32 bit of memory rather than 16 bits used in the older firmware.	BusTools_API_OpenChannel, BusTools_API_Close, BusTools_SetInternalBus, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_MemoryAlloc32, BusTools_MemoryWrite2, BusTools_MemoryAlloc, BusTools_MemoryWrite, BusTools_BoardIsV6, BusTools_BC_StartStop, BusTools_DumpMemory, BusTools_RegisterFunction, BusTools_BC_MessageRead,

Example Name	Descriptions	BusTools/1553-API
example_bc_branch_on_status.c	This example program demonstrates the use of Conditional Branches. There are several options for branching. In this example there are two branches used, CONDITIONAL BRANCH and CONDITIONAL BRANCH 3. CONDITIONAL BRANCH branches on data in the immediately preceding message in the bus list, either command word, status word or data. CONDITIONAL BRANCH 3 branches the same values, but the message buffer is specified in the setup. In this example, 8 message buffers are created. Message 1 is a CONDITIONAL BRANCH that branches when the Status Request bit (SRQ) is set in the status response on message 0, the immediately preceding message. If the SRQ is set, then Msg 2 transact otherwise it is skipped. Message buffer 6 is a CONDITIONAL BRANCH 3 that branches if the data in the last data word of Msg 4 is 0xabcd. If the data is set to 0xabcd then Msg 7 transact, otherwise it is skipped.	BusTools_API_OpenChannel, BusTools_API_Close, BusTools_SetInternalBus, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_StartStop, BusTools_BC_MessageRead, BusTools_TimeGetFmtString, BusTools_RegisterFunction

Example Name	Descriptions	BusTools/1553-API
example_bc_cond_count.c	This application shows how setup a BC messages and use conditional branch on count to run a specific number of frames. There is also a branch on data in a previous message. The BC sets up message buffer 1 as a Conditional Branch 3. That branch looks at the data in the third data word of the previous message. If the third data word equals 0xA002, the branch skips the next message. In this example that condition is always TRUE and the next message in the bus list, RT3 SA3 WC3 TX is never transmitted. You can alter the data and see how that affects the branch. Message buffer 5 is setup as a Conditional Branch 2 that is set to branch on the 10th occurrence. Because of the way the firmware decrements the branch count, it is set to n-1. In this case count is set to 9 (10-1). The count is decremented each time the branch executes. When it hits to zero the branch occurs. In the example the non-branching false condition is a nop message setting the end-of-frame with the next-message pointing to the frame start at message buffer 0. When the branch does execute, the message buffer contains a BC LAST command that stops the Bus Controller. This setup executes the frame 10 times then halts the BC. You change the two count values and see how it alters the execution.	BusTools_API_OpenChannel, BusTools_SetInternalBus, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_StartStop, BusTools_BC_MessageRead, BusTools_BC_IsRunning2, BusTools_RegisterFunction
example_bc_data_transfer.c	This example program sets up a Bus Controller to transfer a block of data to an RT. This approach could be used to transfer any amount of data; in this case 1024 words are transferred. An interrupt event signals when the BC message completes so we can write the next buffer of data to the BC message buffer. A Conditional Branch on count is used to transfer 32 messages of 32 words (32 x 32 = 1024). When the transfer is complete the Bus Controller halts, and the example exits. This program also uses BusTools_BC_MessageUpdate to update only the data portion of the Bus Controller message buffer.	BusTools_API_OpenChannel, BusTools_SetInternal, Bus, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_StartStop, BusTools_BC_MessageRead, BusTools_BC_MessageUpdate, BusTools_RegisterFunction, BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_data_wrap.c	This example shows how to wrap transmit and receive buffer so that data from a BC transits command can send the data from a BC receive command. In this example the data buffer from the first message (0) is linked to the second message (1) by reading the data buffer address in buffer 0, the transmit command, and over-writing the data buffer address in the second buffer (1) with the transmit buffer address. That way the data from the transmit command is used by the receive command.	BusTools_API_OpenChannel, BusTools_SetInternalBus, BusTools_TimeTagMode, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_MessageGetaddr, BusTools_BoardIsV6, BusTools_MemoryRead2, BusTools_MemoryWrite2, BusTools_MemoryRead, BusTools_MemoryWrite, BusTools_BC_StartStop, BusTools_RegisterFunction, BusTools_BC_MessageRead, BusTools_TimeGetFmtString, BusTools_API_Close
example_bc_deadman_timer.c	This example shows how to configure a watchdog timer to stop the Bus Controller if the host does not reset the timer. This example uses a Condition Branch on count (CONDITION2) as the timer. The Conditional Branch is setup to branch when the count decrements to 0. The count value is decremented each time the branch executes. The branch command is a BC_CONTROL_HALT that stops the Bus Controller. Otherwise the frame keeps running. To keep the branch from ever executing the halt, the host must reset the counter value to prevent it from reaching 0. To do this, the example gets the address of the BC buffer containing the branch command and resets the count value by writing the initial count value back into the count register thus restarting the down-count. This example runs until the user enters the kill command from the console. Then the count value is no longer reset, and the Bus controller halts. Two callback functions are used. One for Bus Controller messages prints the message data to the console. The other is for the conditional branch. This callback resets the counter value. See how altering the count value affects the timer.	BusTools_API_OpenChannel, BusTools_SetInternalBus, BusTools_TimeTagMode, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_MessageGetaddr, BusTools_MemoryRead2, BusTools_BoardIsV6, BusTools_BC_StartStop, BusTools_RegisterFunction, BusTools_BC_MessageRead, BusTools_TimeGetFmtString, BusTools_MemoryWrite2, BusTools_MemoryWrite

Example Name	Descriptions	BusTools/1553-API
example_bc_error_inj.c	This example program sets up a Bus Controller message list containing two messages, 1-R-1-32 and 2-T-2-32, in a 20ms minor frame. This BC list runs until stopped by user input. Error are injected into the command or data words of the two messages. The example injects a PARITY error on the command word either for 1-R-1-32 or 2-T-2-32. It also injects a parity error into the first data of the receive command. Notice the effect of the error by monitoring with an external Bus Analyzer like BusTools/1553. When a parity error is injected on a command word it becomes an invalid command the Analyzer (monitor) ignores the message. If error is injected on the data word, then the command is valid and the monitor will show errors like invalid word and parity error.	BusTools_API_OpenChannel, BusTools_SetInternalBus, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_StartStop, BusTools_EI_EbufWriteENH, BusTools_API_Close
example_bc_join.c	Channel Sharing allows multiple applications to run on a single 1553 channel. Channel sharing requires that only one application initialize a channel. That application must share the channel by calling BusTools_API_ShareChannel. Other application can join the shared channel by calling BusTools_API_JoinChannel. There can be only one Bus Controller application, one Bus Monitor application and one Remote Terminal application per channel. This example Shows a BC application joining an already initialized channel.	BusTools_API_JoinChannel, BusTools_SetInternalBus, BusTools_TimeTagMode, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_MessageRead, BusTools_TimeGetString, usTools_RegisterFunction, BusTools_BC_StartStop, BusTools_API_QuitChannel
example_bc_message_read_types.c	This example program shows how to program the Bus Controller to run multiple minor frames and to demonstrate programming all of the message types available to the Bus Controller. The example initializes the channel and sets it for external bus. Following those initialization steps, it initializes the Bus Controller and builds two minor frames. There are examples of each of the BC message in the frames.	BusTools_API_OpenChannel, BusTools_TimeTagMode, BusTools_SetInternalBus, BusTools_SetBroadcast, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_MessageRead, BusTools_BoardIsV6, BusTools_TimeGetFmtString, BusTools_RegisterFunction, BusTools_BC_StartStop, BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_message_types.c	This example program shows how to program the Bus Controller to run multiple minor frames and to demonstrate programming all of the message type available to the Bus Controller. The example initializes the Bus Controller channel. Then it sets the channel for external bus. Following those initialization steps, it initializes the Bus Controller and builds two minor frames. There are examples of each of the BC message in the frames.	BusTools_API_OpenChannel, BusTools_SetInternalBus, BusTools_SetBroadcast, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BoardIsV6, BusTools_BC_StartStop, BusTools_API_Close
example_bc_msg_blk_read.c	This example program shows how to program the Bus Controller to run a simple three message minor-frame. The example initializes the channel the Bus Controller is running on. Then sets the channel for external bus. Following those initialization steps, it initializes the Bus Controller and builds a simple 3-message frame. The messages in the frame are linked together and the frame will run continually until stopped by the user. This example also shows how to process the messages using the function BusTools_BC_ReadLastMessageBlock. This function parses the interrupt queue to determine the BC messages that have transacted since the last call. In order for a BC message to record it must have the BC_CONTROL_INTERRUPT set and the BC initialization must define an interrupt condition like BT1553_INT_END_OF_MESS. This processing differs from BusTools_RegisterFunction in that your application must provide a timing loop to call BusTools_BC_ReadLastMessageBlock periodically. The timing must set so the BC messages (and other messages that may be recorded in the interrupt queue) do not overwrite unprocessed entries. This example uses a 50-millisecond delay. This application uses the Windows function kbhit to break out of timing loop. If using this example on non-Windows systems, you will need to provide a kbhit function.	BusTools_API_OpenChannel, BusTools_SetInternalBus, BusTools_TimeTagMode, BusTools_BC_Init, BusTools_BC_MessageAlloc, BusTools_BC_MessageWrite, BusTools_BC_StartStop, BusTools_BC_ReadLastMessageBlock, BusTools_BoardIsV6, BusTools_TimeGetFmtString, BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_msg_read.c	This example program shows how to program the Bus Controller to run a simple three message minor-frame. The example initializes the channel and sets the channel for external bus. Following those initialization steps, it initializes the Bus Controller and builds a simple 3-message frame. The messages in the frame are linked together and the frame will run continually until stopped by the user. A user callback function is set up by using the function BusTools_RegisterFunction. The user function is invoked each time the registered interrupt event is found in the channels interrupt queue. In this example the callback function is registered for callback on EVENT_BC_MESSAGE. Those events occur on BC message that have interrupt enabled (BC_CONTROL_INTERRUPT. The user callback function then processes the data in each message.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_TimeTagMode BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_StartStop BusTools_RegisterFunction BusTools_BC_MessageRead BusTools_BoardIsV6 BusTools_TimeGetFmtString BusTools_API_Close
example_bc_msg_run.c	This example program shows how to program the Bus Controller to run a simple three message minor-frame. The example initializes the channel the Bus Controller is running on. Then sets the channel for external bus. Following those initialization steps, it initializes the Bus Controller and builds a simple 3-message frame. The message in the frame are linked together and the frame will run continually until stopped by the user.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_StartStop BusTools_API_Close
example_bc_msg_sched.c	This application shows how to setup the bus Controller for Message Scheduling. That is a technique that allows the user to schedule message at differing rates in the major frame. In this example, messages going out on every frame, every other frame, and every fifth frame. The frame rate is set at 1 Hz, so the message traffic can be visually seen on an analyzer. When using Message Scheduling the user programs the start-frame and the repeat rate for each message. A message with a start frame of one and repeat rate of one goes in every frame. A start frame of 1 and repeat rate of 2 causes the message to transact in every other frame. By setting the base frame rate and the start frame and repeat rate for each message the user can control the rate at which the messages in the major frame transact. You can vary frame rate and the start-frame and repeat rates of the messages in this example and see how the changes affect message traffic.	BusTools_API_OpenChannel BusTools_GetFWRevision BusTools_SetInternalBus BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_MessageRead BusTools_TimeGetString BusTools_RegisterFunction BusTools_BC_StartStop BusTools_RT_StartStop BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_multi_buffer.c	This example shows how to initialize the Bus Controller with multiple data buffers. This example requires Firmware version 6.0 or greater and BusTools/1553-API version 8.0 or greater. Previous F/W and API versions only support one or two data buffers. When using the Multiple BC buffer option, you can create a varying number of data buffers for each BC message. This function initializes the board and steps through the process of enabling, creating, filling, and processing the multiple buffers. Three messages are created with 10, 15, and 5 data buffers. An interrupt callback function is registered using BusTools_RegisterFunction. In that callback the data from the buffer is optionally printed and the receive command (RT4) data is incremented to show how to update the data.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_TimeTagMode BusTools_BC_Init BusTools_BC_MessageBlockAlloc BusTools_BC_MessageWrite BusTools_BC_DataBufferWrite BusTools_BC_MessageBufferRead BusTools_BC_DataBufferUpdate BusTools_BC_ReadDataBuffer BusTools_TimeGetString BusTools_RegisterFunction BusTools_BC_StartStop BusTools_API_Close
example_bc_noop.c	This example shows how to NOOP and un-NOOP a message. When a message is NOOPed, It does not transact. The firmware skips over the message as if it were not in the bus list. A single minor frame is created with three messages in the frame. The Second message in the bus list (message 1) is created as a Noop message (BC_CONTROL_MSG_NOP). That means it created as a BC message buffer but set in the NOOP state. When the frame runs only message 0 and 2 transact. The user can toggle message 1 ON or OFF by entering 'U' or 'N" to Un-NOOP or NOOP the message. The function BusTools_BC_MessageNoop is called change the NOOP setting for message 1. You can use BusTools_BC_MessageNoop on any BC message created with BC_CONTORL_MESSAGE or BC_CONTROL_MSG_NOP.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_TimeTagMode BusTools_BC_Init BusTools_BC_MessageBlockAlloc BusTools_BC_MessageWrite BusTools_BC_MessageNoop BusTools_BC_StartStop BusTools_RegisterFunction BusTools_BC_MessageRead BusTools_BoardIsV6 BusTools_TimeGetFmtString BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_options.c	This is anThis is an example program that uses several Bus Controller options to configure a bus list. The options used are Frame-start timing, Message Scheduling, retries and Interrupts. There are two callback functions used. One is for the specific BC message in the bus list excluding the sync mode code. The other callback processes the sync mode code. Retries on no-response or busy are enable on several of the messages. Frame start timing use the gap time as the message delay time from the start of the frame. Message scheduling is used to setup the message in the different frames. Some messages transact at 20 Hz, some at 10 Hz some at 2 Hz and the sync mode code runs at 1 Hz. example program that uses several Bus Controller options to configure a bus list. The options used are Frame-start timing, Message Scheduling, retries and Interrupts. There are two callback functions used. One is for the specific BC message in the bus list excluding the sync mode code. The other callback processes the sync mode code. Retries on no-response or busy are enable on several of the messages. Frame start timing use the gap time as the message delay time from the start of the frame. Message scheduling is used to setup the message in the different frames. Some messages transact at 20 Hz, some at 10 Hz some at 2 Hz and the sync mode code runs at 1 Hz.	BusTools_API_OpenChannel BusTools_GetFWRevision BusTools_SetInternalBus BusTools_TimeTagMode BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_RetryInit BusTools_RegisterFunction BusTools_BC_StartStop BusTools_BC_MessageRead BusTools_BoardIsV6 BusTools_TimeGetFmtString BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_retry.c	This example demonstrates enabling retries on Bus Controller messages. Enabling retries lets the hardware automatically resend a message if the retry condition occurs. In the example the Bus Controller automatically resends the message if a no-response is detected on a retry enabled message or if the RT responds with Busy or Message Error. In order to enable retries you need to configure the Bus Controller to retry by setting the retry condition(s) in BusTools_BC_Init. Once a condition is programmed, each Bus Controller message can then be set to retry. In this example retries are set for message 1 and 2 (0-based) in the bus list. Furthermore, up to eight retries can be programmed by calling BusTools_BC_RetryInit. That function allows programming up to eight retries on either the same or alternate bus. The same and alternate bus designations are from the initial bus setting of the message. For example, if the message transacts Bus A the alternate bus is Bus B. In addition to setting up retries the example also sets up a callback function on a retry.	BusTools_API_OpenChannel BusTools_GetFWRevision BusTools_SetInternalBus BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_RetryInit BusTools_BC_StartStop BusTools_RegisterFunction BusTools_BC_MessageRead BusTools_BoardIsV6 BusTools_TimeGetFmtString BusTools_API_Close
example_bc_rt_bm.c	This console application shows how to configure the Remote Terminal, Bus Controller and Bus Monitor and set up interrupt on BC, RT, BM messages. The RT and BC callback function change the data for Transmit and Receive commands. The Bus Monitor callback displays the data.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BM_Init BusTools_BM_MessageAlloc BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BM_StartStop BusTools_RT_StartStop BusTools_BC_StartStop BusTools_RT_MessageRead BusTools_BM_MessageRead BusTools_BC_MessageReadData BusTools_BC_MessageUpdate BusTools_BC_ControlWordUpdate BusTools_BC_MessageUpdateBuffer BusTools_RegisterFunction BusTools_TimeGetString BusTools_DumpMemory BusTools_ReadBoardTemp BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_rt_broadcast.c	This example shows how to setup broadcast for BC and RT. It also shows the Remote Terminal processes Broadcast messages.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_SetBroadcast BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_CbufbroadWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RT_StartStop BusTools_BC_StartStop BusTools_RegisterFunction BusTools_RT_MessageRead BusTools_API_Close
example_bc_start_frame.c	This example application shows how to configure the Bus Controller for an initial frame that runs once then run a periodic frame. The application uses BusTools_BC_Start to start the initial frame at message 40. After that the frame starting at message zero (0) runs. This example also shows how to set an interrupt on minor-frame-overflow. A minor-frame-overflow occurs when the messages in a minor-frame take longer than the programmed frame-time to transact. When this occurs, messages exceeding the frame-time are suppressed and the new frame starts. In this example the frame rate is set for 1 Hz (1000000). All the messages transact. If you change the frame rate to 1000 Hz (1000), a minor-frame overflow occurs and the last two messages in the frame are suppressed. The minor-frame-overflow interrupt callback increments a counter each time it runs. the number of overflow events are printed out at the end of this example.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_Start BusTools_RegisterFunction BusTools_BC_MessageBufferRead BusTools_TimeGetString BusTools_BC_DataBufferUpdate BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bc_trigger_oneshot.c	This example program demonstrates how to start the Bus Controller using a trigger input. The example sets up a simple BC message list with 1-R-1-32 and 2-T-2-32, in a 500ms minor frame. This BC list runs until stopped by user input. Data is automatically displayed for 1-R-1-32 and 2-T-2-32. The user hits Enter to quit, shutdown the application and exit. BusTools_BC_Trigger is used to setup a 1shot trigger that starts the Bus Controller running. After calling BusTools_BC_StartStop the BC waits for an external trigger input before running. The trigger can be from an external trigger source or generated by the board. If the trigger source in internal to the board you must wrap discrete 7 and 8 together. **Note**: not all boards have discrete channels or are configure with the discrete 7 and 8. You will need to check the configuration of the board installed to make sure that discrete channels are available.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_Trigger BusTools_RegisterFunction BusTools_BoardIsV6 BusTools_SetV6TrigOut BusTools_SetV6TrigIn BusTools_DiscreteWrite BusTools_DiscreteSetIO BusTools_DiscreteTriggerOut BusTools_DiscreteTriggerIn BusTools_BC_StartStop BusTools_BC_MessageRead BusTools_TimeGetFmtString BusTools_API_Close
example_bc_trigger_user.c	This example application shows BC Triggering using the BC_TRIGGER_USER option. In this option the user controls how the BC frames run by adding BC_CONTROL_LAST to stop the Bus Controller. Each time you stop the Bus Controller using BC_CONTROL_LAST a trigger input is needed to restart. You can have one or more frames transacting off each trigger. This example creates six frames. The first four frames each start on a trigger input. The last two frames combined. The fifth frame starts on a trigger. The sixth frame runs after the normal frame delay, then stop when the BC_CONTROL_LAST buffer transacts. The list loops back to the first frame requiring a trigger to start. The application provides an option for internally generated triggers or an external user-supplied trigger. If you select the internal trigger, you will need to connect discrete7 and 8 together. Not all boards have these discrete channels available so check the board's configuration.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_TimeTagMode BusTools_SetV6TrigOut BusTools_SetV6TrigOut BusTools_DiscreteWrite BusTools_DiscreteSetIO BusTools_DiscreteTriggerOut BusTools_DiscreteTriggerIn BusTools_BC_Init BusTools_BC_Trigger BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_StartStop BusTools_ExtTriggerOut BusTools_DumpMemory BusTools_BC_StartStop BusTools_RegisterFunction BusTools_BC_MessageRead BusTools_BoardIsV6 BusTools_TimeGetFmtString BusTools_API_Close
example_bit.c	This example program that initializes a channel and runs the Internal Built-In-Test and the Cable Wrap Test, then exits.	BusTools_API_OpenChannel BusTools_BIT_InternalBit BusTools_BIT_CableWrap BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bm_filter_messages.c	This example program sets up a Bus Monitor to record the message traffic to a file. The user is prompted for the number of messages to capture. The program then captures those messages and writes them to a file. This is a version of example_bm_recorder that demonstrates message filtering. This program only captures messages for RT1 SA2 TX. This is a monitor only example. It needs something to generating bus traffic that includes a RT1 SA2 TX message to capture data.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BM_Init BusTools_GetBoardType BusTools_BM_FilterWrite BusTools_BM_MessageAlloc BusTools_BM_StartStop BusTools_BM_MessageReadBlock BusTools_RegisterFunction BusTools_API_Close
example_bm_process_messages.c	This example program sets up a Bus Monitor to process each message transacted on the 1553 bus. The example shows how to use BusTools_RegisterFunction to setup a user-callback function to processes individual Bus Monitor Messages. It also shows how to write a user-callback that processes each Bus Monitor message. In the callback function the Bus Monitor message data is displayed on the console. This is a monitor only example; you will need to connect to a 1553 bus with traffic to capture messages.	BusTools_API_OpenChannel BusTools_BM_Init BusTools_SetInternalBus BusTools_GetBoardType BusTools_BM_MessageAlloc BusTools_BM_StartStop BusTools_RegisterFunction BusTools_BM_MessageRead BusTools_API_Close
example_bm_process_msg_blk.c	This example sets up a Bus Monitor to process each Bus Monitor message transacted. This example also shows how to process the messages using the function BusTools_BM_ReadLastMessageBlock. This function parses the interrupt queue to determine the BM messages that have transacted since the last call. For a BM message to record the BM initialization must define an interrupt condition like BT1553_INT_END_OF_MESS. This processing differs from using BusTools_RegisterFunction in that your application must provide a timing loop to call BusTools_BM_ReadLastMessageBlock periodically. The timing must be set so the BM messages (and other messages that may be recorded in the interrupt queue do not overwrite unprocessed entries. This example uses a 50-millisecond delay. This application uses the Windows function kbhit to break out of timing loop. If using this example on non-Windows systems, you will need to provide a kbhit function.	BusTools_API_OpenChannel BusTools_BM_Init BusTools_SetInternalBus BusTools_GetBoardType BusTools_BM_MessageAlloc BusTools_BM_StartStop BusTools_BM_ReadLastMessageBlock BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bm_recorder.c	This example program sets up a Bus Monitor to record the message traffic to a file. The user is prompted for the number of messages to capture. The program then captures those messages and writes them to a file. Since this is a monitor only example, you will need something else generating bus traffic so there will be messages to capture.	BusTools_API_OpenChannel BusTools_BM_Init BusTools_SetInternalBus BusTools_GetBoardType BusTools_BM_MessageAlloc BusTools_BM_MessageReadBlock BusTools_BM_StartStop BusTools_RegisterFunction BusTools_API_Close
example_bm_share.c	This application demonstrates how to initialize a channel and the share the channel so other applications can use that channel. This application initializes and shares the channel. It then configures and runs a Bus Monitor. This allows separate Bus Controller and Remote Terminal applications to join this channel.	BusTools_API_OpenChannel BusTools_API_ShareChannel BusTools_SetInternalBus BusTools_BM_Init BusTools_BM_MessageAlloc BusTools_RegisterFunction BusTools_BM_StartStop BusTools_BM_MessageRead BusTools_API_QuitChannel BusTools_API_Close
example_bm_trig_start_stop.c	This example shows how to enable Bus Monitor start trigger and stop trigger using the function BusTools_BM_TriggerWrite. A start trigger specifies a condition or set of conditions that must occur before the Bus Monitor starts processing data. The stop trigger defines a condition of set of conditions that must occur to stop the Bus Monitor processing. When a start trigger is set it prevent Bus Monitor data from being recorded in the interrupt queue. A stop trigger disables the Bus Monitor message from going into the interrupt queue. You can still record all the message traffic. In this example the Bus Monitor is configured to start only after a command is sent to RT1. All traffic transacting prior is not processed. Once started the Monitor stops processing messages when a command word with RT22 occurs when armed first by a command word to RT2.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BM_Init BusTools_BM_MessageAlloc BusTools_BM_TriggerWrite BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_CbufWrite BusTools_RT_AbufWrite BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RegisterFunction BusTools_BM_MessageRead BusTools_TimeGetFmtString BusTools_BC_StartStop BusTools_RT_StartStop BusTools_BM_StartStop BusTools_RegisterFunction BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_bm_trgout.c	This example shows how to setup the Bus Monitor to generate an external trigger out when it records a specific message. In this example the Bus Monitor and optionally the BC and RT are configured. The Bus Monitor will generate an external trigger (trigger-out on discrete 7) when it records a BC message to RT 8. This example also routes the output trigger back to the input trigger (set up on discrete 8) and sets up an interrupt on external trigger. This setup generates an output trigger and captures it as an input trigger for display. In order for this set up to work you need to physically connect the discrete7 to discrete 8. Discrete and triggers vary between boards and this setup may not run on your board variant. Please refer to the "MIL-STD-1553 Hardware Installation Guide" for details about triggers and discrete channels available on the different 1553 products.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_ExtTrigIntEnable BusTools_BM_Init BusTools_BM_MessageAlloc BusTools_BM_TriggerWrite BusTools_SetV6TrigOut BusTools_SetV6TrigIn BusTools_DiscreteWrite BusTools_DiscreteSetIO BusTools_DiscreteTriggerOut BusTools_DiscreteTriggerIn BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_BC_StartStop BusTools_RT_StartStop BusTools_BM_StartStop BusTools_DumpMemory BusTools_RegisterFunction BusTools_API_Close
example_dbca.c	This application shows how to setup Dynamic Bus Controller allocation for both the Remote Terminal and the Bus Controller. This example uses two channels, one as the initial Bus Controller and the other as the initial Remote Terminal. The RT channel also configures a Bus Controller, but it is not started. In this example the active BC is initialized to send a bus list using RTs 2, 4 and 6. There is also an aperiodic message to send a mode code 0 to RT1. If RT1 accepts the DBCA by setting the DBA bit in the Status word, the Bus Controller halts. The inactive BC, configured by the RT channel, has a bus list to RTs 12, 14, and 16. RT1 is setup to accept the DBCA Mode Code 0. It automatically starts the Bus Controller. User input from the console sends the aperiodic Mode Code 0 to RT1. Once that transacts, the initial BC is halted, and the RT side BC takes over.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RT_StartStop BusTools_BC_StartStop BusTools_BC_AperiodicRun BusTools_RT_MessageRead BusTools_RegisterFunction BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_ext_trig.c	This application shows how to setup interrupts on external trigger. In addition, it shows how to setup discrete for input and output triggers. This test requires a wrap connector between discrete 7 and discrete 8 on the D50 connector. The test generates a output trigger and wraps output on discrete 7 to the input on discrete 8. The input trigger generates an interrupt.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_SetVoltage BusTools_BM_Init BusTools_BM_MessageAlloc BusTools_BM_StartStop BusTools_SetV6TrigOut BusTools_SetV6TrigIn BusTools_DiscreteWrite BusTools_DiscreteSetIO BusTools_DiscreteTriggerOut BusTools_DiscreteTriggerIn BusTools_ExtTrigIntEnable BusTools_RegisterFunction BusTools_BM_StartStop BusTools_ExtTriggerOut BusTools_TimeTagRead BusTools_API_Close
example_irig1.c	This test program demonstrates the setup of the IRIG-B output and input. This program just configures IRIG for EXTERNAL or INTERNAL source, sets the IRIG the current time/date and then loops 20 times reading and displaying the time every second. This REQUIRES a board with the IRIG option. The part number should include a 'W', which indicates IRIG. For example: QPCX-1553-4MW. Use BusTools_BoardHasIRIG to find out if you have an IRIG enabled board.	BusTools_API_OpenChannel BusTools_BoardHasIRIG BusTools_BM_Init BusTools_IRIG_Config BusTools_IRIG_Calibration BusTools_IRIG_Valid BusTools_IRIG_SetTime BusTools_TimeTagMode BusTools_TimeTagRead BusTools_API_Close
example_rev.c	This example program initializes a board and displays version information and general board information.	BusTools_GetDevInfo BusTools_StatusGetString BusTools_API_OpenChannel BusTools_GetBoardType BusTools_ReadBoardTemp BusTools_GetRevision BusTools_GetFWRevision BusTools_BoardIsV6 BusTools_BoardIsMultiFunction BusTools_GetCSCRegs BusTools_GetChannelCount BusTools_BoardHasIRIG BusTools_GetSerialNumber BusTools_MemoryAvailable BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_rt_auto_wrap.c	This application shows how to automatically wrap RT receive and transmit messages buffers. When transmit and receive message buffers are wrapped the RT transmits the data from the previous receive command.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_SetVoltage BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_RegisterFunction BusTools_BC_MessageRead BusTools_RT_StartStop BusTools_BC_StartStop BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_rt_buffer_switch.c	The example demonstrates how to setup the Remote Terminal to manually switch between two RT message buffers. The RT is initialized with two message buffer per RT/SA/TX/RX combination. Then, using BusTools_RT_MessageGetaddr and BusTools_MemoryWrite2 the RT message buffer linked list is altered to have each buffer point to its address instead of the next buffer in the list. Instead of looping through the message buffer the RT uses only a single buffer. The user must write the address of the message they want to run into the RT control Buffer. This example sets up a RT3 SA3 TX with 2 buffers. Then manually changes the message buffer linking so only a single buffer is used. By entering 0 or 1 at the command prompt the user can switch between buffer 0 (data = 0x1111) and buffer 1 (data = 0x2222) This example shows this process for both the V6 and V4/5 firmware designs. The high-level API hides the underlying design differences in the firmware. When directly accessing memory, the user needs to understand the memory layout for the firmware they are using. Refer to the *MIL-STD-1553 Universal Core Architecture Reference Manual* for the V4/5 firmware and the *MIL-STD-1553 Enhanced Universal Core Architecture (UCA32) Local Processing Unit (LPU) Reference Manual* for V6 firmware. Also keep in mind that V6 firmware uses 32-bit addressing while V4/5 uses 16-bit addressing. When running this example, it you a dump the board's memory by typing 'd' or 'D'. Reviewing the memory dump will help follow the memory manipulations in this example.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_RT_MessageGetaddr BusTools_BoardIsV6 BusTools_RamAddr BusTools_RelAddr BusTools_MemoryWrite2 BusTools_GetAddr BusTools_MemoryRead2 BusTools_RT_MessageRead BusTools_RegisterFunction BusTools_RT_StartStop BusTools_BC_StartStop BusTools_DumpMemory BusTools_API_Close
example_rt_ei_late_rsp.c	This example program sets up RT1 with two subaddresses, SA1 RECEIVE and SA2 TRANSMIT. It also demonstrates how to inject errors into an RT message. In this case, we inject a LATE RESPONSE ERROR. The user can specify the response time from 7-31us. This can be done for SA1 RECEIVE or SA2 TRANSMIT.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_RT_Init BusTools_RT_AbufWrite BusTools_RT_CbufWrite BusTools_EI_EbufWriteENH BusTools_RT_MessageWrite BusTools_RT_StartStop BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_rt_ei_parity.c	This example program sets up a simple RT1 with two subaddresses, SA1 RECEIVE and SA2 TRANSMIT. It also demonstrates RT error injection. In this case, it creates a PARITY ERROR. This can be done for SA1 RECEIVE (on STATUS word) or on SA2 TRANSMIT (on STATUS or DATA words).	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_RT_Init BusTools_RT_AbufWrite BusTools_RT_CbufWrite BusTools_EI_EbufWriteENH BusTools_RT_MessageWrite BusTools_RT_StartStop BusTools_API_Close
example_rt_extended_status.c	This example shows how to configure a Remote Terminal and enable Extended Status updates. Under normal 1553 operation the status returned by the RT is for all Sub-addresses and transmit and receive buffer. BusTools_RT_AbufWrite sets the status response for the RT and enable extended status for a RT. Using Extended Status mode, the RT can set the status word for each Sub-address, Transmit, Receive and buffer for the RT. In this example the RTs are set up with two buffers per RT/SA/TX/RX combination. For RT 3 Transmit and RT 4 receive the second buffer is set to respond with updated status. RT 3 responds with Busy (BSY) and RT 4 responds with Message Error (ME). This causes the status word for those two messages to toggle between the RT message status and the extended status programmed by BusTools_RT_MessageWriteStatusWord . To use Extended Status set the Extended Status enable in the 'inhibit terminal flag' parameter in call to BusTools_RT_AbufWrite.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_RT_MessageWriteStatusWord BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RegisterFunction BusTools_RT_StartStop BusTools_BC_StartStop BusTools_BC_MessageRead BusTools_TimeGetFmtString BusTools_API_Close
example_rt_join.c	This example shows how an application joins an already initialized channel. This function demonstrates a simple RT application. RTs 1 - 8 are programmed and a callback function is setup to process RT message in the interrupt queue. This function requires that the channel joined already be initialize and shared by another application. The user can run this example with example_bm_share and example_bc_join. Joining a shared channel allows individual Remote Terminal, Bus Monitor and Bus Controller applications to run off a single channel.	BusTools_API_JoinChannel BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_RegisterFunction BusTools_TimeTagRead BusTools_RT_MessageRead BusTools_TimeGetString BusTools_RT_StartStop BusTools_API_QuitChannel

Example Name	Descriptions	BusTools/1553-API
example_rt_mc17.c	Demonstrates how to process Mode Code 17, Sync with Data.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_TimeTagMode BusTools_GetRevision BusTools_GetFWRevision BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_RegisterFunction BusTools_BC_StartStop BusTools_RT_StartStop BusTools_DumpMemory BusTools_RT_MessageRead BusTools_BC_MessageRead BusTools_API_Close
example_rt_mode_code.c	This example shows multi-buffering of mode code data. Uses Mode Code 17, synchronize with data and Mode code 19, Transmit Bit Word. This example shows how to setup data for Mode codes if that are in the same bus list.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_TimeTagMode BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RegisterFunction BusTools_RT_MessageRead BusTools_BC_MessageRead BusTools_RT_StartStop BusTools_BC_StartStop BusTools_DumpMemory BusTools_API_Close
example_rt_monitor.c	This example shows how to setup the Remote Terminal in monitor only mode. In this mode the RT capture all messages to the specified RT but will not respond to the RT messages. With the exception of calling BusTools_RT_MonitorInit the Remote terminal is configured identically to a real Remote Terminal.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_TimeTagMode BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_RT_MonitorEnable BusTools_RegisterFunction BusTools_RT_StartStop BusTools_RT_MessageRead BusTools_RT_StartStop BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_rt_set_status.c	This example program sets up RT1 with two subaddresses, SA1 RECEIVE and SA2 TRANSMIT. This example program also demonstrates: - DISABLING and ENABLING an RT to respond on the 1553 Bus - toggling of the BUSY, SERVICE REQUEST, and TERMINAL FLAG bits in the RT Status Word - ILLEGALIZING commands to an RTSA An external BC device is required and configured to send a BC->RT command to RT1 SA1 and an RT->BC command to RT1 SA2, at a minimum. Recommended BC setup: - BC->RT Command RT1, SA1, RECEIVE, 32 Data Words, Bus A - RT->BC Command RT1, SA1, TRANSMIT, 32 Data Words, Bus A - BC->RT Command RT1, SA1, RECEIVE, 32 Data Words, Bus B - RT->BC Command RT1, SA1, TRANSMIT, 32 Data Words, Bus B After DISABLING the RT, verify that the RT does not respond with Status to the BC Command on Bus A and Bus B. After ENABLING the RT, verify that the RT responds with Status to the BC Command on Bus A and Bus B. Verify the RT Status Word Response using the BC device as the BUSY, SERVICE REQUEST, and TERMINAL FLAG bits are set and cleared in the RT Status Word. The BC List should execute at a rate to allow for monitoring of the changes to the RT Status Word. When ILLEGALIZING commands to RT1-SA1-RX and RT1-SA2-TX, the RT will respond with the MESSAGE ERROR bit set in the status word. And, for the TRANSMIT command, the RT will not send any data words. **Note** that alternative coding options are included for setting and clearing the RT Status Word bits.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_RT_Init BusTools_RT_AbufWrite BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_StartStop BusTools_API_Close

Example Name	Descriptions	BusTools/1553-API
example_rt_wrap.c	This console example program shows how to manually wrap RT receive and transmit messages buffers. This differ from the automatic way is that the example gets the address of the RT buffer and manipulates the buffer address data.	BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_BC_Init BusTools_BC_MessageAlloc BusTools_BC_MessageWrite BusTools_RT_Init BusTools_RT_CbufWrite BusTools_RT_MessageWrite BusTools_RT_AbufWrite BusTools_RT_MessageWrite BusTools_BoardIsV6 BusTools_RT_AbufWrite BusTools_GetAddr BusTools_MemoryRead2 BusTools_RelAddr BusTools_MemoryWrite2 BusTools_MemoryRead BusTools_MemoryWrite BusTools_RT_StartStop BusTools_BC_StartStop BusTools_RegisterFunction BusTools_BC_MessageRead BusTools_API_Close
example_timetag_read.c	This example set the time tag to either zero or the present time and data. Then it reads the time tag register every 500 milliseconds (.5s) and prints the raw ns or µs depending the firmware version. It also converts the time to a string and prints the time string.	BusTools_ListDevices BusTools_API_OpenChannel BusTools_SetInternalBus BusTools_GetRevision BusTools_BoardIsV6 BusTools_TimeTagMode BusTools_RegisterFunction BusTools_TimeTagRead BusTools_API_Close

Glossary

LINK

This glossary only features terms special to this manual. Explanations of more general terms can be found in the Glossary, publication number GLOS1.

1553	A component or message in accordance with 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 an 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 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 do 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 (receive). The second command word instructs a specific RT to ignore the receive command and to transmit data.
BSP	Board Support Package. Software used by VxWorks or Integrity 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. The DLL may be updated or changed without requiring that the application be re-compiled or re-built.
FPGA	Field Programmable Gate Array.
IP Carrier	An interface board designed to adapt one or more IP Modules to a different mechanical and electrical bus structure.
IP Module	A modular mezzanine card based on the VITA 4-1995 IP Module Draft Standard 1.0.d.0.
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 Sub Address 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 are valid, as specified by the standard.
Nanosecond	1/1,000,000,000 (billionth) of a second. Abbreviated as ns.
NT	A Microsoft operating system, Windows NT
Operating System (OS)	The software that operates the computer, such as Windows or Linux.
PC	Personal Computer. A specific type of computer, based on the Intel 80x86 processor line.
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.
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 an RT-BC message or just a Transmit message.
RT-BC	A 1553 message that transfers data from an RT to the BC. Also called an 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.
Sub Address	The address within an RT that acts as the source or destination of a specific message. Sub Addresses 1 through 30 are used for messages, SA 0 and 31 are reserved for mode codes.
WCS	Writeable Control Store. The WCS contains the microcode that runs the 1553 hardware. The WCS is downloaded to the board from the host bus during initialization or is part of the onboard FPGA configuration data that is loaded when the board is powered.
window	A functional component of an application. A display.
Windows	One of several operating systems supplied by Microsoft Corporation.

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

abaco SYSTEMS

Publication No. 1500-038 Rev. 5.11