ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Abaco Systems / Condor Engineering MS1553PC* at our website: *Click HERE*

PC-1553/PC-1773
IP-1553
USER'S MANUAL

SSP

Al Horn
505-856, 5613

Condor Dual ages in Albuquerque
from Ray Dickey 4/4/95

CONDOR ENGINEERING
INCORPORATED

Copyrights

Software Copyright © 1993-1998 Condor Engineering, Inc.

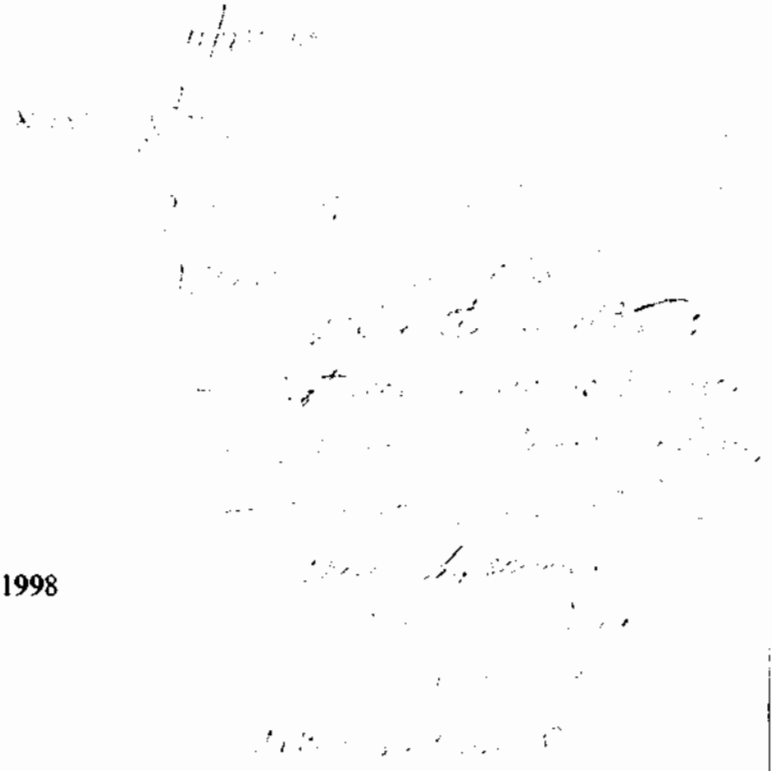
User's Manual Copyright © 1993-1998 Condor Engineering, Inc.

This software product is copyrighted and all rights are reserved. The distribution and sale of this product are intended for the use of the original purchaser only per the terms of the License Agreement.

This User's Manual is copyrighted and all rights are reserved.

This document may not, in whole or part, be; copied; photocopied; reproduced; translated; reduced or transferred to any electronic medium or machine-readable form without prior consent in writing from Condor Engineering, Inc.

BusTools is a registered trademark of Condor Engineering, Inc.
Microsoft is a registered trademark of Microsoft Corporation
Windows is a registered trademark of Microsoft Corporation



PC-1553/PC-1773/IP-1553 User's Manual

Software API Revision:	2.60
PC-1553/PC-1773 Hardware Revision:	2.05
IP-1553 Hardware Revision	2.20
Document Date:	23 July 1998
Document Revision:	1.30

Condor Engineering, Inc.
101 W. Anapamu, Suite G
Santa Barbara, CA 93101
(805) 965-8000
(805) 963-9630 (fax)
support@condoreng.com (e-mail)

Contents

Chapter 1 Overview

About this Manual 1

Chapter 2 PC-1553/PC-1773 Hardware Installation

Installation Procedure 3
PC-1553/PC-1773 Memory Map 5
I/O Map 6
Interrupt Line Select (IRQ3-7,9-12,14-15) 7
1553 - Bus Coupling 7
Factory Settings 9

Chapter 3 IP-1553 Hardware Installation

Installation Procedure 11
IP Connector Description 12
Signal Descriptions 13
  Transformer Versus Direct Coupling 13
  MIL-STD-1553B Signals 14
IP-1553 Jumper Location 14
External Input 15
  BC Trigger 15
  RT Trigger 15
  BM Trigger 15
  Reset Tag Timer 15
External Output 16
  BC Trigger 16
  RT synchronize mode code 16
External Hardwired RT Address 16
IP Carrier Jumper Settings 16
IP-1553 Memory Map 17

Chapter 4 Initialization and Global Routines

Initialization and Shutdown 19
Global Parameters 20
General Purpose Routines 22
Example Programs 22

Chapter 5 BUS Monitor

BM Hardware Operation....23
BM Software Operation....25
BM Recording (16/32-Bit Windows Only)....27

Chapter 6 Remote Terminals

RT Hardware Operation....29
RT Software Operation....31

Chapter 7 Bus Controller

BC Hardware Operation....35
BC Software Operation....37
BC Oneshot Operation....38
Example....39

Chapter 8 BusTools Memory

Hardware Operation....43
Software Operation....44
PC-1553/1773 SEG1 Memory Organization....45
PC-1553/1773 SEG2 Memory Organization....46
IP-1553 Memory Organization....46

Chapter 9 Error Injection

Error Injection Hardware Operation....49
Error Types....50
Error Injection Software Operation....52

Chapter 10 BusTools API Routines

Programming Notes....54
BusTools_API_Close....55
BusTools_API_Init....56
BusTools_API_InitExtended....58
BusTools_BC_AperiodicRun....62
BusTools_BC_AperiodicTest....65
BusTools_BC_Init....66
BusTools_BC_IsRunning....69
BusTools_BC_MessageAlloc....70
BusTools_BC_MessageGetaddr....72
BusTools_BC_MessageGetid....73
BusTools_BC_MessageNoop....74
BusTools_BC_MessageRead....75
BusTools_BC_MessageUpdate....76
BusTools_BC_MessageWrite....77

BusTools_BC_OneShotExecute 78
BusTools_BC_OneShotInit 80
BusTools_BC_Start 82
BusTools_BC_StartStop 83
BusTools_BC_Trigger 84
BusTools_BM_FilterRead 85
BusTools_BM_FilterWrite 86
BusTools_BM_Init 87
BusTools_BM_MessageAlloc 88
BusTools_BM_MessageConvert 89
BusTools_BM_MessageGetid 90
BusTools_BM_MessageRead 91
BusTools_BM_MessageReadBlock 92
BusTools_BM_StartStop 94
BusTools_BM_TriggerRead 95
BusTools_BM_TriggerWrite 96
BusTools_EI_EbufWrite 97
BusTools_EI_Getaddr 98
BusTools_EI_Getid 99
BusTools_ErrorCountClear 100
BusTools_ErrorCountGet 101
BusTools_GetAddr 103
BusTools_GetData 105
BusTools_GetRevision 106
BusTools_InterMessageGap 107
BusTools_InterruptDone 108
BusTools_InterruptRegister 109
BusTools_MemoryAlloc 111
BusTools_MemoryRead 112
BusTools_MemoryReadIDPROM 113
BusTools_MemoryWrite 114
BusTools_RecordDone 115
BusTools_RecordGetFlag 116
BusTools_RecordRegister 117
BusTools_RegisterFunction 118
BusTools_RT_AbufRead 120
BusTools_RT_AbufWrite 121
BusTools_RT_CbufbroadRead 122
BusTools_RT_CbufbroadWrite 123
BusTools_RT_CbufRead 124
BusTools_RT_CbufWrite 125
BusTools_RT_Init 127
BusTools_RT_MessageGetaddr 128
BusTools_RT_MessageGetid 129

BusTools_RT_MessageRead 130
BusTools_RT_MessageWrite 131
BusTools_RT_MessageWriteDef 132
BusTools_RT_MessageWriteStatusWord 133
BusTools_RT_StartStop 135
BusTools_SetBroadcast 136
BusTools_SetExternalSync 137
BusTools_SetInternalBus 138
BusTools_SetOptions 139
BusTools_SetSa31 141
BusTools_SetVoltage 142
BusTools_StatusGetString 143
BusTools_TimeGetString 144

Appendix A Error Codes

General Errors A-1
BC Errors A-2
BM Errors A-5
EI Errors A-6
RT Errors A-6
Low Level Errors A-8
BusTools Driver Errors A-10
Numeric Index A-12

Appendix B Data Structures

1553 Command Word (BT1553_COMMAND) B-2
1553 Status Word (BT1553_STATUS) B-3
BC Retry Parameters (BusTools_BC_Init() arguement) B-4
BC Message Buffer (API_BC_MBUF) B-5
BM Filter Buffer (API_BM_CBUF) B-12
BM Message Buffer (API_BM_MBUF) B-15
BM Trigger Buffer (API_BM_TBUF) B-18
BM Word Status Bits (8/16 bit) B-20
Error Injection Definition (API_EIBUF) B-23
Interrupt Enable / Message Status Bits (32 bit) B-26
Interrupt Register/Filter/FIFO Structure B-30
RT Address Control Block (API_RT_ABUF) B-33
RT Control Buffer (API_RT_CBUF) B-35
RT Control Buffer for Broadcast (API_RT_CBUFBROAD) B-36
RT Message Buffer (read-only) (API_RT_MBUF_READ) B-37
RT Message Buffer (write-only) (API_RT_MBUF_WRITE) B-39
Time Structure (BT1553_TIME) B-40

CHAPTER 1

Overview

The PC-1553 and PC-1773 are multifunction boards that provide an interface between the ISA bus and the MIL-STD-1553 and MIL-STD-1773 buses respectively. The IP-1553 is a multifunction module that provides an interface between an IndustryPack bus and a MIL-STD-1553 bus. These products support simultaneous operation of BC, up to 31 RTs and BM functionality. The PC-1553-S and IP-1553-S are single function bus interfaces that simulate BC, up to 31 RTs or BM functionality.

Since both products are architecturally very similar, they share the same Application Programmer's Interface (API) which is written in the "C" language. This API is available for 16-bit MS-DOS, 16-bit Windows® (e.g., 3.x), and 32-bit Windows® (e.g., Window 95 and Windows NT versions 3.5x and 4.0). The API is capable of supporting concurrent operation of up to 4 PC-1553/1773 or IP-1553 interfaces, in any combination.

This manual describes the installation and API for the PC-1553 and PC-1773 Boards. It also provides installation instructions and API information for the IP-1553 Board. Routines have been defined to perform Bus Monitor (BM), Remote Terminal (RT), and Bus Controller (BC) functions in the 1553 and 1773 bus environments.

The routines have been designed to ease the task of developing applications for the hardware. An application programmer should not need to know details of how the board operates or how on-board memory is organized (though knowledge of these details may be helpful during the development process).

About this Manual

In addition to this chapter, this manual includes:

- **Chapter 2, PC-1553/PC-1773 Hardware Installation** – provides a description of how to configure and install the PC-1553/PC-1773 Boards.

- **Chapter 3, IP-1553 Hardware Installation** – provides a description of how to configure and install the IP-1553 Board.
- **Chapter 4, Initialization and Global Routines** – provides a description of the API initialization and general-purpose functions available in the API.
- **Chapter 5, Bus Monitor** – provides a description of the Bus Monitor (BM) functionality and how to control it from the API.
- **Chapter 6, Remote Terminals** – describes Remote Terminal (RT) functionality and how to control it from the API.
- **Chapter 7, Bus Controller** – describes the Bus Controller (BC) functionality and how to control it from the API.
- **Chapter 8, Memory Organization** – provides a brief description of the 1553 memory and how the API organizes it.
- **Chapter 9, Error Injection** – describes the Error Injection functions of the boards.
- **Chapter 10, API Routines** – provides a detailed description of each API routine.
- **Appendix A, Error Codes** – provides a list and description of the API error codes.
- **Appendix B, Data Structures** – provides a list and description of data structures and definitions used by the API.

NOTE: Throughout this manual you will see references to *BusTools hardware*. This is the general term for the PC-1553, PC-1773, IP-1553 and other supported hardware platforms.

BusTools software refers to the Graphical User Interface (GUI) program which uses the *BusTools API* described herein, and adds the ability to generate 1553 simulations of the Bus Controller and Remote Terminals, as well as record 1553 data to disk for later analysis, using an intuitive point-and-click graphical user interface.

CHAPTER 2

PC-1553/PC-1773 Hardware Installation

Installation Procedure

This section tells how to install the PC-1553 ISA board. If possible, use a static free workstation. Before beginning the installation process, inspect the packaging material to insure that the board hasn't been damaged during shipment. If it has, contact the shipper right away.

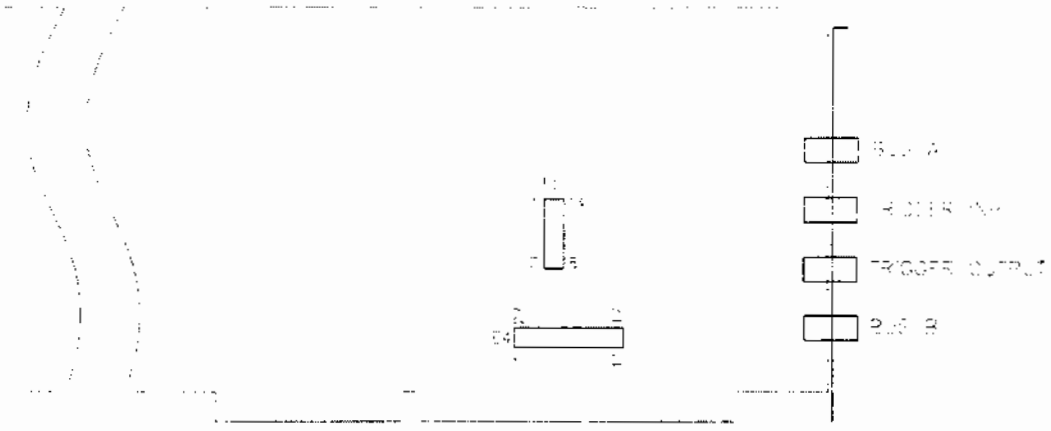
Remove the disks and set them aside. Then follow the steps in this section:

1. Remove the board from the anti-static bubble pack and lay the board on top of the bag. This prevents static discharge from damaging the board when you change the jumpers.
2. Set the jumpers on the board to the desired settings for your operating system. (See the *PC-1553/PC-1773 Memory Map* section on the following pages for specific settings).
3. Remove the cover from your PC. Remove the blank mounting plate from one of the unused slots in your computer. Install the board and bolt the guide plate to the PC with the screw that was removed in the earlier step.
4. On the PC-1553 board, connect your MIL-STD-1553 bus to the PL-74 connectors provided on the back of the interface card. The top connector is for bus A and the bottom is for bus B. (The two BNC connectors in the middle are trigger inputs/outputs. The upper connector is the input, and the lower connector is the output. The trigger functions are discussed in Chapter 5).

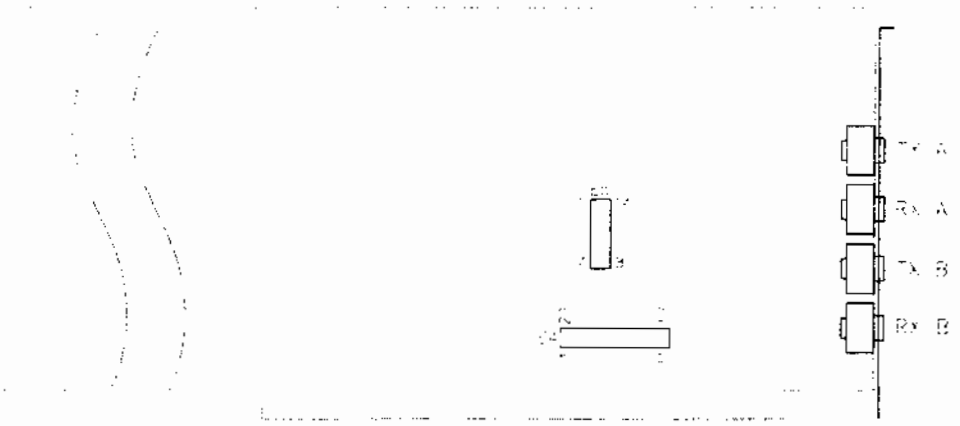
If you are using "transformer" coupling, than connect the two "stubs" from the transformers to the connectors on the back of the interface card.

If you are using "direct" coupling and this is the last connection on the bus a "Tee" should be used and a 78-ohm terminator should be connected to provide the proper termination for the bus. Otherwise you should use a "Tee" with a cable on each end of the "Tee".

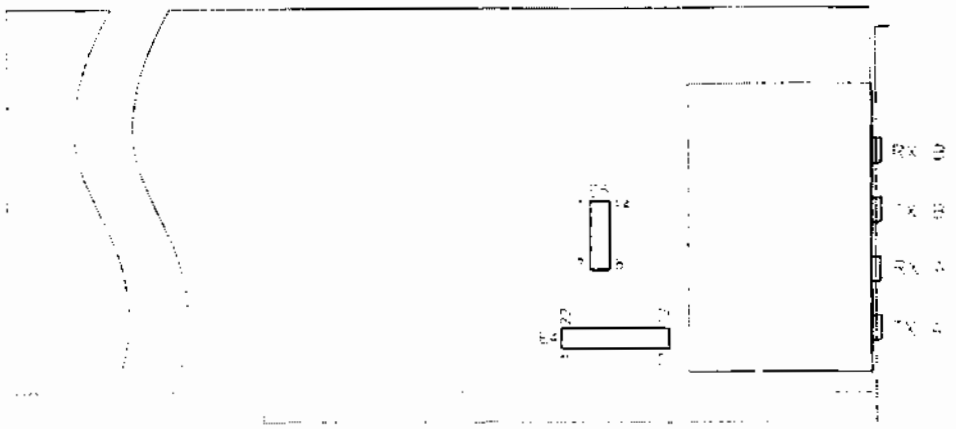
5. On the PC-1773 board, you see, from top to bottom, the channel A transmitter, the channel A receiver, the channel B transmitter, the channel B receiver. Connect four 850 NM fiberoptic cables with low-profile SMA connectors (or ST connectors if you ordered the ST option).



PC-1553 Board Diagram



PC-1773 Board Diagram



PC-1773 (1300 nm only) Board Diagram

PC-1553/PC-1773 Memory Map

The board may be mapped into either a 32K or 64K byte block in upper memory or as a 256K byte block in extended memory, depending on the jumper settings on header E5.

If upper memory is used, then the memory is paged by an internal page register, which is accessed in I/O space. The board can be mapped into any 32K byte area and some 64K byte areas from A0000 through E7FFF (segment addresses A000 - E7FF) starting at the base addresses given in the table below.

If extended memory is used, then the board can be mapped from F00000 through FFFFFF, as shown in the table. Header locations on block E5, 5-10, 4-11, 3-12 and 2-13 are used for selecting the base address. Do not modify the setting of E5-1 and E5-14!

Only 32K byte block upper memory mapping is supported by the the BusTools API (BusAPI.DLL or BusAPI32.DLL), and the *BusTools GUI software*, under 32-bit operating systems.

Only 32K and 64K byte block upper memory mapping is supported under the remaining operating systems.

Memory Block Size/Type	Address Range	E5: 2-13	E5: 3-12	E5: 4-11	E5: 5-10	E5: 1-14
256K/EXTENDED	F00000 - F3FFFF	ON	ON	ON	ON	&
256K/EXTENDED	F40000 - F7FFFF	ON	ON	ON	OFF	&
256K/EXTENDED	F80000 - FBFFFF	ON	ON	OFF	ON	&
256K/EXTENDED	FC0000 - FFFFFF	ON	ON	OFF	OFF	&
32K/UPPER MEMORY *	0A0000 - 0A7FFF	ON	OFF	ON	ON	&
32K/UPPER MEMORY *	0A8000 - 0AFFFF	ON	OFF	ON	OFF	&
32K/UPPER MEMORY *	0B0000 - 0B7FFF	ON	OFF	OFF	ON	&
32K/UPPER MEMORY *	0B8000 - 0BFFFF	ON	OFF	OFF	OFF	&
64K/UPPER MEMORY *	0E0000 - 0EFFFF	OFF	ON	ON	ON	&
32K/UPPER MEMORY	0C0000 - 0C7FFF	OFF	OFF	ON	ON	&
32K/UPPER MEMORY	0C8000 - 0CFFFF	OFF	OFF	ON	OFF	&
32K/UPPER MEMORY	0D0000 - 0D7FFF	OFF	OFF	OFF	ON	&
32K/UPPER MEMORY	0D8000 - 0DFFFF	OFF	OFF	OFF	OFF	&
64K/UPPER MEMORY *	0D0000 - 0DFFFF	OFF	ON	ON	OFF	&
32K/UPPER MEMORY *	0E0000 - 0E7FFF	OFF	ON	OFF	ON	&

*** NOTE - configuration available with hardware revision 1.03 (September 1994) and above.**

& NOTE – Jumper is factory-set correctly, do not change setting

Prior to selecting the memory block that the hardware is mapped into, the system memory resources must be determined. For upper memory, the C800 and D000 segments (96K bytes) are allocated for adapter boards on the ISA platform. Before installing the board, you must find a 32K byte, 64K byte or the complete 256K byte block that is free, starting at the base addresses listed in the table. Both DOS and Windows 3.1 provide a program called Microsoft Diagnostics (MSD) which might assist you in determining which upper memory address space is available.

From the DOS prompt type:

- msd
- Press enter.
- Select the Memory function.

Keep in mind that if multiple boards are being used in the same platform, then sufficient memory space must be available for each board. With four boards, this requires a minimum of 128K free bytes.

If an expanded memory manager (such as EMM386 or QEMM) is being used, reserve the required memory space prior to installing the boards. For

example, if you are using EMM386, the following line in CONFIG.SYS reserves the memory for a board based at offset D000:

DEVICE=C:\DOS\EMM386.EXE X=D000-D7FF

If you are running Windows 3.1 or 3.11, you must include this line in your CONFIG.SYS file. For Windows 95 or Windows NT installations, please refer to the readme file that comes with the driver software for special installation issues. A memory conflict is the most likely reason that a board fails to work. Oftentimes a memory check returns all 1's (FFFF) but this is not always the case. The driver offers some (although limited) assistance to when a memory conflict occurs.

I/O Map

The page register is I/O mapped on the ISA bus. Header locations Block E5, 6-9 and 7-8 select between I/O addresses 200, 220, 300 or 320 hex.

I/O ADDRESS	E5: 6-9	E5: 7-8
200	ON	ON
220	ON	OFF
300	OFF	ON
320	OFF	OFF

Prior to selecting an I/O address, make sure that this address does not conflict with an I/O address of any other adapter board in your computer, including other PC-1553 boards.

Interrupt Line Select (IRQ3-7,9-12,14-15)

Use jumper block E4 to select the interrupt line for the PC. Only one jumper can be in at any time. Interrupt conflict is another of the reasons that PC adapter boards do not work when first installed. Check your PC manual for available interrupts. Also, be sure that each board in your platform PC has a unique IRQ number. The PC-1553 board is set at the factory to IRQ 5, but you may choose any interrupt that is available on your PC. Note that neither the current version of BusTools/1553, nor the current version of BusAPI.DLL or BusAPI32.DLL use this interrupt, however, future versions will, so it is a good idea to correctly setup this interrupt now, and make a note of it for future reference.

IRQ#	Jumper Placement
IRQ3	E4-1 to E4-22
IRQ4	E4-2 to E4-21
IRQ5	E4-3 to E4-20
IRQ6	E4-4 to E4-19
IRQ7	E4-5 to E4-18
IRQ9	E4-6 to E4-17
IRQ10	E4-7 to E4-16
IRQ11	E4-8 to E4-15
IRQ12	E4-9 to E4-14
IRQ14	E4-10 to E4-13
IRQ15	E4-11 to E4-12

Note: Insure that only one of the above jumpers is installed

1553 - Bus Coupling

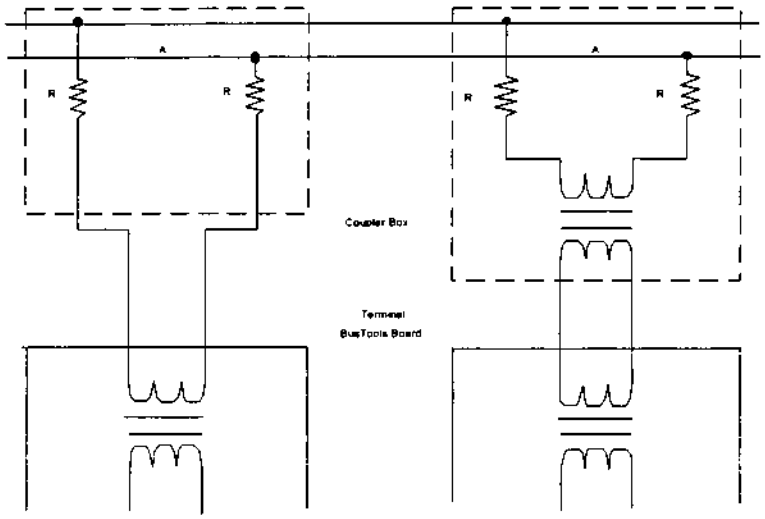
This section is not applicable to PC-1773 boards. The bus outputs can be coupled in two different manners:

- Direct coupled (short stub) - Isolation Transformer 1.4:1
- Transformer coupled (long stub)

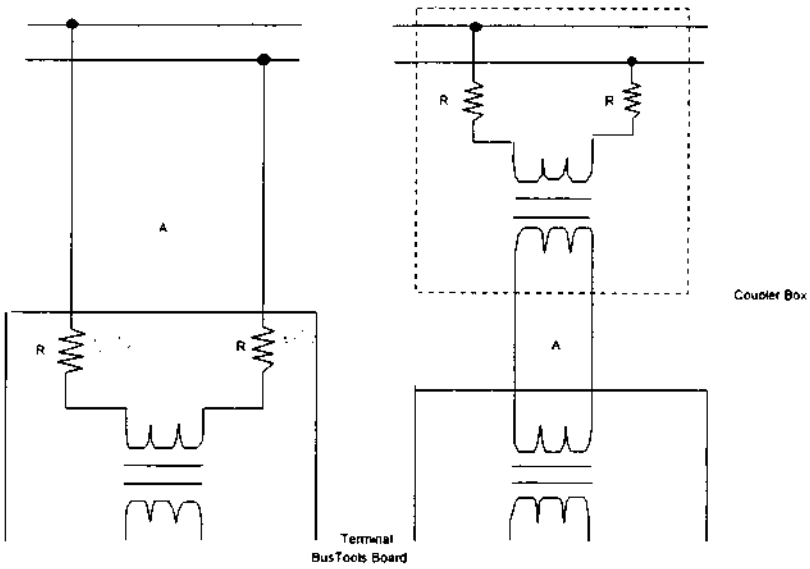
Use direct coupling when the distance from the isolation transformer to the data bus does not exceed one-foot. Use transformer coupling when the distance exceeds one-foot but is less than 25 feet. There are isolation transformers on the board for both direct and transformer coupling.

There are 53.6-ohm isolation resistors on the board for direct coupling. Transformer coupling requires an external transformer and isolation resistor at the junction to the main 1553 bus to match the stub impedance to the bus. (The Air Force mandates that transformer coupling be used.)

The bus coupling (transformer or direct) is controlled by two relays that are selected under software control. See the API routine **BusTools_SetVoltage().**



MIL-STD-1553A Data Bus Interface



a. Direct Coupled (Short Stub) b. Transformer Coupled (Long Stub)

R=Isolation resistor

MIL-STD-1553B Data Bus Interface

Factory Settings

Your board has been configured at the factory for proper operation. The jumper from E5-1 to E5-14 must not be removed from PC-1553 model boards nor inserted into model PC-1773 boards. The two potentiometers at the top of the board have been factory adjusted for proper 1553 operation. Do not readjust them; if you do, you may void your warranty.

However, in an emergency the following procedure may be used to re-adjust the two potentiometers on the PC-1553 card; for further information please contact the factory:

DAC Calibration For PC-1553 BusTools Boards

Offset Adjustment

1. Write all ones to the 12-bit inputs to the DAC, which will produce a zero-voltage output. This can be accomplished by setting up the voltage for 0.0 Volts peak-to-peak on the *BusTools* Configuration menu, or calling the **BusTools_SetVoltage()** function.

2. Place a scope probe on the offset (center) pin of the potentiometer at R8 and adjust the potentiometer until the reading is exactly zero volts.

3. Check that the outside pins of the pot are +12 and -12 volts.

Gain Adjustment

1. Write all zeros to the 12-bit inputs to the DAC, which will produce a maximum voltage output of 9.998 volts. This can be accomplished by setting up the voltage for 6.5 Volts peak-to-peak direct mode or 19.8 Volts transformer coupled mode on the *BusTools* Configuration menu, or calling the **BusTools_SetVoltage()** function.

2. Place a scope probe on the gain (center) pin of the potentiometer at R5 and adjust the pot until the voltage is 9.998 Volts.

CHAPTER 3

IP-1553 Hardware Installation

Installation Procedure

This section tells how to install the IP-1553 board. If possible, use a static free workstation. Before beginning the installation process, inspect the packaging material to insure that the board hasn't been damaged during shipment. If it has, contact the shipper right away.

Remove the disks and set them aside. Then follow the steps in this section:

1. Remove the board from the anti-static bubble pack and lay the board on top of the bag. This prevents static discharge from damaging the board when you change the jumpers.
2. Set the jumpers on the board to the desired settings for your operating system. (See the *Hardware Description* section on the next page for specific settings).
3. If you have purchased an IP-1553 configuration without a carrier board, you must first install the IP on your own carrier board. The connectors are aligned so improper installation is not possible. Re-install the carrier board with the IP module in your system.
4. Connect your MIL-STD-1553 bus to the connectors provided on your carrier board. The IP I/O configuration is defined below.

If this is the last connection on the bus a "Tee" should be used and a 78-ohm terminator should be connected to provide the proper termination for the bus. Otherwise you should use a "Tee" with a cable on each end of the "Tee".

If you are using "transformer" coupling, than simply connect the bus "A" and "B" "stubs" to the connectors on the back of the interface card

IP Connector Description

The IP-1553 is a double size IndustryPack (registered trademark) board designed to IndustryPack Logic Specification revision 0.7.1. It contains four AMP 50-pin plug connectors, part number 173279-3. The connectors are designated IP A Logic (P1), IP A I/O (P2), IP B Logic (P3) and IP B I/O (P4).

IP logic signals are routed through connector P1. Connector P2 is only used for mechanical support. Connector P3 is only used to supply power and ground connections. Connector P4 supplies the I/O, Pin positions for P4 are as follows.

IP Connector P4; Carrier Connector B, C or D (See below)			**CONM-D25 25-Pin D-type Connector**
Pin Number:	**Signal Name:**	**Signal Type:**	**Pin Number:**
1, 39	EXTOUT	TTL Output	21
3, 41	EXTIN	TTL Input	23
5,7,9,11.13,17, 19,21,23	GND	Power Return	1,3,5
8	TAGCLK	TTL	
10	INCMD	TTL	
12	RTAD4	TTL Input	
14	RTAD3	TTL Input	
16	RTAD2	TTL Input	
18	RTAD1	TTL Input	
20	RTAD0	TTL Input	2
22	RTADP	TTL Input	4
25, 37	SHIELDA	N/A	7, 19
27	1553A+	Bi-directional 1553	9
29	1553A-	Bi-directional 1553	11
31	SHIELDB	N/A	13
33	1553B+	Bi-directional 1553	15
35	1553B-	Bi-directional 1553	17
all others	N/A	open	6,8,10,12,14, 16,18,20,22, 24,25

Note that the DB25 connector is wired to the ribbon cable as follows: Pins 19 through 43 (on the ribbon) are connected to pins 1 through 25 (on the DB25).

When connecting the provided transition output cable (part number CONM50-D25) to the carrier, ensure that the proper I/O connector on the carrier board is used. When using the ISA/PCI or VME Bus carriers supplied by Condor Engineering:

1. If the IP is installed in the first dual slot (Slots A and B) the cable needs to be connected to the IP B I/O connector on the carrier.
2. If the IP is installed in the second dual slot (Slots B and C) the cable needs to be connected to the IP C I/O connector on the carrier. This configuration only applies to the ISA-based IP carriers.
3. If the IP is installed in the third dual slot (Slots C and D) the cable needs to be connected to the IP D I/O connector on the carrier.

Signal Descriptions

Transformer versus Direct Coupling

MIL-STD-1553B allows for two methods of connecting a terminal to the bus. Transformer coupling provides noise immunity over long distances and requires external transformer couplers. Direct coupling is often used in the laboratory and requires no transformers between boxes. To set the IP-1553 board for either coupling method, jumpers must be installed. The board comes with factory installed jumpers for the direct coupled mode.

Ensure that only the direct coupling or only the transformer-coupling jumpers are set at any given time, but not both, as follows:

Function	Jumpers Installed
Direct coupled, channel A	E2-1 to E2-16, E2-4 to E2-13
Transformer coupled, channel A	E2-2 to E2-15, E2-3 to E2-14
Direct coupled, channel B	E2-5 to E2-12, E2-8 to E2-9
Transformer coupled, channel B	E2-6 to E2-11, E2-7 to E2-10

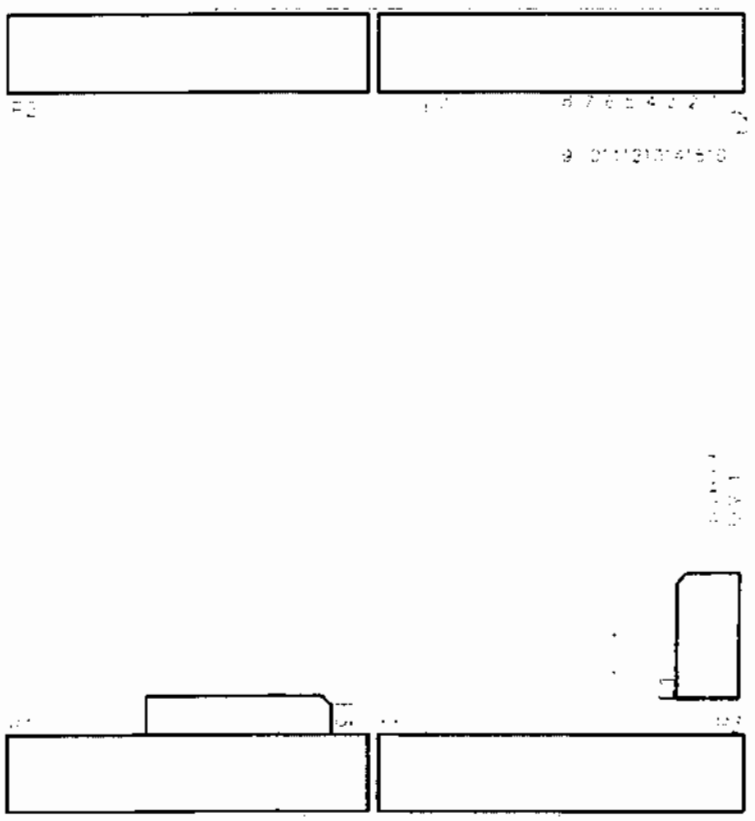
The coupling method is set by these jumpers; the software function BusTools_SetVoltage() may NOT be used to configure the coupling mode (or the output voltage level, which is fixed by the IP hardware to MIL-STD-1553B levels).

MIL-STD-1553B Signals

1553A+ and 1553A- are the differential pair for channel A. SHIELDA is used in most environments to shield the cable used for the stub connections. 1553B+ and 1553B- are the differential pair for channel B. SHIELDB is to shield the channel B pair. Both shields are tied together and to the center tap of the isolation transformers on the IP-1553 board.

IP-1553 Jumper Location

The following illustration shows the location of jumpers on the IP-1553 board.



External Input

EXTIN is an active high TTL signal that must be active for a minimum of 40 microseconds and may stay active indefinitely. It has one of four functions that are programmed in software: BC trigger, RT trigger, BM trigger and Reset Tag Timer.

BC Trigger

After sensing the input to be active, the hardware sets the bc_run bit that starts the Bus Controller. Setting bit 14 of 1553_control_register1, which may be programmed by the software once the board has been set up as a BC, enables this function. Holding the external input signal active or pulsing it again has no affect on the board unless the bc_run bit has been cleared by the software. If the board is currently running as a BC then pulsing this signal has no effect. Multiple buses may be synchronized to within 40 microseconds using this input.

RT Trigger

RT trigger is essentially the same as the BC, but activates all RT's that are previously set up by the application when bit 13 of 1553_control_register1 is set.

BM Trigger

After sensing the input to be active, the hardware starts capturing 1553 data, provided that the software sets the enable_external_trigger bit. Holding this signal active or pulsing it again has no affect on the board unless the software performs another set-up of the bus monitor.

Reset Tag Timer

After sensing the input active the tag timer is reset if bit 6 of 1553_control_register1 is set. A single tag timer is used for the Bus Monitor and all Remote Terminals.

External Output

EXTOUT is an active high TTL signal that goes active for 50 microseconds. Firing this output is under software control. It may be enabled for BM trigger or RT synchronize mode code.

BC Trigger

Fires when the BM trigger occurs when enabled by the external_output_on_trigger flag that is stored in the bm_trigger_buffer in RAM.

RT synchronize mode code

Fires when a RT receives a synchronize mode code, if enabled.

External Hardwired RT Address

This feature is not currently implemented but could be upon user request. The RTAD[4..0] signals are for externally hardwiring the boards to respond as a single RT. The RTADP is the parity signal. If the RT address is hardwired, the board is not able to simulate the Bus Controller or to respond to any other RT address even when so enabled by the software.

IP Carrier Jumper Settings

Please consult the manual supplied with your IP carrier board for proper setting of the jumpers, if any, associated with the carrier. These jumpers set the memory addresses and interrupt settings used by the IP-1553 board.

The PCI carrier jumpers are set at the factory, do not change these settings unless directed to do so by Condor Engineering Technical Support.

When setting up the ISA bus IP carrier, insure that the base address of the carrier is available on your PC and that there are no other boards mapped at that address. Make note of the address and which pair of slots the IP was installed in; it will be necessary to supply this information to the software that will be used to control the board.

Also note if this is a single-function IP or a multi-function IP as the operating software to will also require this information properly control the IP.

IP-1553 Memory Map

The IP module itself uses a 2K-byte block on the carrier board. The actual addresses used depends on the carrier board being used. The IP IO address is mapped into memory space, again depending on the carrier.

When using the ISA bus IP carrier supplied by Condor Engineering, you will map the carrier into a 16K byte region in upper memory, using the instructions supplied with the carrier. The factory default is to map the carrier at address D0000 through D3FFF; the IO address is not used (since the carrier maps the IO port into memory space at an address known by the software driver).

The supplied software supports the ISA Bus IP carrier, with the IP module installed into any available pair of connectors.

When using the PCI bus IP carrier supplied by Condor Engineering, the BIOS will map the carrier into memory space at an address above the total amount of RAM installed in the computer. The IO address in not used since the carrier maps the IO space into memory space. The supplied software supports the PCI carrier with the IP module installed into any available pair of connectors, under 16- or 32-bit Windows®. Operation under DOS is not supported.

When using the VME bus IP carrier supplied by Condor Engineering, the user must map the memory address of the carrier into A32 space and the IO address of the carrier into A16 address space. The IP module may be installed into either slot A or slot C.

See the section "**BusTools_API_InitExtended()**" for further information about configuring the software for the various combinations of modules and carriers.

See the section "PC-1553/PC-1773 Memory Map" for information about setting up memory exclusions under Windows 3.x and DOS.

CHAPTER 4

Initialization and Global Routines

This chapter describes the initialization and setup operations required to operate hardware and API routines. Additionally, the routines that have "global" significance (e.g. board setup and error reporting) are described.

Initialization and Shutdown

The first task in the initialization routine is to perform a hardware reset of the 1553 interface board.

1. Toggle the "RESET" bit on the board.
2. Initialize the RAM registers (certain RAM registers are actually constants, which must be set up before the microcode can operate properly).
3. Load the Micro Code into the on-board Writable Control Store, if the board supports WCS.
4. Initialize global parameters to their default values (e.g. voltage settings).
5. Test and clear the memory on the board.
6. Create and initialize certain structures in hardware memory.

The **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines handles initialization of the board and clears all memory on the board. The Interrupt Queue is created and initialized. Then, in order to initiate interrupt processing as supported by the API, the ten-millisecond timer is started. (See the "Interrupt Queue" chapter in the Hardware Reference Manual for more details about how the interrupt queue works.)

When the application is done with the API routines, the **BusTools_API_Close()** routine ***must*** be called.

THIS ROUTINE MUST BE CALLED – Because of the way in which the API handles interrupts (with the ten-millisecond Windows® multimedia timer), the application must call this routine to turn off the timer. If the timer is not turned off prior to terminating the application, the

Windows® environment may be left in an unstable state. This may create unpredictable (and undesirable) side effects. Future versions of the API will use the hardware interrupts, failure to call this function then will cause the OS to crash when the interrupt function attempts to call code which is no longer in memory,

Global Parameters

There are several parameters that control "global operation" of the hardware and firmware. You can set these parameters by calling the routines described below. While there are default values for each of these parameters, it is a good idea to call these routines with the desired settings in order to be sure of repeatable behavior of your 1553 simulation.

- **BusTools_SetBroadcast():** RT address 31 is normally reserved for "broadcast" messages. However, it is possible to operate the board with "broadcast" operations disabled. In this case, RT address 31 is just like any other address. The setting of this global parameter is handled with this routine. The default setting is for "broadcast" operations to be disabled. This function should be called after **BusTools_API_Init()** or **BusTools_API_InitExtended()** has been called, but before **BusTools_RT_Init()** is called.
- **BusTools_SetInternalBus():** The board normally uses the applications defined simulation to drive the external 1553 bus. However, it is possible to run the entire simulation internally, ignoring the external bus entirely. The setting of this global parameter is handled with this routine. The default setting is for the external bus to be active. This function should be called after **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines have been called, but before any of the **BusTools_XX_StartStop()** functions are called.
- **BusTools_SetOptions():** This routine is used to set how the board handles various 1553 options. It should be called just after **BusTools_API_Init()** or **BusTools_API_InitExtended()** is called. The available options are:
  - **Monitor Illegal Commands:** This option specifies how a RT should respond to an illegal command (one with an invalid subaddress or word count). If the option is set, the RT returns a status word to the BC with the Message Error bit set (and suppress or ignore any data words). If the option is not set, 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:** This option specifies what should be done if a RT detects an illegal command (one with

an invalid subaddress or word count). If the option is set, the RT microcode generates an interrupt if an illegal command is detected. The default setting for this parameter is disabled.

- **Treat Illogical Commands as Invalid:** 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 an invalid command. If this bit is set, then illogical commands is treated as invalid (the command is ignored and no message is processed by the RT). If this bit is not set, then illogical commands is treated as illegal (the RT response depends on the setting of the "Monitor Illegal Commands" option). The default for this parameter is disabled.
- **Reset Timetag on Sync:** This option specifies that if a Sync mode code is detected, the hardware timetag counter is reset. Since the timetag is global to all components simulated on the board, this option should be used sparingly (that is, it should only be used if the board is simulating a single RT). The default for this parameter is disabled (the Sync modecode has no effect on the timetag counter).

- **BusTools_SetSa31():** According to the 1553 specification, the board treats any message to or from subaddress 0 as a modecode message. The specification provides the option of treating any message to or from subaddress 31 as a modecode message. The setting of this option is handled with this routine. The default setting for this parameter is to have subaddress 31 be a modecode. This function should be called after the initialization function **BusTools_API_Init()** or **BusTools_API_InitExtended()** has been called, but before the RT simulation initialization function **BusTools_RT_Init()** is called.
- **BusTools_SetVoltage():** The voltage level and electrical coupling for how the board drives the external 1553 bus is set by this routine. 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 can be used to test external hardware with low voltage situations. The setting of the voltage parameters is handled with this routine. The default setting is for "direct coupled" at 6.5 volts, for "transformer coupled" is 19.8 volts. This function should be called after **BusTools_API_Init()** or **BusTools_API_InitExtended()** has been called, but before any of the **BusTools_XX_StartStop()** functions are called.

General Purpose Routines

- **BusTools_StatusGetString():** All API routines return a status code indicating if the routine executed successfully or detected an error during execution. If the status is non-zero, an error was detected. This routine returns a pointer to an ASCII string that describes the specified status code. This routine is typically used to present you an explanation of the error code. This routine can be called at any time, even before **BusTools_API_Init()** or **BusTools_API_InitExtended()** is called. All of the status codes returned by the BusTools API functions are supported by this function, even those which might have been created after this manual was last updated.
- **BusTools_TimeGetString():** This routine is used to convert the PC1553_TIME structure to an ASCII character string. The string provides a "human-readable" form for the time (as measured from the beginning of the simulation).

Example Programs

Supplied with the API software is a file called "DEMO.C". This file contains an example program, suitable for compiling, linking and running under MS-DOS®, which demonstrates most of the API functions. This program can be used as an example of programming either the PC-1553/PC-1773 or the IP-1553. Note that, as written, this program supports as many as two boards or IP modules, in any combination.

The DEMO.C program is also supplied as an executable, suitable for running directly from the DOS prompt, either under MS-DOS or from a DOS box under Windows 3.x or 95, and as a Windows 3.x executable, and as a 32-bit Windows 95/NT executable. These executables are named DEMO16.EXE and DEMO32.EXE. They are all built from the same source code file, DEMO.C

When examining the DEMO.C code, note that the user inputs which cause the API functions to be executed are arranged in the normal order (letters A through ...) one would use when calling the API functions from a user-written program. This order applies equally to the single function modules as well.

CHAPTER 5

Bus Monitor

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

This chapter describes the BM functionality and the API routines used to access the BM functions.

BM Hardware Operation

The primary function of the BM is to record information about each message (both the actual data contained in the message and status information about how the data is being transmitted) as it appears on the 1553 bus. The BM microcode records the information for each message in a BM Message Buffer set up in memory. The application program must initialize a list of BM Message Buffers in BusTools memory and tell the microcode where the list is located (by storing the BusTools memory address of the list in one of the RAM Registers). The list of BM Message Buffers is a circular linked list (that is, each buffer has a pointer to the location of the next buffer in the list and the last buffer has a pointer to the first buffer in the list).

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

Finally, the microcode sets an internal register to point to the next Message Buffer in the list.

The interrupt queue structure is defined in the "*Interrupt Queue*" chapter of either the **PC-1553/PC-1773 Hardware Reference Manual** or **IP-1553 Hardware Reference Manual**. Once an interrupt is received, the

application's Windows® task assigned to handle BM interrupts receives an Interrupt Message containing the BusTools memory address of the BM Message Buffer which generated the interrupt (See the function **BusTools_InterruptRegister()**). The Windows task converts the memory address to a specific BM Message Buffer (using the **BusTools_BM_MessageGetid()** routine). The BM Message Buffer is then read from BusTools memory (using the **BusTools_BM_MessageRead()** routine). The application can then process the message as required (displaying all or part of the message or updating statistics, etc.).

Note that as messages appear on the 1553 bus, the microcode sequences through the BM Message Buffer list, one buffer after another. There is no checking to see if the next Message Buffer has already been used or if the application program has had a chance to look at that data. Thus, based on the expected message rate and time required to process the information in each message, the application must allocate enough BM Message Buffers to ensure that each buffer can be processed before the microcode "wraps-around" in the list. The *BusTools* GUI uses a Bus Monitor buffer that is 250 messages long. This is sufficient for operation under 16-bit Windows®, and is probably excessive for 32-bit Windows® applications.

The API uses the timetag field in the Bus Monitor buffer to determine if the buffer has wrapped around: It sets the timetag field to zero when it reads the last message in the buffer, then the next time it reads messages it tests the timetag to see if it has been set by the microcode.

Optionally, the microcode can filter out certain messages. The BM Filter defines the filtering which is performed. The filtering operation is based on the message command word (or the first command word in the case of "RT-to-RT" messages). Each possible command word can be enabled or disabled for BM recording into the Message Buffer list. In addition, the filtering operation can be set up to record every "nth occurrence of each message type (where the "message type" is determined by the command word in the message).

Finally, the application can define one or more trigger events used to tell the microcode to start the BM recording operation. Before the defined trigger event occurs, no information is recorded to the BM Message Buffers. Once the trigger event occurs, all of the messages that follow are recorded, subject to the BM Filter. The trigger events are based on checking a particular command, data or status word of incoming messages against a specified value.

For each message recorded by the BM, the microcode stores the following items in the BM Message Buffer:

- each word of the message (command, data and status) with a 16-bit BM Word Status for that word
- the message time tag (the time counter as of the mid-zero crossing of the parity bit of the command word)

- the Message Status Bits for the entire message.

In addition, the RT response time is measured and placed in the Message Buffer. It should be noted that the microcode records information into the Message Buffer as it sees it on the bus. Therefore, the only word of the message which is always in the same place is the command word (the first command word in the case of "RT-to-RT" messages) and its BM Word Status Bits. After that, the contents of the Message Buffer depend on the type of message and number of words in the message.

For example, for a "receive" message with 10 data words, the data immediately following the Word Status Bits for the command word is the data value and BM Word Status Bits for each of the 10 data words (a total of 20 words). Then the response time required for the RT to return its status word is recorded. Finally, the status word and its BM Word Status Bits are recorded. For a "transmit" message, the response time and status information are recorded before the data (because the status word appears before the data in the message).

Luckily, the application need not worry about this complexity. The BusTools API returns the BM Message Buffer as fixed length and formatted structure which can be used more easily by the application. (See the API_BM_MBUF structure).

This same structure is used to write the Bus Monitor data to the BusTools Bus Monitor Recording File.

BM Software Operation

The BusTools API routines that handle BM operations are described here. Detailed calling sequences and error status information can be found in a separate chapter of this manual.

The BusTools API design is based on the premise that, no matter what system is being modeled, you want a Bus Monitor function to be active. Therefore, certain operations which are required for the BusTools hardware to function properly (such as memory initialization, interrupt queue definition and error injection buffer definition) are handled by the Bus Monitor initialization routine.

As a result of this assumption, none of the API routines for BC or RT access can be executed until the Bus Monitor has been initialized. Thus, after BusTools hardware initialization has been completed (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routine), the next routine which should always be called is **BusTools_BM_Init()**. This routine sets up the BM for operation. Specifically, it allocates and initializes the default BM filter (which is set to enable the recording of all messages) and the default BM trigger (which is set to start recording immediately).

The application should then create the required number of BM Message Buffers using the **BusTools_BM_MessageAlloc()** routine. The caller must specify the required number of Message Buffers. This number should be large enough to ensure that the expected message traffic on the 1553 bus doesn't "wrap-around" in the list before the application can retrieve the data and process or store it as required. If there is enough memory on the BusTools board, the routine creates the requested number of Message Buffers and links the buffers together. If there isn't enough memory for the requested number of buffers, the routine creates as many Message Buffers as possible and returns the actual number created to the caller.

You must also specify the Interrupt Enable Bits to be set in each Message Buffer. This value determines which status conditions generate an Interrupt Message to the application. Sometimes, you want to get an interrupt for every message detected on the 1553 bus (this would be done by setting the "PC1553_INT_END_OF_MESS" bit). Other times, you want to get interrupts only if a parity error is detected (this would be done by setting the "PC1553_INT_PARITY" bit). Any combination of bits can be set to specify the conditions that generate an Interrupt Message.

After the Message Buffers have been created, the application typically sets the desired parameters for the BM Filter. This is done with the **BusTools_BM_FilterWrite()** routine. With this routine, you specify the filter settings for a particular RT sub-unit (a RT sub-unit is a specific RT address / subaddress / transmit-receive flag combination). If required, this routine can be called for all possible RT sub-unit combinations. Note that the default BM filter is to allow all messages to be recorded. If this setting is acceptable for your application, this routine need not be called.

Next, the BM trigger must be defined using the **BusTools_BM_TriggerWrite()** routine. You must fill in the trigger definition structure prior to making this call. Note that the default BM trigger is for all messages to be recorded. If this setting is acceptable for your application, this routine need not be called. There is one BM trigger buffer allocated in the hardware memory.

Under 16- or 32-bit Windows®, the **BusTools_InterruptRegister()** function can be called to register a window (specified via its handle) which will receive a PostMessage() whenever an interrupt occurs from the Bus Monitor. This application window function can then read the 1553 message that caused the interrupt as necessary for the application. See the **BusTools_InterruptRegister()** function for more details.

Finally, after the BM has been properly initialized, it should be started. This is done using the **BusTools_BM_StartStop()** routine. The BM begins operations, recording any messages detected on the 1553 bus (subject to the BM filter and BM trigger parameters specified by the application). If any message meets the criteria specified with the Interrupt Enable Bits, an Interrupt Message is generated. (See the "Interrupt

Queue" chapter in the Hardware Reference Manual for more details about how the interrupt queue works.)

BM operation continues until the application stops the BM (using the **BusTools_BM_StartStop()** routine).

BM Recording (16/32-Bit Windows® Only)

The BusTools API can be setup to send a periodic Recording Message to a specific Windows® task. If enabled, a BM Recording Message is sent to the Windows® task every 500 milliseconds, or when 64 Kbytes of messages have been received. In response to the BM Recording Message, the application should process the messages which have been recorded in the Message Buffer list (but which may or may not have generated Interrupt Messages of their own). At its option, the application can transfer the new BM Message Buffers to a PC resident memory buffer, write the messages to disk or tape, or just keep statistics on the messages which have been detected on the 1553 bus.

If the periodic Recording Message is desired, the application must register the handle of the Window and the message id to be used by the PostMessage() Windows® function using the **BusTools_RecordRegister()** routine. Once this routine is called, the specified Windows® task receives the periodic message until the routine is called again to disable the sending of Recording Messages (the "wFlag" parameter determines if the periodic Recording Messages are to be enabled or disabled).

Within the specified Windows® task, the most recent list of BM Message Buffers can be retrieved from the BusTools memory using the **BusTools_BM_MessageReadBlock()** routine. This routine keeps track of which messages are the most recent and also returns an error to the application if a buffer "wrap-around" has been detected.

The routine returns the latest BM Message buffers in an application supplied buffer. The application can then store the information on disk or tape. If the application wishes to display the information or access any of the information in the buffers, the information from the BM Message Buffer can be directly manipulated and displayed. The routine **BusTools_TimeGetString()** can be used to convert the time-tag to a printable ASCII string.

If the **BusTools_RecordRegister()** routine has not been called, the API will not process the Bus Monitor message buffer. Calling the **BusTools_BM_MessageReadBlock()** routine will therefor not return any data.

CHAPTER 6

Remote Terminals

The Remote Terminal (RT) function on the BusTools board provides the ability to simulate the activities of one or more RT's on the 1553 bus. The definition of each RT includes:

- specifying the legal transmit and receive sub-addresses
- specifying the legal word counts for messages to and from each sub-address
- supplying one or more data buffers to capture data sent to each sub-address
- supplying one or more data buffers to be transmitted by each sub-address
- specifying the modecodes to be supported by the RT
- specifying the interrupt enable bits for each of the message lists

This chapter describes the RT functionality and the BusTools API routines used to access that functionality.

RT Hardware Operation

For each possible RT (addresses 0 to 31), there is a four word Address Control Block in BusTools memory. Each Address Control Block contains information used to control microcode operation with respect to a particular RT. The most important entries of the Address Control Block are the enable / disable bits. There is one enable / disable bit for each channel. If both bits are disabled, the RT simulation ignores any messages with that RT address. If either or both bits are enabled, then the microcode responds to commands with that RT address if the message is received on an enabled channel.

The Address Control Block also contains the following:

- a 16-bit status word (initialized by the application and maintained by the microcode). This is the message status word transmitted by the RT in response to any message to the RT. The microcode modifies the "Message Error", "Broadcast Command Received",

and "Terminal Flag" bits of this status word as required by the 1553 specification. The other bits are set by the application and are not modified by the microcode.

- a 16-bit last command word (maintained by the microcode in order to be able to respond to the "Transmit Last Command" modecode).
- a 16-bit Built-In-Test word (initialized by the application and used by the microcode to respond to the "Transmit BIT Word" modecode).
- a 1-bit "inhibit-terminal-flag" (initialized by the application and maintained by the microcode in response to the "Inhibit Terminal" and "Override Inhibit Terminal" modecodes).

If the simulation for a particular RT address is enabled (by enabling either channel in the Address Control Block), then any message with that RT address is processed by the microcode. The processing starts by checking that the command word is legal – that is, the RT address, sub-address, word count and transmit/receive flag form a legal combination. This is done by checking the command word against the contents of the Control Buffer defined for the specified RT sub-address. There are 64 Control Buffers for each RT address (one transmit and one receive for each possible sub-address). These Control Buffers are set up by the application and contain a single bit for each possible word count. If the bit is set, then that word count is legal for the specified RT sub-unit (RT address / sub-address / transmit-receive flag combination). The Control Buffer also contains the address of the first Message Buffer defined for this RT sub-unit.

If the microcode determines that the command is illegal, then one of two things will happen. The action is based on the setting of the "Monitor Illegal Commands" flag (see the "Initialization" chapter). If this flag is set to "Monitor Illegal Commands", then the microcode ignores any data contained in the message (in the case of a transmit command) and sets the "Message Error" bit of the status word back to the BC. If the flag is set to not "Monitor Illegal Commands", then the microcode continues processing normally, either receiving the data into or transmitting the data from a "garbage" data buffer.

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

The final level of complexity is that each RT sub-unit can have multiple RT Message Buffers defined. The "Message Buffer" described above is actually a circular linked list of Message Buffers (*that is, each RT* Message Buffer contains the address of the next Buffer in the list and the last Buffer contains the address of the first Buffer). Each time the

microcode sees a command word for that RT sub-unit, the next Message Buffer in the linked list is used for the message. It is the responsibility of the BusTools API to initialize the Message Buffers properly.

After each message is processed, the microcode checks the Message Status Bits for this message against the set of Interrupt Enable Bits specified in the RT Message Buffer. If there are any matching bits (if the logical AND of the two 32 bit values is non-zero), an Interrupt Message is added to the Interrupt Queue (See the "Interrupt Queue" chapter in the Hardware Reference Manual for more details about how the interrupt queue works).

It should be noted that the operation of the RT simulation microcode is an "all-or-nothing" type of operation. Prior to turning on the RT simulation, the application must have initialized (at a minimum) Address Control Blocks for **all** RT's. For any RT's which are not to be simulated, the channel enable / disable bits in the Address Control Blocks must be set to "disabled". In addition, a 2048 word list of Control Buffer addresses must be initialized. For any disabled RT's, the addresses listed in this list can point to the same default "no-op" Control Buffer. These functions are automatically performed by the call to **BusTools_RT_Init().**

RT Software Operation

The BusTools API routines, which control RT operations, are described here. Detailed calling sequences and error status information can be found in a separate chapter of this manual.

Prior to calling any BusTools RT routine, the RT-specific global parameters must be initialized by calling the following functions:

1. BusTools_SetBroadcast()
2. BusTools_SetSa31()

The RT simulation must then be initialized. This is done with the **BusTools_RT_Init()** routine. This routine sets up the default (disabled) RT Address Control Blocks, the default (disabled) RT Control Buffers and the "garbage" RT Message Buffer used for all disabled or illegal RT messages. At this point, the RT simulation microcode can be enabled with the **BusTools_RT_StartStop()** routine (though all RT's are disabled). Both of these routines are "global" in nature in that they control the simulation of **all** RT's.

In order to define a RT in the BusTools memory, the following steps should be followed. First, the Control Buffers for any legal sub-unit (sub-address and transmit / receive flag) for this RT must be created. In addition, the linked list of RT Message Buffers for this sub-unit must be created. These tasks are handled by the **BusTools_RT_CbufWrite()** routine. The application can call this routine up to 64 times (32 sub-addresses, each with a transmit and a receive type) per simulated RT. This

routine is called to establish legal modecodes as well as word counts. The only thing this routine doesn't do is to establish legal broadcast messages (if broadcast messages are to be allowed – see the "Initialization" chapter).

Next the RT Broadcast control buffers must be initialized (if broadcast messages are to be allowed). You must call the function **BusTools_RT_CbufbroadWrite()** once for each RT, for each subaddress which is to be enabled for Broadcast Receive or Transmit (Mode Codes only). Do not attempt to read or write the Broadcast control buffers if broadcast is not enabled; the function will return an error.

Next, the RT Message Buffers should be filled as required. For all message types (transmit or receive), the RT Message Buffer must contain the desired Interrupt Enable Bits and the Error Injection Buffer pointer (if desired). In addition, for transmit sub-units, the RT Message Buffer(s) should contain the data to be transmitted. The contents of each RT Message Buffer is initialized with the **BusTools_RT_MessageWrite()** routine. This routine should be called for every Message Buffer defined for every sub-unit of the RT.

The BusTools API defines a default (or "garbage") Message Buffer for each RT. While the data values in this default buffer probably do not matter (since so many Control Buffers usually point to this Buffer), the Interrupt Enable Bits should be set by the application. For example, the application could set the Interrupt Enable Bits to generate an interrupt any time this buffer is accessed (using the "PC1553_INT_END_OF_MESS" bit). This would let the application know anytime that an uninitialized RT sub-unit is accessed. The default RT Message Buffer can be set using the **BusTools_RT_MessageWriteDef()** routine.

After the RT Control Buffers and Message Buffers have been initialized, the Address Control Block should be written (using the **BusTools_RT_AbufWrite()** routine). At a minimum, the channel enable bits should be set to enable at least one channel. In addition, the application can initialize the status word and BIT word as required. Once one or both channel enable bits have been set (and assuming the RT simulation has been started with the **BusTools_RT_StartStop()** routine), the RT simulation for the specified RT begins.

The only other action required of the application software is to handle interrupts received from the RT simulation. The interrupt queue structure is defined in the "*Interrupt Queue*" chapter of either the **PC-1553/PC-1773 Hardware Reference Manual** or **IP-1553 Hardware Reference Manual**. Once an interrupt is received, the application's Windows® task assigned to handle this RT's interrupts receives an Interrupt Message containing the BusTools memory address of the RT Message Buffer which generated the interrupt (See the function **BusTools_InterruptRegister()**). The Windows task converts the memory address to a specific RT Message Buffer (using the **BusTools_RT_MessageGetid()** routine). The RT Message Buffer is then read from BusTools memory (using the **BusTools_RT_MessageRead()** routine). The application can then

process the message as required (displaying all or part of the message or updating statistics, etc.).

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

If the simulation for all RT's is to be discontinued, the entire RT simulation can be turned off using the **BusTools_RT_StartStop()** routine. Starting or stopping either a single RT or the entire RT simulation has no effect on the Bus Controller simulation, or the operation of the Bus Monitor.

CHAPTER 7

Bus Controller

The Bus Controller (BC) function on the BusTools board provides the ability to simulate the activities of the BC on the 1553 bus. This chapter describes the BC functionality and the BusTools API routines used to access the BC functions.

BC Hardware Operation

The BC controls all activity on the 1553 bus. It is up to the BC to determine what messages appear on the bus and when they appear. The messages are organized by the application into one or more groups called Minor Frames. Each message in a Minor Frame is executed sequentially, with an optional delay between messages. The execution of each Minor Frame is started at an application specified time interval.

For example, an application could define three Minor Frames, one with 10 messages, one with 25 messages and the last with one message. If the Minor Frame rate is set to 100 milliseconds, then a Minor Frame is started every 100 milliseconds.

When the BC simulation is started, Minor Frame #1 is executed immediately (the 10 messages in that Minor Frame are executed). Then the BC simulation suspends itself until the beginning of the next 100 millisecond time interval at which time Minor Frame #2 is executed (the 25 messages in that Minor Frame are executed). Once again, the BC simulation suspends itself until the beginning of the next 100 millisecond time interval, at which time Minor Frame #3 is executed (the one message in that Minor Frame is executed).

If the BC has been set up to execute the Minor Frames only once ("oneshot" mode), then the BC simulation is halted at this time. If the BC has been set up to "loop" on the defined Minor Frames, then the BC simulation suspends itself until the beginning of the next 100 millisecond time interval at which time it restarts with Minor Frame #1 (or wherever the start of the loop has been defined).

It is the responsibility of the application to ensure that the contents of each Minor Frame can, in fact, be executed within the specified Minor Frame time. If a Minor Frame takes longer than the specified Minor Frame time, then the BC simulation sets the "Minor Frame Overflow" bit and ceases operation.

Each message within a Minor Frame is defined with a BC Message Buffer. Each BC Message Buffer contains the definition of the 1553 message (the command word), a pointer to the data buffer (either containing data to be sent to the RT or to be filled with data received from the RT) and is filled with status information about the message after the message is executed. In addition, a pointer to an Error Injection Buffer is specified which is used to inject one or more errors into the message (see the "Error Injection" chapter). In addition, the time to delay before starting the next message is specified.

After each message is executed, the microcode stores the Message Status Bits for this message in the BC Message Buffer. The microcode then checks these Message Status Bits against the BC Interrupt Enable Bits. If there are any matching bits (if the logical AND of the two 32 bit values is non-zero), an interrupt block is added to the Interrupt Queue and an Interrupt Message is sent to the Windows® task assigned to handle BC interrupts. It is up to the application to ensure that the appropriate information is retrieved from the BC Message Buffer before that particular message is re-executed by the microcode (if the BC has been set up to "loop").

Each BC Message Buffer can specify either one or two data buffers to be used with the message. If only one buffer is specified, then each time this message is executed, the data in that buffer is used (either sent to the RT or filled in from data transmitted by the RT). This mode can be used to increase the number of BC messages that will fit within the memory on the BusTools board.

If two buffers are specified (and the hardware is version 1.03 or below), then the buffers are used in a "ping-pong" mode. That is, the first time this message is executed, the first data buffer is used. The second time it is executed, the second data buffer is used. The third time it is executed, the first buffer is re-used. With this design, the application has a little extra time to process the data received from a RT in transmit messages.

If two buffers are specified (and the hardware is version 1.04 or above), then the buffers are used as specified by the "Use Buffer A" bit in the message. If the "Use Buffer A" bit is set, then the first buffer is used, if it is clear, then the second data buffer is used. This allows the software to write to one of the data buffers, then command that it be used.

A special form of the BC Message Buffer provides the ability to specify a "conditional branch" within the BC Message Buffer list. Using the parameters specified in the Conditional Message Buffer, the microcode tests a word (command, data or status) of the previous message and

change the order of message execution based on the results of the test. The Conditional Message Buffer must specify the address within BusTools memory to be checked, a mask to be ANDed with contents of the memory location, and a data value to be compared with the results of the mask operation. If there is a match (the data value matches the masked contents of the specified memory location), control within the BC Message Buffer list "jumps" to the specified location.

Another form of the BC Message Buffer is the "NOP" message. This message performs no action, other than specifying the address of the next message to be executed. It can be used to skip messages in the bus list, under the control of the programmer.

BC Software Operation

The BusTools API routines that handle BC operations are described here. Detailed calling sequences and error status information can be found in a separate chapter of this manual.

Prior to calling any BusTools BC routine, the BC simulation must be initialized. This is done with the **BusTools_BC_Init()** routine. This routine sets the channel on which the BC operates, the Interrupt Enable Bits for all BC messages, the Minor Frame frequency rate, and the "no-response" and "late-response" time-out periods. It should be noted that this routine should not be called until the BM initialization routine has been called (which routine is responsible for initializing BusTools memory).

The call to the **BusTools_BC_Init()** routine also sets up the interrupts for the Bus Controller mode. Whenever a BC message on the 1553 bus meets the conditions specified in the **BusTools_BC_Init()** routine call, an interrupt entry is written into the hardware interrupt queue.

The interrupt queue structure is defined in the "*Interrupt Queue*" chapter of either the **PC-1553/PC-1773 Hardware Reference Manual** or **IP-1553 Hardware Reference Manual**. Once an interrupt is received, the application's Windows® task assigned to handle BC interrupts receives an Interrupt Message containing the BusTools memory address of the BC Message Buffer which generated the interrupt (See the API function **BusTools_InterruptRegister()**). The Windows task converts the memory address to a specific BC Message Buffer (using the **BusTools_BC_MessageGetid()** routine). The BC Message Buffer is then read from BusTools memory (using the **BusTools_BC_MessageRead()** routine). The application can then process the message as required (displaying all or part of the message or updating statistics, etc.).

The next step in setting up the BC is to allocate and initialize the list of BC Message Buffers. BC Message Buffer allocation is performed with the **BusTools_BC_MessageAlloc()** routine. The application must specify the

number of BC Message Buffers to allocate. Then, each BC Message Buffer which will be used must be initialized with the **BusTools_BC_MessageWrite()** routine. Unused message buffers need not be initialized.

Normally several extra message buffers will be allocated, these buffers may then be used for aperiodic messages by initializing them to the required message parameters, and insuring that the next buffer pointer is zero in the last message in the chain.

The BC Minor Frames are defined using the "BC_CONTROL_MFRAME_BEG" and "BC_CONTROL_MFRAME_END" bits of the "control" word of the BC message structure. The first Message Buffer of each Minor Frame must be marked with the "BC_CONTROL_MFRAME_BEG" bit. The last Message Buffer of each Minor Frame must be marked with the "BC_CONTROL_MFRAME_END" bit. Finally, the last message in the list must be marked with the "BC_CONTROL_LAST" bit.

As each message is defined, a pointer must be specified to the next message in the list to be executed (typically, this is the next message in the list). The hardware expects to see a BusTools memory address for this entry. However, since the BusTools API handles all memory addressing requirements, the application need only refer to messages by number (message #0 is the first message in the BC Message Buffer list). These message numbers are also used for Conditional Messages. Additionally, each message

Once the Message Buffers have been filled in, the BC simulation can be started using the **BusTools_BC_StartStop()** routine.

BC Oneshot Operation

The BusTools API also supports a simplified method for managing BC operation. Namely, if the application just needs to execute one or more BC messages a single time in order to get a set of data to or from RT's on the bus, the BusTools API supports a "oneshot" mode of operation. The application specifies a list of BC messages to be executed. The list is executed and the results (including data and status) are returned to the caller. Using the "oneshot" mode, there can be no BM or RT simulations executed, and no interrupt processing is supported.

To operate in the "oneshot" mode, the application must initialize the BusTools API software (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines). Then, the BC simulation must be initialized (using the **BusTools_BC_OneShotInit()** routine), setting the BC channel and time-out parameters. Finally, the BC microcode is executed (with the **BusTools_BC_OneShotExecute()** routine). The application must specify the list of BC messages to be executed. The list

is downloaded to the BusTools board, the list is executed, and the results (data and status) are returned in the user-supplied buffer.

There are only a few other BusTools API routines that can be executed when operating in the "oneshot" mode. These are specified in the detailed routine descriptions elsewhere in this manual.

Example

The following example shows the sequence of BusTools API calls required to execute a short set of BC messages designed to send a data message to RT #1, subaddress #1 first on Bus B, then on Bus A. This example is provided on the installation disks in the file "1SHOTDMO.C".

```
#include <windows.h>
#include "busapi.h"
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
void main( int argc, char **argv )
{
 int          nStatus;      // Status returned from the API.
 API_BC_MBUF  message;
 BT_UINT      wFlag;
 int          i;                    // Loop counter
 long         baseaddr;     // User-supplied board base address.
 int          ioaddr;       // User-supplied board base address.

 /* If the parameters there, convert them, else ask the user */
 if ( argc != 3 )
 {
  printf("Enter the 5-digit HEX base address\n");
  printf(" of the PC-1553 Board (ex: D0000): ");
  scanf("%lX", &baseaddr);
  printf("Enter the 3-digit HEX IO address\n");
  printf(" of the PC-1553 Board (ex: 200): ");
  scanf("%i", &ioaddr);
 }
 else
 {
  /* Now convert the params */
  sscanf( argv[1], "%lx", &baseaddr );
  sscanf( argv[2], "%i", &ioaddr );
 }
memset((void *)&message, 0x00, sizeof(API_BC_MBUF));
```

```
  wFlag = 1;          // Enable the hardware version of the API.
  printf("Opening the BusTools API...\n");
  if ( (nStatus = BusTools_API_Init(0, 0xD0000L, 0x320, &wFlag))
          != API_SUCCESS )
  {
    printf("Error from API_Init() = %4.4X\n", nStatus);
    printf(BusTools_StatusGetString(nStatus));
    printf("\n");
  }

if ( (nStatus = BusTools_BC_OneShotInit(0, 1, 24, 20)) != API_SUCCESS )
 {
    printf("Error from BC_OneShotInit() = %4.4X\n", nStatus);
    printf(BusTools_StatusGetString(nStatus));
    printf("\n");
  }

if ( (nStatus = BusTools_SetBroadcast(0, 0)) != API_SUCCESS )
  {
    printf("Error from SetBroadcast() = %4.4X\n", nStatus);
    printf(BusTools_StatusGetString(nStatus));
    printf("\n");
  }

if ( (nStatus = BusTools_SetInternalBus(0, 0)) != API_SUCCESS )
  {
    printf("Error from SetInternalBus() = %4.4X\n", nStatus);
    printf(BusTools_StatusGetString(nStatus));
    printf("\n");
  }

if ( (nStatus = BusTools_SetSa31(0, 1)) != API_SUCCESS )
  {
    printf("Error from SetSa31() = %4.4X\n", nStatus);
    printf(BusTools_StatusGetString(nStatus));
    printf("\n");
  }

if ( (nStatus = BusTools_SetVoltage(0, 1980, 1))!= API_SUCCESS )
  {
    printf("Error from SetVoltage() = %4.4X\n", nStatus);
    printf(BusTools_StatusGetString(nStatus));
    printf("\n");
  }

  // Setup the message for the One Shot Execute mode.
```

```
message.control = BC_CONTROL_MESSAGE |
        BC_CONTROL_BUFFERA |/* Use first buffer, e.g., data[0] */
        BC_CONTROL_CHANNELB; /* Transmit on Channel B.    */

message.mess_command1.wcount  = 6;   /* 6 Data words */
message.mess_command1.subaddr = 1;   /* Subaddress   */
message.mess_command1.tran_rec = 0;  /* Receive Data  */
message.mess_command1.rtaddr   = 1;  /* Terminal address */

message.errorid      = 0;
message.gap_time     = 4;
message.data[0][0]   = 0x0100;
message.data[0][1]   = 0x0200;
message.data[0][2]   = 0x0300;
message.data[0][3]   = 0x0400;
message.data[0][4]   = 0x0500;
message.data[0][5]   = 0x0600;

printf("Calling the BusTools API One Shot Execute on Bus B...\n");
for ( i = 0; i < COUNT; i++ )
if ( (nStatus = BusTools_BC_OneShotExecute(0, 1, &message, 1))
        != API_SUCCESS )
{
  printf("Error from BC_OneShotExecute() = %4.4X\n", nStatus);
  printf(BusTools_StatusGetString(nStatus));
  printf("\n");
  break;
}

// Setup the message for the One Shot Execute mode, Channel A.
message.control = BC_CONTROL_MESSAGE |
        BC_CONTROL_BUFFERA |/* Use first buffer, e.g., data[0] */
        BC_CONTROL_CHANNELA; /* Transmit on Channel A.   */

printf("Calling the BusTools API One Shot Execute on Bus A...\n");
for ( i = 0; i < COUNT; i++ )
if ( (nStatus = BusTools_BC_OneShotExecute(0, 1, &message, 1))
        != API_SUCCESS )
{
  printf("Error from BC_OneShotExecute() = %4.4X\n", nStatus);
  printf(BusTools_StatusGetString(nStatus));
  printf("\n");
  break;
}

if ( (nStatus = BusTools_API_Close(0)) != API_SUCCESS )
```

```
  {
    printf("Error from API_Close() = %4.4X\n", nStatus);
    printf(BusTools_StatusGetString(nStatus));
    printf("\n");
  }
  printf("All Done...\n");

#ifdef _Windows
  printf("Alt-F4 to close this window...");
#endif
  exit(0);
}
```

CHAPTER 8

BusTools Memory

The BusTools microprocessor is controlled by information and data structures stored in the memory on the board.

This chapter gives an overview of the memory structure as defined by the hardware and how the API routines create and align various memory structures.

Hardware Operation

There is 256 Kbytes of memory on the PC-1553/1773 BusTools board organized as 128K 16-bit words. The memory is split into two segments of 64KW each, with the segments named **SEG1** and **SEG2**. The first 128 words of SEG1 are reserved as hardware control registers. The remainder of SEG1 and all of SEG2 are used to store various data buffers and control structures used for the 1553 simulation.

There is 128 Kbytes of memory on the IP-1553 BusTools board organized as 64K 16-bit words. In contrast to the PC-1553, the memory is viewed as one single 64KW segment. There are, therefore, no issues with regards to memory segmentation for this product and in the following discussions of data structure allocation, the **SEG1** and **SEG2** differentiation will not apply to this product since all data structures are in the only segment.

There are five groups of memory structures that are stored in the BusTools memory:

- **BM Operation:** The various memory structures used to control the operation of the Bus Monitor. These structures can be located in either segment (though all structures must be in the same segment).
- **RT Simulation:** The various memory structures used to control the operation of one or more Remote Terminal simulations. These structures can be located in either segment (though all structures must be in the same segment).
- **BC Simulation:** The various memory structures used to control the Bus Controller simulation. These structures can be located in

either segment (though all structures must be in the same segment).

- **Error Injection Buffers:** There is a special type of buffer used to control the injection of error conditions into the 1553 data stream. If the BC or any RT's are defined, there must be at least one error injection buffer defined. Depending on the application requirements, there can be many. All error injection buffers must be located in SEG1.
- **Interrupt Queue:** The Interrupt Queue is used by the BusTools microcode to notify the host processor that a significant event has occurred. The Interrupt Queue must be located in SEG1.

Within any control structure, any memory addresses are specified as a single 16-bit word. This means that addresses can only refer to locations within the segment. The only exception is for addresses supplied in the hardware control registers. In this case, the segment being addressed is determined by the settings the segment assignment bits. In addition, any address specifying an Error Injection Buffer is automatically assumed to be in SEG1.

It is up to the application to assign the various simulation control structures to the appropriate memory segment. The details of the contents of the various memory resident structures is beyond the scope of this document, but that information can be found in the PC-1553/1773 or IP-1553 Hardware Reference Manuals.

Software Operation

The BusTools API routines provide a framework in which you need not be concerned with memory organization or how the various structures are allocated. However, some knowledge of the framework used by the API routines may be helpful. Therefore, this section describes the various assumptions made by the API routines.

Since the memory of the PC-1553/1773 is organized into two segments, the API routines assign all BM and BC functions to SEG1. All RT control structures are assigned to SEG2.

The IP-1553 supports but a single segment; therefor all of the structures are allocated within that segment when the API detects that the IP module is being used.

PC-1553/1773 SEG1 Memory Organization

The BusTools API routines assign memory in SEG1 at the following word offsets:

- **(0x0000 - 0x007F) Hardware Control Registers:** These locations are assigned to the hardware and microcode control registers. They include specifications of where the various simulation control structures are located and the segment assignment bits for each function.
- **(0x0080 - 0x0094) BM Trigger Buffer:** This memory block is reserved for the BM Trigger Buffer. Even if no BM triggers have been defined, this block must be allocated and set to the default values of all zeros (which is automatically done when the **BusTools_BM_Init()** routine is called).
- **(0x0095 - 0x0098) Default BM Control Buffer:** This memory block is reserved for the default BM Control Buffer. When the BM is initialized, all addresses in the BM Filter Buffer point to this location. The default Control Buffer enables the recording of all messages to or from the specified RT address / sub-address.
- **(0x0099 - 0x0398) Interrupt Queue:** This memory block contains the Interrupt Queue. The BusTools API operates with 256 blocks in the Interrupt Queue.
- **(0x0399 - 0x07FA) Error Injection Buffers:** This memory block contains the Error Injection Buffers. The BusTools API operates with 33 Error Injection Buffers.
- **(0x07FB - 0x07FF) Unused.**
- **(0x0800 - 0x0FFF) BM Filter Buffer:** This memory block contains the BM Filter Buffer. There is one word in this block for each possible RT address (32), Transmit/Receive(1) and RT subaddress (32). This block is always defined by the BusTools API routines (though, by default, all addresses point to the default BM Control Buffer). The block must begin on an even 2KW boundary.
- **(0x1000 - 0xFFFF) Available.** This area is available for BM and BC control structures and data buffers (60 KWords). Assuming the use of 250 Bus Monitor buffers, there is room for over 880 single-buffered Bus Controller messages.

PC-1553/1773 SEG2 Memory Organization

The BusTools API routines assign memory in SEG2 at the following word offsets:

- **(0x0000 - 0xF1BF) Available.** This area is available for user defined RT Message Buffers and Broadcast RT Control Buffers. If Broadcast is enabled, 3906 or 4032 words will be reserved for the Broadcast RT Control Buffers, depending on the state of the Subaddress 31 Mode Code Enable switch. This allows the creation of over 1000 message buffers.
- **(0xF160 - 0xF1BF) Default RT Control Buffers:** There is a default RT Control Buffer for each RT address for a total of 96 words.
- **(0xF1C0 - 0xF77F) Default RT Message Buffers:** There is one default message buffer assigned to each RT address. These buffers are located here (32 RT's with 43 words per Message Buffer, for a total of 1376 words).
- **(0xF780 - 0xF7FF) RT Address Buffer:** This memory block contains the RT Address Buffer, which must be located on an even double word boundary. There are four words in this block for each possible RT address, for a total of 128 words.
- **(0xF800 - 0xFFFF) RT Filter Buffer:** This memory block contains the RT Filter Buffer. There is one word in this block for each possible RT address (32), Transmit/Receive(2) and RT subaddress (32), for a total of 2048 words. These words point to the associated RT Control Buffer. This block is always initialized to the default values by the **BusTools_Init()** routine. This buffer must begin on an even 2KW boundary within SEG2.

IP-1553 Memory Organization

The BusTools API routines assign IP-1553 memory at the following word offsets:

- **(0x0000 - 0x007F) Hardware Control Registers:** These locations are assigned to the hardware and microcode control registers. They include specifications of where the various simulation control structures are located and the operational control bits.
- **(0x0080 - 0x0094) BM Trigger Buffer:** This memory block is reserved for the BM Trigger Buffer. Even if no BM triggers have been defined, this block must be allocated and set to the default values of all zeros (which is automatically done when the **BusTools_BM_Init()** routine is called).
- **(0x0094 - 0x0098) Default BM Control Buffer:** This memory block is reserved for the default BM Control Buffer. When the

BM is initialized, all addresses in the BM Filter Buffer point to this location. The default Control Buffer enables the recording of all messages to or from the specified RT address / sub-address.

- **(0x0099 - 0x0398) Interrupt Queue:** This memory block contains the Interrupt Queue. The BusTools API operates with 256 blocks in the Interrupt Queue.
- **(0x0399 - 0x07FA) Error Injection Buffers:** This memory block contains the Error Injection Buffers. The BusTools API operates with 33 Error Injection Buffers.
- **(0x07FB - 0x07FF) Unused.**
- **(0x0800 - 0x0FFF) BM Filter Buffer:** This memory block contains the BM Filter Buffer. There is one word in this block for each possible RT address (32), Transmit/Receive(1) and RT subaddress (32). This block is always defined by the BusTools API routines (though, by default, all addresses point to the default BM Control Buffer). The block must begin on an even 2KWord boundary.
- **(0x1000 - 0xF15F) Available.** This area is available for BM and BC control structures and data buffers, and for RT broadcast control buffers, RT message buffers and RT broadcast control buffers (56 KWords). If Broadcast is enabled, 3906 or 4032 words will be reserved for the Broadcast RT Control Buffers, depending on the state of the Subaddress 31 Mode Code Enable switch.
- **(0xF160 - 0xF1BF) Default RT Control Buffers:** There is a default RT Control Buffer for each RT address for a total of 96 words.
- **(0xF1C0 - 0xF77F) Default RT Message Buffers:** There is one default message buffer assigned to each RT address. These buffers are located here (32 RT's with 43 words per Message Buffer, for a total of 1376 words).
- **(0xF780 - 0xF7FF) RT Address Buffer:** This memory block contains the RT Address Buffer, which must be located on an even double word boundary. There are four words in this block for each possible RT address, for a total of 128 words.
- **(0xF800 - 0xFFFF) RT Filter Buffer:** This memory block contains the RT Filter Buffer. There is one word in this block for each possible RT address (32), Transmit/Receive(2) and RT subaddress (32), for a total of 2048 words. These words point to the associated RT Control Buffer. This block is always initialized to the default values by the **BusTools_Init()** routine. This buffer must begin on an even 2KWord boundary.

CHAPTER 9

Error Injection

The Error Injection (EI) function on the BusTools board provides the ability to create specific types of errors in the 1553 data stream to test how other equipment responds to these error conditions. This chapter describes the Error Injection functionality and the BusTools API routines used to access the Error Injection functions.

Error Injection Hardware Operation

There are several kinds of errors that can be injected into the 1553 data stream. Some errors effect individual words within the message. Others effect the entire message. Some cause the RT to perform the wrong action. In all cases, the errors are specified in an Error Injection Buffer that is associated with a particular operation. All Error Injection Buffers must be located in the first memory segment (SEG1 - the lower 64 KWords).

Typically, one Error Injection Buffer is defined with no errors. This buffer is used by all BC and RT message buffers for which there are to be no errors (this is API error injection buffer "0").

Each word of an Error Injection Buffer controls the error(s) to be injected for a particular word being transmitted on the 1553 bus. For each 1553 message, there are up to three Error Injection Buffers that can be involved. If the BusTools board is simulating the BC, there is an Error Injection Buffer associated with the BC Message Buffer (see API_BC_MBUF in Appendix B). If the BusTools board is simulating the specified RT, there is an Error Injection Buffer associated with the RT Message Buffer (see API_RT_MBUF_WRITE in Appendix B). Finally, if the message is a RT-to-RT message, there could be two RT's being simulated by the BusTools board, each with its own Error Injection Buffer.

While the maximum size of an Error Injection Buffer is 34 words, the actual size is a function of the Error Injection Buffer Type. There are five types of Error Injection Buffers:

- **BC Receive:** This Error Injection buffer type is used to specify the errors to be injected by the BC Simulation into a Receive Message. This buffer contains 34 error specification words. There is one error specification word for the message command word followed by 33 error specification words for message data words (33 words are reserved in case a Word Count Error is specified)..
- **BC Transmit:** This Error Injection buffer type is used to specify the errors to be injected by the BC Simulation into a Transmit Message. This buffer contains only one error specification word that is applied to the message command word.
- **BC RT-to-RT:** This Error Injection buffer type is used to specify the errors to be injected by the BC Simulation into a RT-to-RT Message. This buffer contains two error specification words, one for each message command word to be generated.
- **RT Receive:** This Error Injection buffer type is used to specify the errors to be injected by the RT Simulation into a Receive Message. This buffer contains one error specification word that is applied to the message status word.
- **RT Transmit:** This Error Injection buffer type is used to specify the errors to be injected by the RT Simulation into a Transmit Message. This buffer contains up to 34 error specification words. There is one error specification word for the message status word followed by up to 33 error specification words for message data words (33 words are reserved in case a Word Count Error is specified).

Error Types

The possible errors that can be injected into the 1553 data stream are described here. As words (command, status, or data) are transmitted on the bus, the BusTools board checks the current position in the associated Error Injection Buffer to see if any error conditions have been specified for that particular word. If present, the error condition is added to the word as it is transmitted.

The following list describes each of the possible error conditions. These errors can be injected into any word of a 1553 message (command word, status word, and / or data words). In some cases, there may be limitations as to the practical uses of the error condition.

- **Bit Count Error:** The hardware normally transmits a sync pulse followed by 16 data bits and ending with a parity bit for each word sent out on the 1553 bus. When this error is selected, the actual number of data bits transmitted in the word can be specified. 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 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.

- **Inverted Sync Error:** This error is used to transmit a word with an inverted sync bit. This error can be specified for any entry in any EI buffer type.
- **Parity Error:** This error is used to transmit a word with an inverted parity bit. This error can be specified for any entry in any EI buffer type.
- **Respond with Wrong Address:** Normally, the RT status word contains the address of the RT. This error forces the status word to contain a different address. The address can be any number from 0 to 31. This error can be specified only for "RT Receive" and "RT Transmit" EI buffer types.
- **Response on Both Buses:** This error forces an RT to transmit a word (either status or data) on both channels, not just on the channel on which the command word was received. The hardware actually allows this error to be specified and injected on a word-by-word basis. However, this is not the intended use of this error condition. This error is intended to be a message-wide condition. When the BC sends a command word, the RT responds with its entire response on both channels. Therefore, you should ensure that if this error is desired, the error bit is set for every word of the Error Injection buffer. This error can be specified only for "RT Receive" and "RT Transmit" EI buffer types. This error has no meaning when set in any BC buffer type.
- **Response on Wrong Bus:** This error forces a RT to transmit a word (either status or data) on the opposite channel from which the command word was received. The hardware actually allows this error to be specified and injected on a word-by-word basis. However, this is not the intended use of this error condition. This error is intended to be a message-wide condition. When the BC sends a command word on one channel, the RT responds with its entire response on the opposite channel. Therefore, you should ensure that if this error is desired, the error bit is set for every word of the Error Injection buffer. This error can be specified only for "RT Receive" and "RT Transmit" EI buffer types. This error has no meaning when set in any BC buffer type.
- **Word Count Error:** Normally, the command word specifies the number of words that is contained in a message. This error can force the actual number of words transmitted in the message to be different from that specified in the command word. The hardware accepts any word count from 1 to 33 for this error. This error condition must be specified on the command word of a BC message (for "BC Receive" buffer types) or the status word of a RT message (for "RT Transmit buffer types).

Error Injection Software Operation

The BusTools API routine which handle Error Injection Buffers are described here. Detailed calling sequences and error status information can be found in a separate chapter of this manual.

When the Bus Monitor operation is initialized (using the **BusTools_BM_Init()** routine), BusTools memory is initialized with 33 Error Injection Buffers. A number between 0 to 32 identifies the buffers. On initialization, all buffers are filled with "zeros", indicating that there are no errors to be injected for any Error Injection Buffer type.

The first buffer (buffer id #0) should be left in its initialized state. It is reserved as the "No Error" buffer. All BC Message Buffers and RT Message Buffers which do not want any error injection should reference it.

The remaining buffers must be filled in, as required, with error injection information. This is done by using the **BusTools_EI_EbufWrite()** routine. The caller must specify which Error Injection Buffer to be accessed and a structure which indicates the buffer type and buffer contents. The elements of the error injection definition structure (API_EIBUF) are defined in Appendix B. It is up to the caller to ensure that each element of the structure matches the requirements of the Error Injection Buffer Type.

CHAPTER 10

BusTools API Routines

This chapter contains a detailed description of each routine in the BusTools API. For each routine, the function of the routine is described, the calling sequence for the routine is defined, and a list of possible return status codes is presented. The status codes are listed mnemonically. Descriptions of each status code is provided in Appendix A.

There are three files required when developing and running 16-bit Windows® applications that use the BusTools API:

- **busapi.h:** This is the header file associated with the BusTools API. It contains all structure definitions, sub-routine prototypes and parameter definitions required to use the BusTools API. It must be included in any source file which accessed the BusTools API routines.
- **busapi.lib:** This is the library file which must be linked with the application program. It contains the references to the API routines contained in the DLL file.
- **busapi.dll:** This is the DLL file which contains the executable version of the API routines. This file must be present and is loaded when running an application which uses the BusTools API.

There are three files required when developing and running 32-bit Windows® applications (e.g., Windows 95 and Windows NT) which use the BusTools API:

- **busapi.h:** This is the header file associated with the BusTools API. It contains all structure definitions, sub-routine prototypes and parameter definitions required to use the BusTools API. It must be included in any source file which accessed the BusTools API routines.
- **busapi32.lib:** This is the library file which must be linked with the application program. It contains the references to the API routines contained in the DLL file. Note that there are two libraries supplied; one for use with the Microsoft compilers and the other for use with the Borland compilers.

- **busapi32.dll:** This is the DLL file which contains the executable version of the API routines. This file must be present and is loaded when running an application which uses the BusTools API.

There are four files required when developing and running 16-bit MS-DOS™ applications which use the BusTools API:

- **busapi.h:** This is the header file associated with the BusTools API. It contains all structure definitions, sub-routine prototypes and parameter definitions required to use the BusTools API. It must be included in any source file which accessed the BusTools API routines. The label "_DOS" should be defined when compiling a source file for use in DOS.
- **busapidb.lib:** This is the DOS library file which must be linked with the application program if one is using the Borland tools. It contains the API routines in a format compatible with Borland "C/C++" versions 3.1 and above.
- **busapidm.lib:** This is the DOS library file which must be linked with the application program if one is using the Microsoft tools. It contains the API routines in a format compatible with Microsoft "C/C++" version 7 or Microsoft Visual C/C++ version 1.0 - 1.5x.

Programming Notes

- Be sure to set your compiler to generate "1 byte alignment" of structures. The normal default is typically 2- or 4-byte alignment, which will *not* correctly interface with the *BusTools* library.
- If you have a single function interface (PC-1553-S or IP-1553-S) you can only start ONE of the functions (BC, BM or RT). Any attempt to start multiple functions will cause a failure.
- Be sure to use the latest version of BusAPI.H when re-compiling your program; various structures and calling sequences may have been changed. Using the latest version will give the compiler a chance to flag any changes which might affect your program.
- Be sure that the symbol "_DOS" is defined when compiling a DOS-based program; if compiling for Windows® (GUI or console mode, if 32-bits) make sure it is **not** defined.
- 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 file.
- If you are using a compiler other than Borland "C/C++" Version 3.1 or above, or Microsoft "C" Version 7.00 or above, or Microsoft Visual "C/C++" Version 1.0 or above, you may have to change the BusAPI.H file. Consult your compiler's documentation for further information.

BusTools_API_Close

Description

This routine is used to terminate BusTools API operations. All resources allocated by the API routines are released. The interrupt service and timer functions are terminated.

This function must be called before exiting a program which has accessed the BusTools API routines. Failure to call this routine may have unpredictable (and generally catastrophic) effects on the DOS or Windows® environment.

Under Windows® 3.x, 95 and NT the API attempts to detect that the user program has terminated and call this function to clean up the environment, but that mechanism is not fool proof, and should not be relied upon.

Multiple calls to this function are harmless.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_API_Close(wCardnum);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_API_Init

Description

This routine is used to initialize the BusTools API routines. This routine must be called before any other BusTools API function can be accessed.

This function is provided for compatibility with existing applications only, new applications should use **BusTools_API_InitExtended().**

This function is typically used to initialize the PC-1553 hardware on the PC platform, to initialize the PC-1553-IP on either the VME or PCI carriers, see the **BusTools_API_InitExtended()** function.

Note that, in order to reduce the memory consumed by the driver, the MS-DOS® version of the driver only supports 2 boards (0 and 1) while all of the Windows® versions support 4 boards (0 through 3). Should support for additional boards be required from within a single application, please contact Condor Engineering for modified drivers and assistance in mapping the boards into the available memory space. For information concerning the exact memory requirements of the current driver, please contact Condor Engineering.

The initialization process includes the following: the appropriate "DLL" files are loaded, the resources required by the API are allocated, interrupt service processing is initialized, the BusTools board is reset, all memory on the BusTools board is cleared, and microcode registers are initialized. In the case of the IP module the Writable Control Store(WCS) is initialized with the current microcode.

The caller must supply the base address of the board (as selected by the hardware jumpers). The 32-bit address specified is the base of a 32-KB window into the 256-KB resident in the BusTools hardware in the case of the PC-1553/1773 board. The 32-bit address specified is the base of a 16-KB window into the carrier board in the case of one or more IP modules on an ISA bus carrier.

The caller must supply the IO base address of the PC-1553/1773 board (as selected by the hardware jumpers). This address can be 0x200, 0x220, 0x300 or 0x320, as supported by the hardware. In the case of an IP module, this parameter specifies the type of carrier and which slots the IP module occupies. Specify the following constants, "ORed" together as necessary:

- IP1553_ISA - Indicates IP-1553 on ISA Bus IP Carrier
- SLOT_A - Indicates first and second slots are occupied by IP
- SLOT_B - Indicates second and third slots are occupied by IP
- SLOT_C - Indicates third and fourth slots are occupied by IP

To initialize a PC-1553-IP module on either the PCI carrier or the VME carrier, see the **BusTools_API_InitExtended**() function.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_API_Init(Cardnum,dwAddress,wIOaddress, pwFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3, 0 - 1 if MS-DOS).
dwAddress	(BT_U32BIT) base memory address of board (the address selected by the jumper on the board). The default factory setting for this jumper is 0xD0000 for a PC-1553/1773 card or an ISA bus IP carrier. Note that this address is a linear address, not a "segment:offset" address!
wIOaddress	(BT_UINT) base I/O address of the board (selected by jumpers on the board). The default factory setting is 0x200 for the PC-1553/1773 card.
pwFlag	(BT_UINT*) pointer to a flag used to specify the software-only version (0) or hardware version (1) of the driver. Should be set to one (1) for normal operation. It always returns a "1".

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_INITED;
BTD_ERR_BADDETECT:

BusTools_API_InitExtended

Description

This routine is used to initialize the BusTools API routines. This routine must be called before any other BusTools API function can be accessed.

This is an extended version of **BusTools_API_Init()**, a full-featured initialization function, which is capable of initializing any of the supported combinations of BusTools hardware on any supported platform. The API supports all combinations of PC-1553/PC-1773 boards and PC-1553-IP modules (both single- and multi-function) on all supported carriers in a single PC. This function will be used to support future new products.

Note that, in order to reduce the memory consumed by the driver, the MS-DOS® version of the driver only supports 2 boards (0 and 1) while all of the Windows® versions support 4 boards. Should support for additional boards be required from within a single application, please contact Condor Engineering for modified drivers and assistance in mapping the boards into the available memory space. For information concerning the exact memory requirements of the current driver, please contact Condor Engineering.

The initialization process includes the following: the appropriate "DLL" files are loaded, the resources required by the API are allocated, interrupt service processing is initialized, the BusTools board is reset, all memory on the BusTools board is cleared, and microcode registers are initialized. In the case of the PC-1553-IP module the Writable Control Store(WCS) is initialized with the current microcode.

The caller must supply the base address of the board (as selected by the hardware jumpers). The 32-bit address specified is the base of a 32-KB window into the 256-KB resident in the BusTools hardware in the case of the PC-1553/PC-1773 board. The 32-bit address specified is the base of a 16-KB window into the carrier board in the case of one or more IP-1553 modules on an ISA bus carrier, and is not used for either the PCI carrier or the 32-bit Windows® NT or 95 operating systems.

The caller must supply the IO base address of the PC-1553/PC-1773 board (as selected by the hardware jumpers). This address can be 0x200, 0x220, 0x300 or 0x320, as supported by the hardware. In the case of an IP-1553 module, this parameter specifies the base address of the IO port, which is the base address of the IO space of the carrier. This parameter is not used for either the PCI carrier or the 32-bit Windows® NT or 95 operating systems. See the description below for each supported carrier.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_API_InitExtended(Cardnum,dwAddress,wIOaddress, pwFlag, platform, cardType, carrier, slot, mapping);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0-3, 0-1 if DOS).
dwAddress	(BT_U32BIT) base memory address of the board (the address selected by the jumper on the board). The default factory setting for this jumper is 0xD0000 for a PC-1553/1773 card or an ISA bus IP carrier. Note that this address is a linear address, not a "segment:offset" address!
wIOaddress	(BT_UINT) base I/O address of the board (selected by jumpers on the board). The default factory setting is 0x200 for the PC-1553/1773 card.
pwFlag	(BT_UINT*) pointer to a flag used to specify the software-only version (0) or hardware version (1) of the driver. Should be set to one (1) for normal operation. It always returns a "1".
platform	(BT_UINT) execution platform: PLATFORM_PC or PLATFORM_VMIC.
cardType	PC1553, IP1553 or PCI1553
carrier	NATIVE, IP1553_ISA, IP1553_PCI, VME_VMIC
slot	(BT_UINT) SLOT_A, SLOT_B, SLOT_C, SLOT_D
mapping	(BT_UINT) carrier memory map: CARRIER_MAP_DEFAULT

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_INITED;
BTD_ERR_BADDETECT:

Notes

This initialization function provides the ability to initialize the various combinations of PC-1553/PC-1773 boards and PC-1553-IP modules on the various supported carriers, under the various operating systems. Since the API does not attempt to automatically detect the board type, it relies on the user-supplied values in this call to specify which board type is being used.

Some of the parameters passed to this function are ignored, depending on the programming environment and/or the board type. For example, in 32-bit Windows® the board address is ignored; the actual address is fetched from the registry. PCI boards and carriers are programmed by the Operating System (OS), which determines their starting address. The API queries the OS for this information; the information specified by the user in this call is ignored.

The parameters should be set as follows when opening the given combinations of hardware and operating systems:

PC-1553/PC-1773 board:

dwAddress	should be set to the base address of the board, as selected by the board jumpers. Ignored under 32-bit Windows®, the actual address is obtained from the Registry based on wCardnum.
wIOaddress	should be set to the IO address of the board, as selected by the board jumpers. Ignored under 32-bit Windows®, the actual address is obtained from the Registry based on the wCardnum.
platform	should be set to PLATFORM_PC.
cardType	should be set to PC1553
carrier	should be set to NATIVE
slot	should be set to NATIVE
mapping	should be set to CARRIER_MAP_DEFAULT

PC-1553-IP; ISA carrier with IP1553 modules:

dwAddress	should be set to the base address of the carrier, as selected by the carrier jumpers. Ignored under 32-bit Windows®, since the actual address is obtained from the Registry based on wCardnum.
wIOaddress	ignored for the ISA carrier on all platforms.
platform	should be set to PLATFORM_PC.
cardType	IP1553, IP1553MF or IP1553SF
carrier	should be set to IP1553_ISA.
slot	should be set to SLOT_A, SLOT_B, or SLOT_C, depending on which slot on the carrier the IP is installed in.
mapping	should be set to CARRIER_MAP_DEFAULT

PCI-1553-IP; PCI carrier with IP1553 modules:

dwAddress	Ignored under 32-bit Windows®, the actual address is obtained from the Registry based on wCardnum. Under 16-bit Windows®, this is the number of the PCI carrier (zero for the first carrier, 1 for the second, etc.), the actual address is obtained from the BIOS. The PCI carrier is not supported under DOS..
wIOaddress	ignored for the PCI carrier on all platforms.
platform	should be set to PLATFORM_PC.
cardType	IP1553, IP1553MF or IP1553SF
carrier	should be set to IP1553_PCI.
slot	should be set to SLOT_A or SLOT_C, depending on which slot in the carrier the IP is installed in.
mapping	should be set to CARRIER_MAP_DEFAULT

VME Platforms: VME-1553-IP with driver for the VME Microsystems International Corporation (VMIC) Pentium®-Based VME controllers, using the VMIC VME Manager:

dwAddress	should be set to the base address of the carrier board, as selected by the board jumpers, in A32 address space. A16 and A24 space mapping are not supported.
wIOaddress	should be set to the IO address of the board, as selected by the board jumpers, in A16 space. A24 and A32 space mapping are not supported.
platform	should be set to PLATFORM_VMIC.
cardType	IP1553, IP1553MF or IP1553SF
carrier	should be set to IP1553_VME.
slot	should be set to SLOT_A or SLOT_C, depending on which slot in the carrier the IP is installed in.
mapping	should be set to CARRIER_MAP_DEFAULT.

VME Platforms: VME-1553-IP with VxWorks driver:

dwAddress	should be set to the base address of the carrier board, as selected by the board jumpers, in A24 address space. A16 and A32 space mapping are not supported.
wIOaddress	should be set to the IO address of the board, as selected by the board jumpers, in A16 space. A24 and A32 space mapping are not supported.
platform	should be set to PLATFORM_VME.
cardType	IP1553, IP1553MF or IP1553SF
carrier	should be set to IP1553_VME.
slot	should be set to SLOT_A or SLOT_C, depending on which slot in the carrier the IP is installed in.
mapping	should be set to CARRIER_MAP_DEFAULT.

BusTools_BC_AperiodicRun

Description

IP-1553-xx modules and the PCI-1553 support aperiodic messages. The PC-1553/1773 boards do not support aperiodic messages. Aperiodic messages are messages that are inserted into the running Bus Controller bus list on a one-shot basis. The execution of these aperiodic messages is initiated by this function..

This function can be called either before or after the bus list is started. If this function is called before the bus list has been started, the aperiodic messages specified will be executed as soon as the bus list is started by a call to **BusTools_BC_Start()** or **BusTools_BC_StartStop()**.

The BC supports two types of aperiodic messages: High Priority and Low Priority.

High Priority aperiodic messages are executed by the BC following the completion of the current message. The specified list of messages is executed, in sequence, until complete. A next message pointer of zero terminates the specified list of aperiodic messages. When the BC encounters this condition it returns to processing the regular message list. If the execution of the next message in the high priority list would cause the minor frame to overflow, the message is not transmitted, the frame over run bit is NOT set, and the BC waits for the beginning of the next minor frame to transmit the message.

Low Priority aperiodic messages are executed by the BC at the end of the minor frame. Before transmitting a message from the aperiodic message list, the BC determines if the message would cause the minor frame to overflow. If not, the message is executed, otherwise the message is not executed, and the BC suspends processing low priority aperiodic messages until the end of the following minor frame, when the process is repeated. The time computed by the microcode uses the 1553 command word and the rt31_bcst and sa31_mc bits (see **BusTools_SetBroadcast()** and **BusTools_SetSa31()**) to determine approximately how much time in microseconds (µs) is needed as follows:

Message Timing Computations

For normal messages the time computed is 125 µs + 25 µs/data word.
For mode codes the time computed is either 125 µs or 150 µs (without/with data word).
For broadcast messages time is computed as 300 µs + 25 µs/data word.

The firmware does not allow for error injection when computing the message time, so it is still possible to overflow the minor frame if high word error injection or programmable response time are used on aperiodic messages (or regular messages) that push the limit. That is, the following are NOT added to the computed time requirement:

Message Timing Computations--Ignored Factors

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

For this reason, care should be taken when setting up the bus list and the aperiodic message list. Iinsure that either enough time is available in the minor frames, or that if a frame does not have "plenty" of spare time, that the last messages in the frame (or the aperiodic messages) do not use the ignored factors from the above list. If this rule is not followed, the minor frame timing may not be maintained.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using both the **BusTools_BC_Init()** and **BusTools_BC_MessageAlloc()** routines). Additionally, the **BusTools_BC_MessageWrite()** function must be called for each message in the *wMessageid* list to initialize the messages.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_AperiodicRun( wCardnum, wMessageid,
                    wHipriority, wWaitFlag, wWaitTime );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wMessageid	(BT_UINT) BC message number ("0" based).
wHipriority	(BT_UINT) 1 -> Hi Priority, 0 -> Low Priority msgs
wWaitFlag	(BT_UINT) 1 -> Wait for BC to complete executing aperiodic messages, 0 -> No wait-return from function immediately after starting aperiodic processing. See the function **BusTools_BC_AperiodicTest()** to determine if list has been completed.
wWaitTime	(BT_UINT) Timeout in seconds (16-bits) or milliseconds (32-bits). Ignored if no wait was specified.

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_HARDWARE_NOSUPPORT
API_BC_ILLEGAL_MBLOCK
API_BC_APERIODIC_RUNNING
API_BC_APERIODIC_TIMEOUT

Notes:

1. See the function BusTools_BC_AperiodicTest() to determine when the aperiodic messages have been completed, if no wait was specified.
2. The time out value specified is assumed to be in milliseconds if the application is a 32-bit application (Window NT or 95), and in seconds if the application is compiled with the DOS or Windows 3.x versions of the API.
3. The High Priority and Low Priority message lists are independent, either or both can be active at the same time.
4. The minor frame counter counts in 25 microsecond ticks.
5. The wMessageid passed to this function is assumed to be the first message in a list of messages. Any legal message, conditional, Stop BC or Noop can be contained in this list, which can be any length. The only restriction is that the last logical message in the list must have a next message pointer value of zero.
6. See the function definitions for **BusTools_BC_MessageAlloc()** and **BusTools_BC_MessageWrite()** in this section, and the BC Message buffer definition (API_BC_MBUF) in Appendix B for additional information.

BusTools_BC_AperiodicTest

Description

IP-1553-xx modules and the PCI-1553 support aperiodic messages. The PC-1553/1773 boards do not support aperiodic messages. Aperiodic messages are messages that are inserted into the running Bus Controller bus list on a one-shot basis.

BusTools_BC_AperiodicRun() initiates transmission of aperiodic messages(). **BusTools_BC_AperiodicTest()** may be used to determine if a previously initiated list of aperiodic messages has been completed, and if the hardware is ready to accept another set of aperiodic messages.

This function is especially useful when using a long list of low priority aperiodic messages, since such a message list might take several minor frames to be executed by the BC.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BC_AperiodicRun(wCardnum, wHipriority);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wHipriority	(BT_UINT) 1 -> Hi Priority msgs, 0 -> Low Priority msgs

Return Value

API_SUCCESS
API_BUSTOOLS_BADCARDNUM
API_BUSTOOLS_NOTINITED
API_BC_NOTINITED
API_HARDWARE_NOSUPPORT
API_BC_APERIODIC_RUNNING

Notes:

This function polls the hardware register which controls the execution of aperiodic messages, and returns API_BC_APERIODIC_RUNNING if the specified list is still executing, API_SUCCESS if the aperiodic processing is complete, or one of the other error conditions.

BusTools_BC_Init

Description

This routine is used to initialize BC operations in the BusTools API. It must be called prior to calling any other routine which accesses any BC function or structure. While it can be called multiple times to change any of the BC setup parameters, it cannot be called while the BC is running. The caller specifies the following parameters:

- **Channel:** The "wChannel" parameter specifies on which channel the BC activity is output (either channel A or B) for microcode version 1.03 and below. Not used by currently shipping microcode versions.
- **Interrupt Enable Bits:** The "dwIntEnable" parameter specifies the conditions which cause a BC interrupt to occur. When an enabled interrupt condition occurs, the BusTools API software sends the specified message to the specified window (as defined in the **BusTools_InterruptRegister()** routine). The interrupt condition definitions are provided in Appendix B.
- **Retry Enable Bits:** The "wRetry" parameter specifies the conditions under which the BC retries a message upon detection of an error. The retry condition definitions are provided in Appendix B, See the section "BC Retry Parameters (BusTools_BC_Init() argument)".
- **No Response Timeout:** The "wTimeout1" parameter specifies the "No Response" time-out period (in microseconds). If the RT specified in any BC command does not respond to the command within this time-out period, a "No Response" error is logged by the BC (and an interrupt is generated if that interrupt condition is enabled). The specified value must be between 4 and 32 microseconds. The normal value for MIL-STD-1553B is 14 microseconds.
- **Late Response Timeout:** The "wTimeout2" parameter specifies the "Late Response" time-out period (in microseconds). If the RT specified in any BC command does not respond to the command within this time-out period, a "Late Response" error is logged by the BC (and an interrupt is generated if that interrupt condition is enabled). The "Late Response" time-out period must be less than the "No Response" time-out period. The specified value must be between 4 and 32 microseconds. The normal value for MIL-STD-1553B is 12 microseconds
- **Minor Frame Period:** The "dFrame" parameter specifies the minor frame period in microseconds. Each minor frame in the BC message list is started according to this parameter. Note that a global error occurs if the time required to complete any minor frame is larger than the time specified by the minor frame frequency. The specified value must be between 2000 and 16000000 microseconds (0.65 and 500.0 Hertz) for the PC-1553,

and between 1000 and 16000000 microseconds (0.65 and 1000.0 Hertz) for the IP-1553.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_Init( wCardnum, wChannel, dwIntEnable,
                            wRetry, wTimeout1, wTimeout2, dFrame,
                            num_buffers );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wChannel	(BT_UINT) channel for BC output (microcode 1.03 only, not used for newer versions of microcode, which includes all current shipping versions): 1 -> channel A 0 -> channel B
dwIntEnable	(BT_U32BIT) interrupt enable bits (see Appendix B for bit definitions).
wRetry	(BT_U16BIT) retry enable bits (see Appendix B for bit definitions).
wTimeout1	(BT_UINT) timeout period for the "No Response" error -- must be between 4 and 32 microseconds.
wTimeout2	(BT_UINT) time-out period for the "Late Response" error -- must be between 4 and 32 microseconds.
dFrame	(BT_U32BIT) minor frame period in microseconds -- must be between 2000 and 1600000 microseconds (0.65 and 500.0 Hertz). See note below.
num_buffers	(BT_UINT) number of data buffers associated with each BC message, either 1 or 2. If 1 buffer is created, than both pointers in the BC message structure will refer to the same data buffer. This parameter can be used to reduce the size of BC messages, allowing more Bus Controller messages to be defined. See Notes.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BC_RUNNING:
API_BC_BADTIMEOUT1:
API_BC_BADTIMEOUT2:
API_BC_BADFREQUENCY:

Notes:

1. The "dFrame" parameter was previously a "float" which gave the minor frame frequency in Hertz. DOS versions previous to version

2.22 still require this parameter to be a "float". API versions previous to 2.22 do not support the "num_buffers" parameter.

2. The "num_buffers" parameter can be set to one (1), which will cause the API to generate only one Bus Controller Data Buffer per Message. This will reduce the number of words required by each BC message by 32 words, enabling the definition and use of more BC messages.

BusTools_BC_IsRunning

Description

This routine returns a flag which indicates whether the BC is currently running. If the returned flag is zero, the BC is not running. If the returned flag is non-zero, the BC is running.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BC_IsRunning(wCardnum, pwFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
pwFlag	(BT_UINT*) pointer to returned flag: 0 -> not running 1 -> running

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BC_NOTINITED:



BusTools_BC_MessageAlloc

Description

This routine allocates memory on the BusTools board for the specified number of BC message buffers. The buffers are cleared. It is up to the caller to insert appropriate information in the message buffers using **BusTools_BC_MessageWrite()**.

If the BC is not being used, then call this routine and allocate 2 messages to satisfy the memory allocation requirements of the API.

This routine can be called multiple times. However, the number of message buffers specified in subsequent calls must be equal or less than the value specified when the routine was first called. A larger number of message buffers can only be specified if memory management on the board is re-initialized (by calling **BusTools_BM_Init()** and **BusTools_BC_Init()** again).

Messages allocated by this function are numbered from zero ("0") to wCount-1.

If aperiodic messages are to be used (see **BusTools_BC_AperiodicRun()**) than "extra" messages should be allocated using this function. Create your bus list using **BusTools_BC_MessageWrite()** to initialize and link the first messages into a chain, leaving the unused messages at the end of the list uninitialized. These messages will be used for aperiodic traffic.

When it is necessary to create a list of aperiodic messages, initialize the first of the "extra" messages with the required parameters, point it to the next message, initialize that message, etc., for as many messages as need be transmitted in this aperiodic group. As few as one message, or as many as remain in the message list may be specified in one **BusTools_BC_AperiodicRun**() function call. Insure that the last message to be executed has a next message pointer of zero. Then call the BusTools_BC_AperiodicRun function and specify the index of the first message to be transacted. It will transact messages, beginning with the first specified message, chaining to the next message (using the next message pointer) until it encounters a next message pointer of zero, when it will revert to the normal bus controller message list.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended**() routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine). In addition, the BC must not be running.

The BC message buffers are created allocating either one or two data buffers, depending on the "num_buffers" parameter passed to the **BusTools_BC_Init()** function. One buffer may be specified to reduce the size of a BC message, which will allow more messages to be created.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BC_MessageAlloc(wCardnum, wCount);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wCount	(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:

BusTools_BC_MessageGetaddr

Description

This routine converts a BC message number to a BusTools hardware address. The returned address is a byte offset from the beginning of memory on the BusTools board. The address is in the range 0x00000000 to 0x0003FFFF. This address can be passed to the **BusTools_Memory_Read()** or **BusTools_MemoryWrite()** functions to perform direct access to the BC message structure.

Note that this routine should only be required for debugging operations. Normally, the you don't need to access absolute memory addresses on the BusTools board.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_MessageGetaddr( wCardnum, wMessageid,
                                      pdwAddr );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wMessageid	(BT_UINT) BC message number ("0" based).
pdwAddr	(BT_U32BIT*) pointer to returned address of specified BC message.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BC_NOTINITED:
API_BC_ILLEGAL_MBLOCK:

BusTools_BC_MessageGetid

Description

This routine converts a BusTools hardware address to a BC message number. The specified address is a byte offset from the beginning of memory on the BusTools board. The address must be in the range 0x00000000 to 0x0003FFFF.

Typically, this routine is used by the Windows® task which processes the BC interrupt message. The BC message address supplied as a parameter to this task is converted to a message number and then the message can be retrieved from the board using the **BusTools_BC_MessageRead()** routine.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_MessageGetid(wCardnum,dwAddr,
                                   pwMessageid);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
dwAddr	(BT_U32BIT) memory address (0x00000000 - 0x0003FFFF).
pwMessageid	(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:

BusTools_BC_MessageNoop

Description

This routine is used convert the specified BC message to or from a no operation (Noop) message. This is useful during execution of a bus list when a message must be enabled or disabled for a given period of time, then restored. The contents of the BC message buffer is defined in Appendix B.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** and **BusTools_BC_MessageAlloc()** routines).

Prior to the first time this function is called for a particular message, the specified message must have been setup as a normal 1553 message by a call to **BusTools_BC_MessageWrite()**. This function then can be used to NOOP and restore the message.

For the PC-1553 and IP-1553 (V1.x firmware), note that if the specified message is not a 1553 message or a Noop message created by calling this routine, an error will result.

For the IP-1553 (V2.1 firmware and above) any message, conditional or stop BC can be set to enabled or ignored by this function. Only Noop messages cannot be changed by this function (Noop a Noop message?).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_MessageNoop ( wCardnum, wMessageid,
                                    NoopFlag, WaitFlag );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wMessageid	(BT_UINT) BC message number ("0" based).
NoopFlag	(BT_UINT) Set non-zero to Noop the message, set zero to reactivate the message.
WaitFlag	(BT_UINT) If non-zero the API will test and wait until the Bus Controller is not accessing the message. If zero, the message is updated immediately.

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:

BusTools_BC_MessageRead

Description

This routine is used to read the specified BC message from the BusTools board. The contents of the BC message buffer is defined in Appendix B.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

Note that if the message being read is a RT to RT message, the data might be all zero, since the Bus Controller does not monitor the data transacted on the bus during a RT to RT transfer. Only the IP-1553 using firmware version 2.x or above monitors a RT to RT message's data.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_MessageRead ( wCardnum, wMessageid,
                                    pMessage );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wMessageid	(BT_UINT) BC message number ("0" based).
pMessage	(API_BC_MBUF*) pointer to BC message definition structure (see Appendix B for structure definition).

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:

BusTools_BC_MessageUpdate

Description

This routine allows for the update of BC message data. It is used to update the data of a specific message block. The data area to be updated (there are two for each message block unless **BusTools_BC_Init()** was called with "num_buffers" equal to one) is determined by the current state of the A/B buffer value in the BC Message Block Control Word.

If **BusTools_BC_Init()** was called with "num_buffers" equal to one (1), than this routine will update the contents of the single data buffer associated with the BC message.

This function is only available with microcode revision 1.04 or later. Use the **BusTools_GetRevision()** routine to determine the current microcode revision on your board. Currently shipping hardware is above version 1.04.

The number of words written to the BC message buffer is determined by the word count specified by the first 1553 command word in the message indexed by "wMblock_id".

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_MessageUpdate(wCardnum, wMblock_id,
                                    pBuffer);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_INT) BusTools board number
wMblock_id	(BT_UINT) message block ID ("0" based)
pBuffer	(BT_U16BIT*) pointer to the buffer of length "BT1553_BUFCOUNT" from which the data is transferred

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

B: Al Horn: output new data to the buffer not in use, if swap buffers

BusTools_BC_MessageWrite

Description

This routine is used to write the specified BC message to the BusTools board. The contents of the BC message buffer is defined in Appendix B.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()**or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

Any valid control or 1553 data message may be created using this routine. This includes Mode Codes, RT to BC, BC to RT, RT to RT and Broadcast messages. Additionally, beginning and end of minor frame and end of BC message list may be specified. See Appendix B for the definition of the API_BC_MBUF structure, which is used to describe the message being created.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BC_MessageWrite(wCardnum,wMessageid,pMessage);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wMessageid	(BT_UINT) BC message number ("0" based).
pMessage	(API_BC_MBUF*) pointer to BC message definition structure (see Appendix B for structure definition).

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:

BusTools_BC_OneShotExecute

Description

This routine is used to execute a sequence of 1553 BC messages in the BC-ONLY environment. The message sequence is executed, with status and data information recorded for return to the caller.

The caller specifies the number of messages and the format of each message. The messages can be any 1553 type (BC-to-RT, RT-to-BC, Broadcast, etc.). For "Receive" messages, the caller specifies the data to be sent to the RT. For "Transmit" messages, the data from the message on the bus is collected and returned to the caller in the supplied structures.

> **NOTE:** This routine is used in the BC-ONLY environment. It cannot be used in conjunction with the BM or any simulated RT's. In addition, there can be no interrupt capturing when using this routine. See Chapter 6, "**Bus Controller**", for an example which uses this mode to transmit messages on both Bus A and Bus B.
>
> See Chapter 7 for an example which uses this function.

This function may be called to execute between 1 and 100 messages per call. If the **wWaitFlag** is TRUE, than the function returns after all of the messages have been transacted on the bus, and the **pMessages** structures have been updated by the API. If the **wWaitFlag** is FALSE, the function returns just as soon as the Bus Controller has been started; it is left up to the user's program to determine when the bus list is complete and update the **pMessages** structures using the BusTools_BC_MessageRead() function.

The only routines which can be used with this routine (and which should be called prior to calling this routine, in the following order) are:

- **BusTools_API_Init()** or **BusTools_API_InitExtended():**
- **BusTools_BC_OneShotInit():**
- **BusTools_SetBroadcast():** **(Optional)**
- **BusTools_SetInternalBus():** **(Optional)**
- **BusTools_SetSa31():** **(Optional)**
- **BusTools_SetVoltage():** **(Required)**

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_OneShotExecute( wCardnum, wCount,
                                      pMessages, wWaitFlag );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wCount	(BT_UINT) number of BC messages to be executed, value from 1 to 100.
pMessages	(API_BC_MBUF*) pointer to array of BC messages.

wWaitFlag (BT_UINT) indicates whether routine should return immediately on startup of BC or wait until the messages have been executed:
0 -> return immediately (do not wait).
1 -> wait until messages have been executed.

Return Value

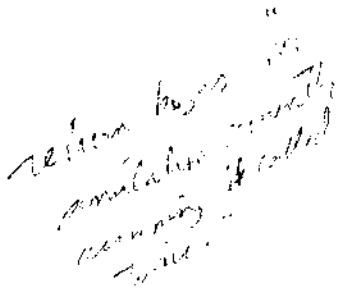
API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_BC_OneShotInit

Description

This routine is used to initialize the BC for BC-ONLY operation. The caller must specify the following parameters:

- **Channel:** The "wChannel" parameter specifies on which channel the BC activity is output (either channel A or B).
- **No Response Timeout:** The "wTimeout1" parameter specifies the "No Response" time-out period (in microseconds). If the RT specified in any BC command does not respond to the command within this time-out period, a "No Response" error is logged by the BC (and an interrupt is generated if that interrupt condition is enabled). The specified value must be between 4 and 32 microseconds.
- **Late Response Timeout:** The "wTimeout2" parameter specifies the "Late Response" time-out period (in microseconds). If the RT specified in any BC command does not respond to the command within this time-out period, a "Late Response" error is logged by the BC (and an interrupt is generated if that interrupt condition is enabled). The "Late Response" time-out period must be less than the "No Response" time-out period. The specified value must be between 4 and 32 microseconds.



NOTE: This routine is used in the BC-ONLY environment. It cannot be used in conjunction with the BM or any simulated RT's. In addition, there can be no interrupt capturing when using this routine. See Chapter 6, "**Bus Controller**", for an example which uses this mode to transmit messages on both Bus A and Bus B.

See Chapter 7 for an example of using this function.

NOTE: Call the function BusTools_BC_Init() to setup retries, if necessary, after calling this function and before calling **BusTools_BC_OneShotExecute()**. Calling this function clears all of the BC retry bits.

The only routines which can be used with this routine are:

- **BusTools_API_Init() or BusTools_API_InitExtended():**
- **BusTools_BC_Init():**
- **BusTools_BC_OneShotExecute():**
- **BusTools_SetBroadcast():**
- **BusTools_SetInternalBus():**
- **BusTools_SetSa31():**
- **BusTools_SetVoltage():**

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BC_OneShotInit( wCardnum, wChannel,
                                   wTimeout1, wTimeout2 );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wChannel	(BT_UINT) channel for BC output (microcode v1.03 only - newer versions do not use this; they use the field within the API_BC_MBUF structure which is passed to the **BusTools_OneShotExecute()** function): 1 -> channel A 0 -> channel B
wTimeout1	(BT_UINT) time-out period for the "No Response" error (in microseconds) -- must be between 4 and 32 microseconds.
wTimeout2	(BT_UINT) time-out period for the "Late Response" error (in microseconds) -- must be between 4 and 32 microseconds.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_BC_Start

Description

This routine is used to start the BC running at a specified message. It differs from BusTools_BC_StartStop in that the former routine can start the BC only with the first message in the list.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BC_StartStop(wCardnum, mblock_id);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
mblock_id	(BT_UINT) Message number in BC message list at which to start the BC running. Must be between 0 and 1 less than the number of messages defined in the call to BusTools_MessageAlloc().

Return Value

API_SUCCESS:
API_BUSTOOLS_NOTINITED:
API_BC_NOTINITED:
API_BC_RUNNING:
API_BUSTOOLS_BADCARDNUM:
API_BC_ILLEGAL_MBLOCK:

BusTools_BC_StartStop

Description

This routine is used to turn the BC on or off (with the requested state specified by the "wFlag" parameter).

For turning the BC off, the routine waits until the end of a minor frame before clearing the "BC Running" bit. However, if the minor frame does not end within approximately two seconds, the BC is forced off without regard to its current operational status. This can potentially leave the 1553 bus in an indeterminate state.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BC_StartStop(wCardnum, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wFlag	(BT_UINT) requested state for BC: 0 -> stop the BC. 1 -> start the BC.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BC_NOTINITED:
API_BC_NOTRUNNING:
API_BC_RUNNING:
API_BC_HALTERROR:

BusTools_BC_Trigger

Description

This routine is used to set the BC trigger mode (with the requested state specified by the "wFlag" parameter).

The Bus Controller can be started in one of three ways:

- Immediately after the "BC Run" bit is set (see **BusTools_BC_StartStop()**).
- After the BC Run bit is set, upon receipt of a trigger pulse on the hardware trigger input; thereafter it runs as programmed by the software (Microcode versions 1.04 and above).
- After the BC Run bit is set, upon receipt of a trigger pulse on the hardware trigger input; the Bus Controller runs one minor frame then waits for another hardware trigger input. This mode is only available for bus lists which have at least 20 milliseconds of "dead" time following the last message in the minor frame, to allow time for the API software to re-arm the hardware trigger (Microcode versions 2.01 and above).

The default is to start immediately after the "run" bit is set via a call to **BusTools_BC_StartStop()**. The function can be used to force the Bus Controller to run synchronized to an external pulse. For further information about hardware triggering of the Bus Controller, please refer to the **Hardware Reference Manual**.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Controller must be initialized (using the **BusTools_BC_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BC_StartStop(wCardnum, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wFlag	(BT_INT) requested state for BC: BC_TRIGGER_IMMEDIATE BC_TRIGGER_ONESHOT BC_TRIGGER_REPETITIVE

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BC_NOTINITED:
API_BC_NOTRUNNING:
API_BC_RUNNING:
API_BC_HALTERROR:

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 combination of RT address, RT sub-address, transmit/receive flag, and word count (or modecode) can be enabled or disabled for BM recording.

This procedure reads the specified BM Filter Buffer from BusTools memory for the specified RT sub-unit. The information is returned in the caller-supplied structure. The enable/disable information is in the form of a 32-bit word, with each bit enabling or disabling a particular word count (or modecode) for the specified RT sub-unit (see the definition of the API_BM_CBUF structure in Appendix B).

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BM_FilterRead( wCardnum, wRtaddr, wSubaddr,
                                  wTrans, pFilter);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0 -> receive)
pFilter	(API_BM_CBUF*) pointer to BM filter structure (see Appendix B for structure definition).

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:

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 combination of RT address, RT sub-address, transmit/receive flag, and word count (or modecode) can be enabled or disabled for BM recording.

This procedure creates/updates the Filter Buffer information for the specified RT sub-unit. If this is the first time this routine has been called for the specified RT sub-unit (since calling the **BusTools_BM_Init()** routine), the Filter Buffer is allocated in BusTools memory. If this Filter Buffer has already been allocated, it is updated with the new information.

By default all combinations of RT address, sub address, transmit and receive, and word count are enabled. This is accomplished by pointing every entry of the filter buffer to a single BM filter buffer which enables all of the combinations.

Note that by default all Bus Monitor addresses are *enabled*, so it is not necessary to call this function at all unless some RT address combinations are to be filtered out, e.g., not recorded. Following this rule will conserve memory on the BusTools board.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BM_FilterWrite( wCardnum, wRtaddr, wSubaddr,
                                   wTrans, pFilter );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0 -> receive)
pFilter	(API_BM_CBUF*) pointer to BM filter structure (see Appendix B for structure definition).

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:
API_BM_MEMORY_OFLOW:

BusTools_BM_Init

Description

This procedure performs a full initialization of the Bus Monitor functionality. In addition, since the BM is assumed to be a part of every 1533 simulation, this routine also initializes and resets the memory management functions for BusTools memory.

This routine performs a hardware reset of the BusTools board. Then, BusTools memory is initialized and reset to the default configuration. All error counters are reset. Time counters are reset (all time values reported in BusTools structures are relative to the time at which this routine was last called). Finally, the BM is enabled for the channels specified in the arguments to this routine.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines). In addition, if this routine is called after other BusTools activity has occurred, the caller must ensure that no part of the simulation (BC, BM or RT) is currently running.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BM_Init(wCardnum, wEnableA, wEnableB);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wEnableA	(BT_UINT) flag indicating if the BM should monitor channel A: 0 -> do not monitor channel A. 1 -> monitor channel A.
wEnableB	(BT_UINT) flag indicating if the BM should monitor channel : 0 -> do not monitor channel B. 1 -> monitor channel B.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BC_RUNNING:
API_BM_RUNNING:

BusTools_BM_MessageAlloc

Description

This routine allocates and initializes the specified number of BM Message Buffers in the BusTools memory. If the specified number of Message Buffers doesn't fit, (or the caller specified "-1" buffers), the routine allocates as many buffers as fit into the remaining memory. The routine returns the number of Message Buffers actually allocated to the caller.

If the application is not going to use the Bus Monitor, this function should still be called to allocate 2 buffers to insure that the API memory allocation functions are correctly setup.

Note that BusTools allocates 250 message buffers; you should allocate enough message buffers to hold those messages which will be received between calls to **BusTools_BM_MessageReadBlock()** or **BusTools_BM_MessageRead()**. This can depend on the operating system and the expected level of 1553 message traffic to be monitored.

The Interrupt Enable Bits specified by the caller are placed into all Message Buffers. See Appendix B for definitions of the Interrupt Enable Bits ("Interrupt Enable / Message Status Bits (32 Bit)").

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BM_MessageAlloc( wCardnum, wCount,
                pwActual, dwIntEnable );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wCount	(BT_UINT) requested number of BM Message Buffers.
pwActual	(BT_UINT*) pointer to actual number of buffers allocated (returned by routine).
dwIntEnable	(BT_U32BIT) interrupt enable flags for BM activity (see Appendix B for bit definitions).

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BM_NOTINITED:
API_BM_RUNNING:

BusTools_BM_MessageConvert

Description

This routine has been deleted, since it is no longer needed.

This routine was previously used to convert a raw BM Message Buffer (structure API_BM_MBUF_RAW) into the formatted BM Message Buffer (structure API_BM_MBUF). The raw Message Buffers were returned by the **BusTools_BM_MessageReadBlock()** routine during BM recording operations.

Since the **BusTools_BM_MessageReadBlock()** function supported by this version of the API converts the Bus Monitor message before returning it, the API_BM_MBUF_RAW format and this conversion routine no longer exist.

BusTools_BM_MessageGetid

Description

This routine converts a BusTools hardware address to a BM Message Buffer number. The specified address is a byte offset from the beginning of memory on the BusTools board. The address must be in the range 0x00000000 to 0x0003FFFF. The BM Message Buffer number is "0" based and has a range of "0" to the number of Message Buffers allocated by the **BusTools_BM_MessageAlloc()** routine, less one.

Typically, this routine is used by the Windows® task which processes the BM interrupt message. The BM message address supplied as a parameter to this task is converted to a message number and then the message can be retrieved from the board using the **BusTools_BM_MessageRead()** routine.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BM_MessageGetid( wCardnum, dwAddr,
                                    pwMessageid);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
dwAddr	(BT_U32BIT) BusTools memory address of BM Message Buffer (as supplied to the Windows® task handling BM interrupts).
pwMessageid	(BT_UINT*) address of returned BM Message Buffer number (returned by routine).

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BM_NOTINITED:
API_BM_MBUF_NOMATCH:

BusTools_BM_MessageRead

Description

This routine transfers the contents of the specified BM Message Buffer to the caller-supplied structure. The specified Message Buffer number must be in the range of the available Message Buffers (as specified in the **BusTools_BM_MessageAlloc()** routine).

Note that the Message Buffer returned by this routine is API_BM_MBUF.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_BM_MessageRead( wCardnum, wMessageid,
                                   pMessage );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wMessageid	(BT_UINT) BM Message Buffer number ("0" based).
pMessage	(API_BM_MBUF*) Address of Message Buffer structure to be filled by this routine.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BM_NOTINITED:
API_BM_ILLEGAL_MBUFID:

BusTools_BM_MessageReadBlock

Description

This routine transfers all BM Message Buffers recorded in BusTools memory since the last call to the routine. The caller supplies the buffer into which the data is stored (an array of API_BM_MBUF structures used as a circular buffer). The routine returns the number of messages actually transferred. In addition, this is the routine which updates the cumulative error counters (to get the current settings of the error counters, use the **BusTools_ErrorCountGet()** routine).

Note that the Message Buffer structures returned by this routine are the formatted Messages (API_BM_MBUF).

The application will receive a PostMessage every 500 ms, or when there are 64K bytes of messages available for recording. Note that this routine never returns more than 65535 bytes of data in a single call, which is the largest array of data which can be written to the disk in a single operation under 16-bit Windows®. Your application should insure that the supplied array of Message Buffers is large enough to handle the volume of messages expected between calls.

If the user has NOT called **BusTools_RecordRegister()**, the API will not be collecting and saving BM messages. In this case, calling the **BusTools_BM_MessageReadBlock()** function is not supported. Any calls to this function will return zero messages. See the routine **BusTools_RecordRegister()** in this chapter for further information.

This routine is currently not supported under DOS.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

16- and 32-bit Windows®

Calling Sequence

```
wStatus = BusTools_BM_MessageReadBlock( wCardnum, pMessages,
                                        wSize, wCurpos, pwCount );
```

wStatus	(BT_INT) status from routine, 0 indicates success, API_BM_WRAP_AROUND if API buffer has overflowed.
wCardnum	(BT_UINT) BusTools card number (0 - 3).
pMessages	(API_BM_MBUF*) address of beginning of caller's message array.
wSize	(BT_UINT) number of message structures in "pMessages" array.
wCurpos	(BT_UINT) current position within "pMessages" array (this is the message number within the array where this routine stores the first message transferred).

pwCount (BT_UINT*) number of messages transferred (returned by routine).

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BM_NOTINITED:
API_BM_WRAP_AROUND:

BusTools_BM_StartStop

Description

This routine is used to turn the Bus Monitor simulation on or off (with the requested state specified by the "wFlag" parameter).

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BM_StartStop(wCardnum, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wFlag	(BT_UINT) requested state for BM: 0 -> stop the BM. 1 -> start the BM.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BM_NOTINITED:
API_BM_NOTRUNNING:
API_BM_RUNNING:

BusTools_BM_TriggerRead

Description

This routine transfers the contents of the BM Trigger Buffer to the caller-supplied structure.

Currently this routine returns success, but does **not** actually read and return the contents of the BM Trigger Buffer.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BM_TriggerRead(wCardnum, pTrigger);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
pTrigger	(API_BM_TBUF*) address of Trigger Buffer structure to be filled in by routine.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BM_NOTINITED:

BusTools_BM_TriggerWrite

Description

This routine transfers the contents of the caller-supplied structure to the BM Trigger Buffer in BusTools memory.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_BM_TriggerWrite (wCardnum, pTrigger);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
pTrigger	(API_BM_TBUF*) address of Trigger Buffer structure to be transferred to BusTools memory.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BM_NOTINITED:

BusTools_EI_EbufWrite

Description

This routine specifies the contents of an Error Injection Buffer. The contents of the Error Injection Buffer definition structure is defined in Appendix B, under the heading "API_EIBUF".

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines)and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

BusTools normally creates buffer "0" as all zero, indicating no errors are to be injected, and uses this buffer for all BC and RT functions which take a pointer to an error injection buffer. This is a good convention for a user-written application to follow.

For the PC-1553/PC-1773 note that although the RT simulation is located in the second segment, the error injection buffers it uses are located in the first segment.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_EI_EbufWrite(wCardnum, wBufferid, pBuffer);

wStatus	(BT_INT) status from the routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wBufferid	(BT_UINT) Error Injection Buffer number ("0" based).
pBuffer	(API_EIBUF*) pointer to Error Injection Buffer definition structure (see Appendix B for structure definition).

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:

BusTools_EI_Getaddr

Description

This routine converts an Error Injection Buffer number to a BusTools hardware address. The returned address is a byte offset from the beginning of memory on the BusTools board. The address is the range 0x00000000 to the highest board address within the first segment.

Note that this routine should only be required for debugging operations. Normally, the programmer does not need to access absolute memory addresses on the BusTools board.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_EI_Getaddr(wCardnum, wBufferid, pdwAddr);

wStatus	(BT_INT) status from the routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0-3).
wBufferid	(BT_UINT) Error Injection Buffer number ("0" based).
pdwAddr	(BT_U32BIT*) pointer to returned address of the specified Error Injection Buffer

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BM_NOTINITED:
API_EI_ILLERRORNO:

BusTools_EI_Getid

Description

This routine converts a BusTools hardware address to an Error Injection Buffer number. The specified address is a byte offset from the beginning of memory on the BusTools board. The Address must be in the range 0x00000000 to 0x0003FFFF.

Typically, this routine is used internally by the BusTools API to convert a memory address stored in the hardware to the EI Buffer number returned in various message structures. This routine typically is not be used by the programmer.

Prior to calling routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the BusTools_BM_Init() routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_EI_Getid(wCardnum, dwAddr, pwBufferid);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
dwAddr	(BT_U32BIT) memory address (0x00000000 - 0x0003FFFF).
pwBufferid	(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:

BusTools_ErrorCountClear

Description

When the BM recording is active, the BusTools API maintains a running count of how many messages have been detected and the cumulative number of errors detected of each type (for the definitions of the error/status bits, see the "Interrupt Enable / Message Status Bits" in Appendix B). The counters are reset each time the Bus Monitor simulation is re-initialized (using the **BusTools_BM_Init()** routine). This routine is used to reset these counters at any time after the simulation has been started.

Note that the counters are updated when BM Message Buffers are retrieved from the BusTools memory using the **BusTools_BM_MessageReadBlock()** routine.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_ErrorCountClear(wCardnum);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_ErrorCountGet

Description

When the BM recording is active, the BusTools API maintains a running count of how many messages have been detected and the cumulative number of errors detected of each type (for the definitions of the error/status bits, see the "Interrupt Enable / Message Status Bits" in Appendix B). The counters are reset each time the Bus Monitor simulation is re-initialized (using the **BusTools_BM_Init()** routine). This routine is used to get the current values of the counters.

The counters are all DWORD values. The buffer supplied by the caller to be filled in must contain at least 32 DWORD entries. The buffer entries are 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_MID_SYNC errors(NI).
- 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_TWO_BUS errors(NI).
- 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 msgs.
- buffer[19]: number of BT1553_INT_RESET_RT messages.
- buffer[20]: number of BT1553_INT_SELF_TEST messages.
- buffer[21]: number of BT1553_INT_BIT_FAILURE errors(NI).
- buffer[22]: number of BT1553_INT_INVALID_CMDW errors. (Not Implemented IP-1553 V2.x)
- buffer[23]: number of 1553 Status Word "Service Request"
- buffer[24]: number of 1553 Status Word "Broadcast Received".
- buffer[25]: number of 1553 Status Word "Busy"
- buffer[26]: number of BT1553_INT_NO_RESP errors.
- buffer[27]: number of BT1553_INT_ME_BIT errors (ME bit)
- buffer[28]: number of 1553 Status Word "Subsystem Flag"
- buffer[29]: number of 1553 Status Word "Terminal Flag"
- buffer[30-31]: (reserved)

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_ErrorCountGet(wCardnum, pdwCount, pdwBuffer);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
pdwCount	(BT_UINT*) pointer to variable to receive the total number of messages processed.
pdwBuffer	(BT_U32BIT*) pointer to a 32 element buffer to receive the 32 error counters.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_GetAddr

Description

This procedure returns the starting and ending memory addresses for a specified block of memory. The returned values are byte offsets from the beginning of memory on the BusTools board. The offsets are in the range 0x00000000 to the highest board offset.

The available memory block types are as follows:

- GETADDR_HWREG: Hardware registers(Value is 1).
- GETADDR_RAMREG: Microcode registers(2)
- GETADDR_BCMESS: BC Message Blocks(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)

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_GetAddr(wCardnum,wMemtype,pdwStart,pdwEnd);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wMemtype	(BT_UINT) Memory type (see above for possible types).
pdwStart	(BT_U32BIT*) pointer to returned beginning address of specified memory block.
pdwEnd	(BT_U32BIT*) pointer to returned ending address of specified memory block.

Return Value

API_SUCCESS:

Notes

1. The symbol "GETADDR_COUNT" defines the number of available memory block types.

2. The "GETADDR_*" definitions are "1" based, not "0" based, as is the convention in the "C" programming language, so be sure to dimension the receiving array (if one is used) by "GETADDR_COUNT"+1.

3. 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".

BusTools_GetData

Description

This routine is obsolete, and will be removed from a future version of the API. Please see BusTools_MemoryRead() for an equivalent function.

BusTools_GetRevision

Description

This routine returns to the caller the revision of the API and the microcode.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_GetRevision(wCardnum,pwRev1,pwRev2);

wCardnum	(BT_INT) BusTools card number (0 - 3)
pwRev1	(BT_UINT*) pointer to variable to be filled with microcode revision number. Rev number will be 100, 101, 102, etc., representing rev 1.00, 1.01, 1.02, etc. The IP-1553 adds 10000 to this revision for the ISA-based carrier while the PCI-based carrier adds 20000. The VME carrier adds 30000.
pwRev2	(BT_UINT*) pointer to variable to be filled with API revision number. Rev number will be 100, 101, 102, etc., representing rev 1.00, 1.01, 1.02, etc.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_InterMessageGap

Description

This routine is used to compute the intermessage time gap between two successive messages which have been recorded by the Bus Monitor. This is accomplished by first calculating the difference between the two timetags, then subtracting the amount of time that the first message used on the bus.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

dStatus = BusTools_InterMessageGap(first, second);

first	(API_BM_MBUF FAR *) pointer to first message.
second	(API_BM_MBUF FAR *) pointer to second message.

Return Value

dStatus(BT_U32BIT) - Intermessage gap time in microseconds.

BusTools_InterruptDone

Description

This routine must be called by an application program when the high-level (Windows® level) processing for a particular interrupt (either BC, BM or RT) has been completed.

The interrupt service routines maintain a count of "unfinished" interrupt messages. If this number gets too high or doesn't go to zero in a specified amount of time (a few seconds), the driver issues an interrupt overflow error and reduces the number of interrupt messages sent to the Windows® task.

This "filtering" of messages eliminates the oldest messages from the API queue, and only sends the latest messages to the application.

OS Support

16- and 32-bit Windows®

Calling Sequence

wStatus = BusTools_InterruptDone(wCardnum);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).

Return Value

API_SUCCESS:

BusTools_InterruptRegister

Description

This routine registers a window and message id to be used when there is an entry in the Interrupt Queue for the specified module (BC, BM, or specific RT). The caller specifies the card number (0 - 3) and the interrupt type (e.g. "API_INTTYPE_BC").

This routine should not be called until your window has been initialized and is ready to receive messages. The API delays for a fraction of a second before sending messages to a newly registered window to help insure that the window has time to display itself.

When the specified interrupt type is detected by the API interrupt routine, the API performs a "PostMessage()" function to send a message to the registered window. The API automatically queues these messages internally, and meters them out at a rate which does not exceed 20 Hz, to avoid overrunning the Windows® PostMessage() queue, and causing the host to appear to "freeze" or "lock up" due to excessive PostMessage() traffic.

If the rate of interrupts fills the internal API buffer, interrupts will be lost.

Format of PostMessage() call by the API:

PostMessage(hwnd, messageid, 0, addr_message);

where "hwnd" is the handle of the window registered to receive this message,

"messageid" is the message ID specified in the BusTools_InterruptRegister() call, and

"addr_message" is the byte offset of the message, on the board, which caused this PostMessage() to be sent.

The application must call **BusTools_InterruptDone()** when it is ready to receive another PostMessage(), otherwise the API will not post another message to the window.

See **BusTools_RegisterFunction()** for another interrupt register function and method which is more suited for operation under 32-bit operating systems. The API supports both interrupt methods concurrently in a single program, although new 32-bit programs will probably not want to use the **PostMessage()** interface supported by this function.

OS Support

16- and 32-bit Windows®

Calling Sequence

```
wStatus = BusTools_InterruptRegister( wCardnum, wIntType, hWnd,
                                      wMessageid, wFlag );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wIntType	(BT_UINT) Interrupt type (e.g. "API_INTTTYPE_BC", etc.).
hWnd	(HWND) Window handle of task to get message.
wMessageid	(BT_UINT) Message id to be sent when interrupt detected. A value of zero "unregisters" the interrupt window; e.g., no further PostMessages() are desired.
wFlag	(BT_UINT) flag indicating what to do: 0 -> "unregister" this routine. 1 -> register the window/messageid.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BUSTOOLS_BADINTTYPE:

BusTools_MemoryAlloc

Description

This routine allows for a block of memory to be allocated on the BusTools board for use by a user's program. Currently, the only use for this block would be for the referencing of one or more words by conditional BC messages.

It should be noted that this routine must only be called after **BusTools_BM_Init()** has been called, and before **BusTools_XX_StartStop()** has been called to start the execution of the board's microcode.

This function only allocates memory in SEG1.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus =BusTools_MemoryAlloc (wCardnum, wSegnum, dwCount,
                               pdwAddr);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3)
wSegnum	(BT_UINT) memory segment number (0->SEG1)
dwCount	(BT_U32BIT) number of bytes to reserve
pdwAddr	(BT_U32BIT*) pointer to variable to be filled with the starting address of the reserved block.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BUSTOOLS_BADSEGNUM
API_BUSTOOLS_EVENBCOUNT

BusTools_MemoryRead

Description

This routine provides direct read access to BusTools memory.

The caller must supply the offset from the beginning of BusTools memory, the number of bytes to transfer and the host address of a buffer to receive the data.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_MemoryRead(wCardnum, dwAddr, wdwCount,
                              pBuffer);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools board number
dwAddr	(BT_U32BIT) BusTools memory address to be read
dwCount	(BT_U32BIT) number of bytes to be read (must be even, and less than 64K bytes)
pBuffer	(VOID*) Host buffer into which data will be transferred. Buffer must NOT cross a 64K-segment boundary in 16-bit applications.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED

BusTools_MemoryReadIDPROM

Description

The IP-1553-xx modules provide an ID PROM area, in accordance with the IP Modules Draft Standard, VITA 4-1995.

This routine provides read access to the ID PROM on the IP-1553 module. This function is not supported for PC-1553 or PCI-1553 boards.

The caller supplies the BusTools board number, the number of bytes to read, and the address of a buffer that will receive the contents of the ID PROM upon function return.

The ID PROM is read as bytes, and returned in the caller's buffer as bytes.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_MemoryReadIDPROM(wCardnum, nChars,
                                    lpsPROM);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools board number
nChars	(BT_INT) number of characters to read
lpsPROM	(CHAR *) buffer into which data will be transferred. Must NOT cross a 64K-segment boundary in 16-bit applications.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM;
API_HARDWARE_NOSUPPORT
API_BUSTOOLS_NOTINITED

Bustools_MemoryWrite

Description

This routine provides direct write access to BusTools memory.

The caller must supply the offset from the beginning of BusTools memory, the number of bytes to transfer and the host address of a buffer containing the data to be written to the BusTools board.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_MemoryWrite(wCardnum, dwAddr, dwCount,
                               pBuffer);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools board number
dwAddr	(BT_U32BIT) BusTools memory address to be written
dwCount	(BT_U32BIT) number of bytes to be written (must be even, and less than 64K bytes)
pBuffer	(VOID*) buffer from which data will be transferred. Must NOT cross a 64K-segment boundary in 16-bit applications.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_RecordDone

Description

This routine is called by an application when the high-level (Windows® level) processing for Bus Monitor recording has been completed. The interrupt service routines maintain a count of "unfinished" recording messages. While this number is non-zero, the driver does not issue another recording message. If, when the application finally calls **BusTools_BM_MessageReadBlock()**, the internal API buffer has overflowed, that routine returns an overflow status to the calling routine. Once the overflow condition has been detected. there is no guarantee that the BM recording activity is getting all messages.

Likewise, the application must insure that it calls this routine once for each recording message which it receives, since it does not receive another message until it has acknowledged the previous one by calling this function.

OS Support

16- and 32-bit Windows®

Calling Sequence

wStatus = BusTools_RecordDone(wCardnum);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_RecordGetFlag

Description

This routine can be called by an application to determine if Bus Monitor recording is still active.

This function returns a flag which indicates if a window is currently registered to receive Bus Monitor recorder interrupts via the PostMessage() Windows® interface.

OS Support

16- and 32-bit Windows®

Calling Sequence

wStatus = BusTools_RecordGetFlag(wCardnum, pFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
pFlag	(BT_INT *) pointer to a flag which is set to zero if no window is currently registered to receive Bus Monitor record interrupts, or non-zero if such a window exists.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:

BusTools_RecordRegister

Description

This routine registers a window and messageid to be used for the periodic Bus Monitor recording message. The caller specifies the card number.

The API calls PostMessage(hwnd, API_BMRECORD_MESSNO, 0, 0); where "hwnd" is the handle of the window specified in the call to **BusTools_RecordRegister()** whenever there is 64 Kbytes of BM data ready to record, or every 500 milliseconds.

The API will call PostMessage(hwnd, API_BMRECORD_MESSNO, 0, n); whenever a BM trigger event occurs, where "n" contains a one (1) if this is the trigger start event, or a two (2) if this is the trigger end event, if BM triggering is enabled.

The API will call PostMessage(hwnd, API_LOWERROR_MESSNO, API_LOW_INT_OFLOW, 0); if the API's internal Bus Monitor buffer overflows.

Once the application program has completed its processing, it must call **BusTools_RecordDone(),** or it will never receive another PostMessage().

OS Support

16- and 32-bit Windows®

Calling Sequence

wStatus = BusTools_RecordRegister(wCardnum, hWnd, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
hWnd	(HWND) Windows® handle.
wFlag	(BT_UINT) flag indicating what to do: 0 -> "unregister" this routine. 1 -> register the window/messageid, and begin sending record messages to the window.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_RegisterFunction

Description

This routine creates and registers a function that is to be called by a newly created thread when any of a list of user-specified events occurs. This thread waits on an Event Object until one of a list of user-specified event occurs. The API signals the Event Object; this causes the user function to be called. Since the user's function is called by a thread, priority changes and multi-processing by multiple threads is supported.

Information about the events is supplied to the function via a user-supplied FIFO structure. The API fills in an entry in the user-supplied FIFO structure and calls the user function passing it arguments that identify the exact FIFO that contains the event information.

The user's function should process all of the entries in the FIFO, as appropriate for the application. The thread should then return to the API (simply execute a "return") and sleep (e.g., "WaitForSingleObject()") until another event was detected and reported to it.

It is important that the user function process all of the entries in the FIFO when it is called, since an Event Object will only release the thread once, and there may be multiple entries in the FIFO when the thread is released.

The user is free to set the priority of the specified thread as appropriate for the application. Normally the priority of the threads should be adjusted based on their importance; for example, normally a display thread would have a lower priority than a thread that was performing a simulation function, or a thread that was recording data for later analysis.

OS Support

32-bit Windows®

Calling Sequence

wStatus = BusTools_RegisterFunction(wCardnum, &sIntFIFO, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
sIntFIFO	(API_INT_FIFO *) Pointer to Thread Interrupt Filter, Control and FIFO structure.
wFlag	(BT_UINT) flag indicating what to do: 0 -> "unregister" this 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:

1. The function whose address is specified in the API_INT_FIFO structure variable "**function**" is called by a new thread which is created by the call to **BusTools_RegisterFunction()**. This new thread is created with the priority specified in the API_INT_FIFO structure variable "**iPriority**". This thread waits on the Event Object created by the **BusTools_RegisterFunction()** until an interrupt event occurs which is selected by the API_INT_FIFO filtering structure. At that time the thread is released and calls the user function. When the user function has completed processing all of the event(s) in the FIFO, it simply returns to wait for the next event.
2. The calling sequence used by the API to call the user function is described in the API_INT_FIFO structure definition, which is defined in BusApi.H and documented in Appendix B, "Data Structures". Note that the function is passed both the "**cardnum**" and a pointer to the API_INT_FIFO structure. This information is sufficient for a single function to determine the required processing for multiple event registrations.
3. The user may update the API_INT_FIFO filtering structure at any time. Changes take effect immediately. Since the API is interrupt driven, take care to modify the filtering structure as quickly as possible to avoid being flooded with unwanted events, remembering that the API might look at the structure while you are updating it.
4. The API_INT_FIFO structure will be updated by this function call, if "**wFlag**" is non-zero, to contain the handle of an Event Object that will be signaled by the API whenever an event of interest is added to the FIFO. The user should not modify this field, as well as all fields that are marked "RO", since the API uses these parameters internally.
5. Before exiting, your program must call **BusTools_RegisterFunction()** a second time with a "**wFlag**" of zero for each API_INT_FIFO you have setup. This will cause the thread that was created to terminate, and the objects released. Otherwise these threads will not be "cleaned up" by Windows®, and the API and your program or .DLL will remain in memory after your application has terminated. This second call will raise the thread priority to THREAD_PRIORITY_TIME_CRITICAL, request the thread to terminate, signal the thread's Event Object, and then return to the caller without waiting for the thread to terminate and release its objects. "**bForceShutdown**" may be polled to determine when the thread has terminated.
6. The priority of a terminating thread calling a user exit function is set to THREAD_PRIORITY_TIME_CRITICAL, so a user exit function should not perform a lot of processing. If it is necessary to perform lengthy processing, the thread should first lower its priority.
7. Do not call **BusTools_RegisterFunction()** with a "**wFlag**" of zero from within the thread function to terminate the thread! This will cause a crash. If the thread function wishes to terminate itself, return from the function with a non-zero return value. This will cause the API to terminate the thread and cleanup the objects. In this case you should not call **BusTools_RegisterFunction()** to terminate the thread!
8. See the definition of the API_INT_FIFO structure in BusApi.H and in Appendix B, "Data Structures" for further information.

BusTools_RT_AbufRead

Description

This routine is used to read the information currently stored in the RT Address Control Buffer for the specified RT address. This Buffer controls the operation of the specified RT. Elements in this structure control whether the specified RT is active on the bus.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_RT_AbufRead(wCardnum, wRtaddr, pAbuf);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
pAbuf	(API_RT_ABUF*) address of structure to be filled in by routine.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_RT_NOTINITED:
API_RT_ILLEGAL_ADDR:

BusTools_RT_AbufWrite

Description

This routine is used to write the "enables" word to the RT Address Control Block for the specified RT address. This Block controls the operation of the specified RT. Elements in this structure control whether the specified RT is active on the bus.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_RT_AbufWrite(wCardnum, wRtaddr, pAbuf);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
pAbuf	(API_RT_ABUF*) address of structure to be transferred to BusTools memory.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_RT_NOTINITED:
API_RT_ILLEGAL_ADDR:

BusTools_RT_CbufbroadRead

Description

This routine reads the Broadcast Control Buffer for the specified RT sub-address. This Buffer contains the "legalization" bits used to enable or disable broadcast messages for the specified sub-address. The caller-supplied structure contains a single bit for each possible RT address and each possible word count (for a total of 32 x 32 or 1024 bits).

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_RT_CbufbroadRead( wCardnum, wSubaddr,
                                     pCbufBroad );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
pCbufBroad	(API_RT_CBUFBROAD*) address of RT Broadcast Control Buffer to be filled in by routine.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_RT_NOTINITED:
API_RT_ILLEGAL_SUBADDR:

BusTools_RT_CbufbroadWrite

Description

This routine writes the Broadcast Control Buffer for the specified RT sub-address. This Buffer contains the "legalization" bits used to enable or disable broadcast messages for the specified sub-address. The caller-supplied structure contains a single bit for each possible RT address and each possible word count (for a total of 32 x 32 or 1024 bits).

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_RT_CbufbroadWrite (wCardnum, wSubaddr, wTrans, pCbufBroad);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0->receive)
pCbufBroad	(API_RT_CBUFBROAD*) address of RT Broadcast Control Buffer to be transferred to BusTools memory.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_RT_NOTINITED:
API_RT_ILLEGAL_SUBADDR:

BusTools_RT_CbufRead

Description

This routine reads the RT Control Buffer for the specified RT sub-unit. This structure contains a "legalization" bit for each possible word count for the sub-unit (see the structure definition in Appendix B). This routine 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 calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_RT_CbufRead( wCardnum, wRtaddr, wSubaddr,
                                wTrans, pwCount, pCbuf );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0 -> receive)
pwCount	(BT_UINT*) number of RT Message Buffers originally defined for this RT sub-unit (returned by routine).
pCbuf	(API_RT_CBUF*) address of RT Control Buffer to be filled in by routine.

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:

BusTools_RT_CbufWrite

Description

This routine writes the RT Control Buffer for the specified RT sub-unit. This structure contains a "legalization" bit for each possible word count for the sub-unit (see the structure definition in Appendix B). The caller also specifies the number of RT Message Buffers to be allocated for the specified RT sub-unit. The Message Buffers are allocated and the data area is cleared (filled with zeros). For "transmit" buffers, the caller should write data to the Message Buffers (using the **BusTools_RT_MessageWrite()** routine).

The first time this routine is called (after having BusTools memory reset by the **BusTools_BM_Init()** routine), the specified number of RT Message Buffers are allocated. This routine can be called again to change the RT sub-unit "legalization" bits. However, any future calls can't specify a larger number of RT Message Buffers (an error is returned).

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_RT_CbufWrite( wCardnum, wRtaddr, wSubaddr,
                                 wTrans, wCount, pCbuf );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0 -> receive).
wCount	(BT_UINT) number of RT Message Buffers to be allocated.
pCbuf	(API_RT_CBUF*) address of RT Control Buffer to be filled in by routine.

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:

BusTools_RT_Init

Description

This function performs full initialization of RT functionality on the board. The following actions are performed:

- check for any illegal conditions (exit with error if these conditions aren't true):
- BusTools must be initialized
- no RT's can be running
- setup memory in hardware:
- available memory starts at beg of segment (address = 10000)
- allocate filter buffer
- allocate address control blocks
- allocate default message buffer
- allocate broadcast control buffer (if broadcast is enabled)
- allocate default control buffer (non-broadcast)
- setup PC data structures
- show no RT's running

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines) and the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine).

The wFlag 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

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_RT_Init(wCardnum, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wFlag	(BT_UINT) Test Flag. **Must** be **zero** for normal operation!

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_RT_RUNNING:
API_RT_SELF_TEST_MODE:
API_MEMORY_OFLOW:

BusTools_RT_MessageGetaddr

Description

This routine returns the physical board offset of the specified RT Message Buffer. The caller specifies the MBUF number for a specific RT sub-unit.

Typically, this routine is used by the a user's application which repeatedly updates a particular RT message using the **BusTools_MemoryWrite()** function instead of the **BusTools_RT_MessageWrite()** function.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_RT_MessageGetaddr( wCardnum, wRtaddr,
                wSubaddr, wTrans, wMessageid, pwMbuf);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0 -> receive).
wMessageid	(BT_UINT) RT Message Buffer number ("0" based).
pwMbuf	(BT_U32BIT *) address of dword to receive the board offset to the specified RT_MBUF structure, 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:

BusTools_RT_MessageGetid

Description

This routine converts a BusTools hardware address to an RT sub-unit and Message Buffer number. The specified hardware address is a byte offset from the beginning of memory on the BusTools board. The address must be in the range 0x00000000 to 0x0003FFFF.

Typically, this routine is used by the Windows® task that processes the RT interrupt messages. The RT message address supplied as a parameter to this task is converted to an RT sub-unit address and message number and then the message can be retrieved from the board using the **BusTools_RT_MessageRead()** routine.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_RT_MessageGetid(wCardnum, dwAddr, pwRtaddr, pwSubaddr, pwTrans, pwMessageid);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
dwAddr	(BT_U32BIT) BusTools memory address for RT Message Buffer.
pwRtaddr	(BT_UINT*) RT address for message (returned by routine).
pwSubaddr	(BT_UINT*) RT sub-address for message (returned by routine).
pwTrans	(BT_UINT*) transmit/receive flag for message (returned by routine).
pwMessageid	(BT_UINT*) RT Message Buffer number for specified RT sub-unit (returned by routine, "0" based).

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_RT_NOTINITED:
API_RT_MBUF_NOMATCH:

BusTools_RT_MessageRead

Description

This routine reads the specified RT Message Buffer from BusTools memory. The specified structure is filled with data.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_RT_MessageRead( wCardnum, wRtaddr, wSubaddr,
                                   wTrans, wMessageid, pMessage );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0 -> receive).
wMessageid	(BT_UINT) RT Message Buffer number ("0" based).
pMessage	(API_RT_MBUF_READ*) address of structure to be filled by routine.

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:

BusTools_RT_MessageWrite

Description

This routine is used to write the specified information to the specified RT Message Buffer in BusTools memory. Message Buffers must have been allocated for the specified RT sub-unit (using the **BusTools_RT_CbufWrite()** routine). The data (for "transmit" messages) and the Interrupt Enable Bits are the significant elements of the RT Message Buffer that must be initialized in BusTools memory.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_RT_MessageWrite(wCardnum, wRtaddr, wSubaddr, wTrans, wMessageid, pMessage);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0 -> receive).
wMessageid	(BT_UINT) RT Message Buffer number ("0" based).
pMessage	(API_RT_MBUF_WRITE*) address of RT message 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_ILLEGAL_MBUFID:

BusTools_RT_MessageWriteDef

Description

The BusTools API maintains a default (or "garbage collection") Message Buffer for each RT. The default Message Buffer is used for any messages to or from any sub-addresses which do not have specific RT Message Buffers allocated (using the **BusTools_RT_CbufWrite()** routine). This routine 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. The main reason that this routine 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 calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_RT_MessageWriteDef( wCardnum, wRtaddr,
                                       pMessage );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
pMessage	(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:

BusTools_RT_MessageWriteStatusWord

Description

The BusTools RT simulation provides a single Status Word for each RT address that is being simulated. This Status Word is transmitted by the simulation in response to any message directed to the associated RT.

The API extends this capability by providing the ability to change the RT Status Word that it transmitted by the RT simulation on a per-message basis. This is accomplished by having the Status Word update bits stored as a part of the RT message, along with a flag which tells the API to either modify the status word (based on the stored value), if the flag is set, or not to modify the current status word (if the flag is clear).

The RT Status Word is modified by the API interrupt function just after the associated message is referenced by the Bus Controller, and therefor will take effect on the *following* message to/from this RT. Note also that there must be enough time for the API to actually change the Status Word.

This function provides the means for modifying the message-specific Status Word and the associated flag.

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_RT_MessageWriteStatusWord( wCardnum,
                    wRtaddr, wSubaddr, wTrans, wMessageid,
                    wStatusWord, wFlag);
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wRtaddr	(BT_UINT) RT address (0 - 31).
wSubaddr	(BT_UINT) RT subaddress (0 - 31).
wTrans	(BT_UINT) Transmit/receive flag (0 -> receive).
wMessageid	(BT_UINT) RT Message Buffer number.
wStatusWord	(BT_UINT) New RT Status word
wFlag	(BT_UINT) flag; 0->do not modify status word, 1->modify status word;

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:

BusTools_RT_StartStop

Description

This routine is used to turn the RT simulations on or off (with the requested state specified by the "wFlag" parameter).

Note that there is no parameter indicating which RT to start -- all defined RT simulations are affected by this routine. Enabling and disabling of individual RT's is handled through the RT Control Buffer (see the **BusTools_RT_CbufWrite()** routine).

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_RT_StartStop(wCardnum, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wFlag	(BT_UINT) requested state for RT: 0 -> stop the RT. 1 -> start the RT.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_RT_NOTINITED:
API_RT_NOTRUNNING:
API_RT_RUNNING:

BusTools_SetBroadcast

Description

This routine is used to set the broadcast mode of the BusTools hardware. The BusTools hardware has two modes of operation with respect to broadcast messages:

- **Broadcast enabled:** In this mode, broadcast messages are supported. Any receive command sent to RT address 31 is treated as a message that is to be sent to all RT's. The PC1553_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 just like any other RT address.

This function must be called after **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines but before BusTools_RT_Init().

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_SetBroadcast(wCardnum, wBroadcast);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wBroadcast	(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:

BusTools_SetExternalSync

Description

The IP-1553 BusTools board has a free-running counter which counts a one megahertz clock. This counter is used to time-tag all of the messages that are recorded by the board.

This function enables the external reset of that counter by the TTL input signal. A high level pulse of at least 400 nanoseconds is required to reset the time-tag counter. Pulses less than 200 nanoseconds in width are ignored, while pulses between 200 and 400 nanoseconds may or may not reset the counter.

This is the same time-tag counter which is controlled by the *"Reset Timetag on Sync"* option which specifies that if a Sync modecode is detected, the hardware timetag counter is reset. Since the timetag is global to all components simulated on the board, this option should be used carefully. For more information see the **BusTools_SetOptions()** function.

Note that the API will not be notified of the fact that the time-tag counter has been reset.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_SetExternalSync(wCardnum, wflag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wFlag	(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:

BusTools_SetInternalBus

Description

The PC-1553 BusTools board has been designed to drive and / or monitor an external 1553 bus. However, it is also possible to create and run a BusTools model locally (totally on the board) without requiring an external bus cable or termination. This routine is used to set the bus operation mode of the BusTools 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 model (BC, BM, and RT's). The BM monitors all activity on the external bus. Any bus messages (even between two locally defined components) are transmitted on the external bus.
- **Internal Operation Only:** In this mode of operation, the external bus is ignored. All messages are simulated internally with no output to the external bus.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_SetInternalBus(wCardnum, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wFlag	(BT_UINT) external or internal bus operation mode: 0 -> external bus enabled 1 -> external bus disabled (internal operation)

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_HARDWARE_NOSUPPORT:

BusTools_SetOptions

Description

This function sets the various operation options for the BusTools board. operate. The available options are:

- **Interrupt on Illegal Commands:** This option specifies what should be done if an RT detects an illegal command (one with an invalid subaddress or word count). If the option is set, the RT microcode generates an interrupt if an illegal command is detected.
- **Reset Timetag on Sync:** This option specifies that if a Sync modecode is detected, the hardware timetag counter is reset. Since the timetag is global to all components simulated on the board, this option should be used sparingly (that is, it should only be used if the board is simulating a single RT).
- **Monitor Illegal Commands:** This option specifies how an RT should respond to an illegal command (one with an invalid subaddress or word count). If the option is set, the RT returns a status word to the BC with the Message Error bit set (and suppress or ignore any data words). If the option is not set, the RT responds to the command word as required (receiving or transmitting data), but the data are invalid. The Message Error bit is not set.
- **Treat Illogical Commands as Invalid:** Illogical commands are undefined modecode commands. They should never be transmitted by the BC. If an illogical command is received, it may be treated as an illegal command or an invalid command. If this bit is set, then illogical commands are treated as invalid (the command is ignored and no message is processed by the RT). If this bit is not set, then illogical commands are treated as illegal (the RT response depends on the setting of the "Monitor Illegal Commands" option).

Prior to calling this routine, BusTools must be initialized (using the **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines), the Bus Monitor must be initialized (using the **BusTools_BM_Init()** routine), and the Remote Terminal must be initialized (using the **BusTools_RT_Init()** routine).

This function must be called after **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines but before BusTools_RT_Init().

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

```
wStatus = BusTools_SetOptions( wCardnum, wIntFlag, wResetTimer,
                               wMonIllegal, wIllogicalIsInvalid );
```

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).

wIntFlag (BT_UINT) setting for "Interrupt on Illegal Commands" option:
0 -> option is disabled.
1 -> option is enabled.

wResetTimer (BT_UINT) setting for "Reset Timetag on Sync" option:
0 -> option is disabled.
1 -> option is enabled.

wMonIllegal (BT_UINT) setting for "Monitor Illegal Commands" option:
0 -> option is disabled.
1 -> option is enabled.

wIllogicalIsInvalid (BT_UINT) setting for "Treat Illogical Commands as Invalid" option:
0 -> option is disabled.
1 -> option is enabled.

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_RT_NOTINITED:

BusTools_SetSa31

Description

The BusTools board can support two modes of operation with respect to subaddress 31 and modecodes. This routine is used to select which mode to use. The two modes of operation are:

- Subaddress 31 is not used for modecodes.
- Subaddress 31 is used for modecodes.

This function must be called after **BusTools_API_Init()** or **BusTools_API_InitExtended()** routines but before BusTools_RT_Init().

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_SetSa31(wCardnum, wFlag);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wFlag	(BT_UINT) flag indicating subaddress 31 operation mode: 0 -> subaddress 31 is not used for modecodes 1 -> subaddress 31 is used for modecodes

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_SetVoltage

Description

The PC-1553 BusTools board can drive the 1553 bus with either "direct coupling" or "transformer coupling". In addition, the voltage levels generated on the bus can be precisely controlled. This routine 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.

Note that the value of voltage passed to this function is an **integer**, which is the value of the voltage in volts multiplied by 100.

Note that the IP-1553 product line, which provides jumpers to select "direct" or "transformer" coupling, does not support this function, and supplies fixed output voltage levels.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

wStatus = BusTools_SetVoltage(wCardnum, wVoltage, wCoupling);

wStatus	(BT_INT) status from routine (0 indicates success).
wCardnum	(BT_UINT) BusTools card number (0 - 3).
wVoltage	(BT_UINT) output voltage -- in volts*100
wCoupling	(BT_UINT) flag indicating coupling: 0 -> direct coupling 1 -> transformer coupling

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:
API_BUSTOOLS_BADCOUPLING:
API_BUSTOOLS_BADVOLTAGE:
API_HARDWARE_NOSUPPORT:

BusTools_StatusGetString

Description

This routine is used to convert a BusTools API error number into an ASCII string suitable for display to a user. The string would typically be used in a Windows® Message Box to inform you that there was an unexpected condition with respect to the BusTools hardware and/or software.

For a list of the error codes, which was current at the time of the writing of this manual, see **Appendix A, "Error Codes".**

Note that *all* error codes returned by the BusTools API are supported by this function, for the most recent list, see the current file "BUSAPI.H".

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

pString = BusTools_StatusGetString(wStatus);

pString	(char*) returned address of ASCII string (note string is actually a "static" string in the BusTools API routines -- just the address is returned -- do not attempt to modify this string).
wStatus	(BT_INT) BusTools API status code (from some routine).

Return Value

API_SUCCESS:
API_BUSTOOLS_BADCARDNUM:
API_BUSTOOLS_NOTINITED:

BusTools_TimeGetString

Description

This routine is used to convert a BusTools time structure into a string suitable for display to a user. The BusTools time structure is contained in all BM and RT messages. The output string has the format "h:mm:ss.ffffff", where:

- h = hours (as many digits as required).
- mm = minutes (2 digits in the range "00" to "59").
- ss = seconds (2 digits in the range "00" to "59").
- ffffff = fraction of a second -- microseconds (6 digits in the range "000000" to "999999").

All times used in the BusTools API are measured from when the BM simulation is turned on (using the **BusTools_BM_StartStop()** routine). BusTools time values have a microsecond resolution and have a range of over 150 days.

OS Support

DOS, 16-bit and 32-bit Windows

Calling Sequence

BusTools_TimeGetString(pTime, pString);

pTime	(BT1553_TIME*) far pointer to time structure
pString	(char*) far pointer to output string (should be at least 16 characters long).

Return Value

None

APPENDIX A

Error Codes

This appendix lists the BusTools API error codes and a description for each error. The errors are listed with both the numeric value and the mnemonic used for that error code. The numeric values are provided as a quick reference. However, when using the BusTools API routines, the mnemonics should be used, as the actual value assigned to each mnemonic may change across revisions of the software.

The function **BusTools_StatusGetString()** can be called at run-time to convert any of the following numeric error codes to an ASCII string which describes the error condition. Additionally, these codes are defined in the file BusAPI.H, which is supplied with the software.

All BusTools API routines return an error status. If that status is non-zero, an error has occurred. For convenience, the following mnemonic is defined for the successful return code:

API_SUCCESS: Routine executed successfully.

General Errors

API_BUSTOOLS_INITED: Specified BusTools board has already been initialized. **BusTools_API_Init()** can only be called once per board. Execute **BusTools_API_Close()** on the requested board before attempting to call this routine again.

API_BUSTOOLS_NOTINITED: Specified BusTools board has not been initialized. User must execute **BusTools_API_Init()** routine before executing this routine.

API_BUSTOOLS_BADCARDNUM: Specified "wCardnum" parameter is illegal and not supported. Legal values are in the range 0 - 3 (inclusive).

API_BUSTOOLS_BADINTTYPE: Illegal interrupt type specified. Legal values use the "API_INTTYPE_XXXX" mnemonics.

API_BUSTOOLS_BADCOUPLING: An illegal value has been specified for the "wCoupling" parameter. This parameter must be either "0" or "1".

API_BUSTOOLS_BADVOLTAGE: An illegal value has been specified for the "fVoltage" parameter. For direct coupling, this parameter must be in the range 0.0 to 6.5V. For transformer coupling, this parameter must be in the range 0.0 to 19.8V.

API_BUSTOOLS_BADSEGNUM: The routine **BusTools_MemoryAlloc()** can only operate in the SEG1 memory partition of the BusTools board. This error is generated when the caller asks for any other memory partition.

API_BUSTOOLS_EVENBCOUNT: **BusTools_MemoryAlloc()** can only allocate whole words of memory. Thus, the byte count specified for this routine must be even.

API_BUSTOOLS_BADMEMORY: The memory self-test failed.

API_BUSTOOLS_TOO_MANY: Too many user interrupt functions registered

API_BUSTOOLS_FIFO_BAD: User API_INT_FIFO structure is corrupted, contains a bad entry, or has a parameter out of range.

API_BUSTOOLS_NO_OBJECT: Error creating event object or thread.

API_HARDWARE_NOSUPPORT: The requested function is not supported by the installed hardware.

API_BAD_ADDR_TYPE: Illegal address type specified. The "wAddrType" parameter must use the GETADDR_XXXX mnemonics.

BC Errors

API_BC_NOTINITED: The BC has not been initialized. **BusTools_BC_Init()** must be executed prior to calling this routine.

API_BC_INITED: The BC has already been initialized.

API_BC_RUNNING: The BC is currently running. This routine cannot be called while the BC is running.

API_BC_NOTRUNNING: The BC is not currently running.

API_BC_MEMORY_OFLOW: There is insufficient memory on the BusTools board to allocate the specified number of messages.

API_BC_ILLEGAL_MBLOCK: The specified BC message number is illegal or exceeds the number of messages previously allocated.

API_BC_MBLOCK_NOMATCH: There is no BC message that begins on the specified address.

API_BC_MBUF_NOT_ALLOC: BC message buffers have not allocated. **BusTools_BC_MessageAlloc()** must be executed prior to calling this routine.

API_BC_MBUF_ALLOCD: BC message buffers have already been allocated with a smaller number of messages than specified in this call. **BusTools_BM_Init()** must be re-executed to re-initialize memory allocation and management.

API_BC_ILLEGAL_NEXT: The message number specified as the next BC message is illegal or exceeds the number of messages allocated.

API_BC_ILLEGAL_PREV: The message number specified as the previous BC message is illegal or exceeds the number of messages allocated.

API_BC_ILLEGAL_BRANCH: The message number specified as the branch BC message is illegal or exceeds the number of messages allocated.

API_BC_MESS1_COND: The first BC message in an array of messages cannot be a conditional message.

API_BC_BAD_COND_ADDR: Comparison address in BusTools hardware memory does not point to a location in the previous BC message.

API_BC_BADTIMEOUT1: The "wTimeout1" parameter is not in the proper range. It must be in the range 4 - 31 microseconds (inclusive).

API_BC_BADTIMEOUT2: The "wTimeout2" parameter is not in the proper range. It must be in the range 4 - 31 microseconds (inclusive).

API_BC_BADFREQUENCY: The "frame" parameter is not in the proper range. It must be in the range 2000 microseconds to 1600000 microseconds (inclusive).

API_BC_HALTERROR: Error detected when trying to halt the BC simulation. Typically, this occurs after a Minor Frame Overflow is detected. After this error is detected, the BusTools board may need a full reset (a new call to **BusTools_API_Init()** or **BusTools_BM_Init()**, depending upon the reason for the error).

API_BC_MCODE3_BUFFER: The BC Message Buffer control word specified the BC_CONTROL_BUFFERA or _BUFFERB bits. This function is not available with microcode revision 1.03 or earlier.

API_BC_MCODE3_BUS: The BC Message Buffer control word specified the BC_CONTROL_CHANNELA or _CHANNELB bits. This function is not available with microcode revision 1.03 or earlier.

API_BC_MCODE4_PINGPONG: The BC Message Buffer control word specified the BC_CONTROL_PINGPONG bit. This function is not available with microcode revision 1.04 or later.

API_BC_MCODE4_USEBUFFA: The BC Message Buffer control word specified the BC_CONTROL_USEBUFFA bit. This function is not available with microcode revision 1.04 or later.

API_BC_BOTHBUFFERS: The BC Message Buffer control word cannot select both buffers for operation. Only one of these control bits can be set (BC_CONTROL_BUFFERA or BC_CONTROL_BUFFERB).

API_BC_BOTHBUSES: The BC Message Buffer control word cannot select both channels for operation. Only one of the control bits can be set (either BC_CONTROL_CHANNELA or BC_CONTROL_CHANNELB).

API_BC_MCODE4_UPDATE: The **BusTools_BC_MessageUpdate()** routine is only available with microcode revision 1.04 or later.

API_BC_UPDATEMESSTYPE: The **BusTools_BC_MessageUpdate()** routine can only be used with messages of type BC_CONTROL_MESSAGE.

API_BC_ILLEGALMESSAGE: The BC message in memory is not legal.

API_BC_ILLEGALTARGET: The BC branch data message number is not legal.

API_BC_NOTMESSAGE: The BC message is not a proper 1553-type message..

API_BC_NOTNOOP: The BC message is not a proper no0p-type message, e.g., it was not first modified by a call to the API function **BusTools_BC_MessageNoop().**

API_BC_APERIODIC_RUNNING: The BC aperiodic message list is still running and cannot start a new aperiodic message list

API_BC_APERIODIC_TIMEOUT: BC aperiodic messages did not complete in specified amount of time

API_BC_CANT_NOOP: A BC NOOP message cannot be changed between a normal message and a NOOP message.

BM Errors

API_BM_NOTINITED: The BM has not been initialized. **BusTools_BM_Init()** must be executed prior to calling this routine.

API_BM_INITED: The BM has already been initialized.

API_BM_RUNNING: The BM is currently running. This routine cannot be called while the BM is running.

API_BM_NOTRUNNING: The BM is not currently running.

API_BM_MEMORY_OFLOW: The specified number of BM message blocks doesn't fit in available memory. Reduce the number of message blocks or reduce other memory usage and try again.

API_BM_CBUF_EXISTS: The BM subunit has already been defined.

API_BM_CBUF_NOTDEFINED: The BM subunit has not yet been defined.

API_BM_ILLEGAL_ADDR: An illegal RT address was specified in the "wRtaddr" parameter to this routine. RT addresses must be in the range 0 - 31 (inclusive).

API_BM_ILLEGAL_SUBADDR: An illegal RT subaddress was specified in the "wSubaddr" parameter to this routine. RT subaddresses must be in the range 0 - 31 (inclusive).

API_BM_ILLEGAL_TRANREC: An illegal value was specified in the "wTrans" parameter to this routine. Legal values are "0" and "1".

API_BM_ILLEGAL_MBUFID: The specified BM message number is illegal or exceeds the number of messages previously allocated.

API_BM_MBUF_NOMATCH: There is no BM message, which begins on the specified address.

API_BM_WRAP_AROUND: The number of BM messages received since the last call to **BusTools_BM_MessageReadBlock()** have exceeded the number BM message buffers allocated. Increasing the number of BM message buffers, decreasing the time between successive calls to the routine, or reducing the message rate in your model can solve the problem.

API_BM_ILLEGAL_MESSAGE: An illegal message type or data configuration was detected in the API_BM_MBUF_RAW structure passed to **BusTools_BM_MessageConvert()**. Check to see that the structure supplied has not been modified or corrupted since it was read from the BusTools hardware. Illegal message types include "Broadcast RT-to-BC", "RT-to-RT" with the first command being a transmit command, and "RT-to-RT" with mode code.

EI Errors

API_EI_BADMSGTYPE: An illegal error type was specified in a call to the **BusTools_EI_EbufWrite()** routine. Make sure all error types are of the form EI_XXXX.

API_EI_ILLERRORNO: The specified error injection buffer number is too large, since no such buffer exists.

API_EI_EILLERRORADDR: The address specified in the call to the **BusTools_EI_Getid()** routine is not a legal error injection buffer address.

RT Errors

API_RT_NOTINITED: The RT has not been initialized. **BusTools_RT_Init()** must be executed prior to calling this routine.

API_RT_INITED: The RT has already been initialized.

API_RT_RUNNING: The RT is currently running. This routine cannot be called while the RT is running.

API_RT_NOTRUNNING: The RT is not currently running.

API_RT_MEMORY_OFLOW: There is insufficient memory on the BusTools hardware for the specified operation. Either reduce the size of the current request or reduce memory usage elsewhere.

API_RT_SELF_TEST_MODE: RT Self Test Wrap-Around Mode selected, normal operation inhibited. The selection of the self-test mode will cause normal operation to be erratic, and should be used with caution.

API_RT_CBUF_EXISTS: The RT subunit has already been defined.

API_RT_CBUF_NOTDEFINED: The RT subunit has not been defined.

API_RT_ILLEGAL_ADDR: An illegal RT address was specified in the "wRtaddr" parameter to this routine. RT addresses must be in the range 0 - 31 (inclusive).

API_RT_ILLEGAL_SUBADDR: An illegal RT subaddress was specified in the "wSubaddr" parameter to this routine. RT subaddresses must be in the range 0 - 31 (inclusive).

API_RT_ILLEGAL_TRANREC: An illegal value was specified in the "wTrans" parameter to this routine. Legal values are "0" and "1".

API_RT_ILLEGAL_MBUFID: The specified RT message number is illegal or exceeds the number of messages previously allocated.

API_RT_TOOMANY_MBUFS: The specified number of message buffers (parameter "wCount") is larger on the second call to this routine than on the first call. The initial allocation of message buffers is the largest allocation allowed until the BusTools API is re-initialized.

API_RT_CBUF_BROAD: Specified "wRtaddr" is for broadcast only. Use the routine **BusTools_RT_CbufbroadRead()** or **BusTools_RT_CbufbroadWrite()**.

API_RT_CBUF_NOTBROAD: Specified "wRtaddr" is for non-broadcast only. Use the routine **BusTools_RT_CbufRead()** or **BusTools_RT_CbufWrite()**.

API_RT_MBUF_NOMATCH: There is no RT message buffer that begins on the specified address.

API_RT_BROADCAST_DIASBLE: RT 31 broadcast is disabled, calling BusTools_RT_CbufbroadRead() or BusTools_RT_CbufbroadWrite() is illegal.

LabView-Specific Errors

API_LV_BADARRAY: LabView array structure not correctly setup

API_NO_LV_SUPPORT: Function not supported in LabView

Low Level Errors

The error codes defined in this section are "low-level" error codes generated internally by the BusTools API. Typically, these errors are detected during the servicing of interrupts or the handling of direct communication with the hardware.

The error codes are logged by sending a "API_LOWERROR_MESSNO" message to the window assigned to BM recording (see the **BusTools_RecordRegister()** routine). One of the error codes listed below is transmitted in the "wParam" parameter in this message. It is up to that message handler to communicate the error condition to the user in some manner (typically generating a message window).

API_LOW_IQPTR_LOW: Interrupt queue address error. The interrupt queue address stored in the BusTools hardware memory does not point to a legal interrupt queue address. The value is below the interrupt queue location. Typically this indicates that the BusTools micro-code is not responding in a timely manner to requests to read memory. Contact Customer Service for assistance.

API_LOW_IQPTR_HIGH: Interrupt queue address error. The interrupt queue address stored in the BusTools hardware memory does not point to a legal interrupt queue address. The value is above the interrupt queue location. Typically this indicates that the BusTools micro-code is not responding in a timely manner to requests to read memory. Contact Customer Service for assistance.

API_LOW_IQPTR_ERROR: Interrupt queue address error. The interrupt queue address stored in the BusTools hardware memory does not point to a legal interrupt queue address. The value is within the normal range but does not point to the beginning of an interrupt queue block. Typically this indicates that the BusTools micro-code is not responding in a timely manner to requests to read memory. Contact Customer Service for assistance.

API_LOW_BM_WRAP: BM message buffer wrap-around. This indicates that the number of messages received since the last call to **BusTools_BM_MessageReadBlock()** exceeds the number of BM message buffers allocated in memory. Typically, the message rate must be reduced, the number of BM message buffers must be increased, or the routine must be called more often (or all of the above).

API_LOW_MINOR_OFLOW: Minor frame time overflow. The time required to complete a minor frame (as defined in the BC message list) is larger than the amount of time specified by the BC minor frame frequency. If this occurs, the 1553 bus may be left in an indeterminate state. Typically, the size of one or more minor frames must be reduced or the minor frame frequency must be reduced.

API_LOW_BC_HALTERROR: Error detected when trying to halt BC. Whenever possible, the BusTools API will halt the BC after it has completed processing an entire minor frame. The **BusTools_BC_StartStop()** routine waits for approximately one second for a gap between minor frames. If a gap is not detected during that time, it is assumed that the firmware or the 1553 bus interface is "hung". The BC is forced off (possibly leaving the 1553 bus in an indeterminate state) and this error is reported.

API_LOW_INT_OFLOW: Interrupt timeout or overflow. As interrupt conditions are detected by the BusTools hardware, the interrupt queue is updated with information indicating the type of interrupt condition and the location of the message buffer (either BC, BM or RT) in BusTools memory. The BusTools API software then passes this information to the application by sending a "API_CALLBACK_MESSNO" message to the window specified in the call to **BusTools_InterruptRegister()**. When the application has completed processing the interrupt information, it must call **BusTools_InterruptDone()** to indicate to the API that interrupt processing has been completed. The API keeps track of the number of interrupt conditions that have not yet been processed by the application. If this number gets too high, this error is reported and the rate at which interrupts are processed is reduced (meaning that the application may not be notified about all interrupts).

API_LOW_BM_REC_OFLOW: BM recording timeout or overflow. Periodically (approximately once every 500 milliseconds, or when there are 64K bytes of messages available), the BusTools API sends a "API_BMRECORD_MESSNO" message to the window specified in the call to **BusTools_RecordRegister()**. It is up to the application to manage the BM recording function (retrieving BM messages from the BusTools hardware and storing them in memory or on disk).

When the application has completed processing the recording function, it *must* call **BusTools_RecordDone()** to indicate to the API that processing has been completed. The application will not receive another record message until **BusTools_RecordDone**() has been called.

BusTools Driver Errors

BTD_OK: Same as API_SUCCESS, successful completion.

BTD_ERR_PARAM: Invalid parameter passed to low-level driver.

BTD_ERR_NOACCESS: Unable to access adapter.

BTD_ERR_INUSE: Adapter is already in use.

BTD_ERR_BADADDR: Invalid address.

BTD_ERR_NODETECT: Board detection failed; board not detected at specified physical address.

BTD_ERR_BADCFG: The specified board configuration is invalid or inconsistent.

BTD_ERR_NOTSETUP: Attempt to access an adapter, which has not been setup.

BTD_ERR_READFAIL: The low-level driver failed to read the specified region of the adapter.

BTD_ERR_WRITEFAIL: The low-level driver failed to write the specified region of the adapter.

BTD_ERR_NOMEMORY: Error occurred while allocating memory for use by the low-level driver.

BTD_ERR_BADDETECT: Could not read the board detect word.

BTD_ERR_BADADDRMAP: The low-level driver was not able to successfully map the adapter into host memory.

BTD_ERR_BADEXTMEM: The supplied extended memory mapping information is invalid.

BTD_ERR_BADBOARDTYPE: The adapter board type is unknown.

BTD_ERR_BADWCS: An error occurred while reading back the IP WCS.

BTD_NO_PCI_SERVICES: PCI services not available/not installed

BTD_NO_PCI_CARRIER: Cannot locate IP 1553 PCI carrier

BTD_NO_PLATFORM: Platform specified unknown or not supported

BTD_BAD_MANUFACTURER: IP ID PROM Manufacturer code not 0x79

BTD_BAD_MODEL: IP ID PROM Model number not 0x05

BTD_BAD_SERIAL_PROM: IP Serial PROM needs update, no support for this hardware version

BTD_ERR_NOWINRT: The low-level WinRT driver is not running.

BTD_ERR_BADREGISTER: The parameters in the Win NT or Win 95 registry are incorrect or conflict with another device.

BTD_ERR_BADOPEN: The WinRT low-level device driver open failed.

BTD_NO_SUPPORT: Bus/Carrier/OS combination not supported by API

Numeric Index

0	API_SUCCESS
201	API_BUSTOOLS_INITED
202	API_BUSTOOLS_NOTINITED
203	API_BUSTOOLS_BADCARDNUM
204	API_BUSTOOLS_BADINTTYPE
206	API_BUSTOOLS_BADCOUPLING
207	API_BUSTOOLS_BADVOLTAGE
208	API_BUSTOOLS_BADSEGNUM
209	API_BUSTOOLS_EVENBCOUNT
210	API_BUSTOOLS_BADMEMORY
211	API_BUSTOOLS_TOO_MANY
212	API_BUSTOOLS_FIFO_BAD
213	API_BUSTOOLS_NO_OBJECT
225	API_HARDWARE_NOSUPPORT
252	API_EI_BADMSGTYPE
253	API_EI_ILLERRORNO
254	API_EI_ILLERRORADDR
271	API_BAD_ADDR_TYPE
301	API_BC_NOTINITED
302	API_BC_INITED
303	API_BC_RUNNING
304	API_BC_NOTRUNNING
305	API_BC_MEMORY_OFLOW
306	API_BC_ILLEGAL_MBLOCK
307	API_BC_MBLOCK_NOMATCH
308	API_BC_MBUF_NOT_ALLOC
309	API_BC_MBUF_ALLOCD
310	API_BC_ILLEGAL_NEXT
311	API_BC_ILLEGAL_PREV
312	API_BC_ILLEGAL_BRANCH
313	API_BC_MESS1_COND
314	API_BC_BAD_COND_ADDR
315	API_BC_BADTIMEOUT1
316	API_BC_BADTIMEOUT2
317	API_BC_BADFREQUENCY
318	API_BC_HALTERROR
319	API_BC_MCODE3_BUFFER
320	API_BC_MCODE3_BUS
321	API_BC_MCODE4_PINGPONG
322	API_BC_MCODE4_USEBUFFA

323	API_BC_BOTHBUFFERS
324	API_BC_BOTHBUSES
325	API_BC_MCODE4_UPDATE
326	API_BC_UPDATEMESSTYPE
327	API_BC_ILLEGAL_TARGET
328	API_BC_ILLEGALTARGET
329	API_BC_NOTMESSAGE
330	API_BC_NOTNOOP
331	API_BC_APERIODIC_RUNNING
332	API_BC_APERIODIC_TIMEOUT
333	API_BC_CANT_NOOP
401	API_BM_NOTINITED
402	API_BM_INITED
403	API_BM_RUNNING
404	API_BM_NOTRUNNING
405	API_BM_MEMORY_OFLOW
406	API_BM_CBUF_EXISTS
407	API_BM_CBUF_NOTDEFINED
408	API_BM_ILLEGAL_ADDR
409	API_BM_ILLEGAL_SUBADDR
410	API_BM_ILLEGAL_TRANREC
411	API_BM_ILLEGAL_MBUFID
412	API_BM_MBUF_NOMATCH
413	API_BM_WRAP_AROUND
414	API_BM_ILLEGAL_MESSAGE
501	API_RT_NOTINITED
502	API_RT_INITED
503	API_RT_RUNNING
504	API_RT_NOTRUNNING
505	API_RT_MEMORY_OFLOW
506	API_RT_CBUF_EXISTS
507	API_RT_CBUF_NOTDEFINED
508	API_RT_ILLEGAL_ADDR
509	API_RT_ILLEGAL_SUBADDR
510	API_RT_ILLEGAL_TRANREC
511	API_RT_ILLEGAL_MBUFID
512	API_RT_TOOMANY_MBUFS
513	API_RT_CBUF_BROAD
514	API_RT_CBUF_NOTBROAD
515	API_RT_MBUF_NOMATCH
516	API_RT_BROADCAST_DISABLE
517	API_RT_SELF_TEST_MODE

601 API_LOW_IQPTR_LOW
602 API_LOW_IQPTR_HIGH
603 API_LOW_IQPTR_ERROR
604 API_LOW_BM_WRAP
605 API_LOW_MINOR_OFLOW
606 API_LOW_BC_HALTERROR
607 API_LOW_INT_OFLOW
608 API_LOW_BM_REC_OFLOW

700 API_LV_BADARRAY
701 API_NO_LV_SUPPORT

BusTools Driver (BTD) error return constants:

1 BTD_ERR_PARAM
2 BTD_ERR_NOACCESS
3 BTD_ERR_INUSE
4 BTD_ERR_BADADDR
5 BTD_ERR_NODETECT
6 BTD_ERR_BADCFG
7 BTD_ERR_NOTSETUP
8 BTD_ERR_READFAIL
9 BTD_ERR_WRITEFAIL
10 BTD_ERR_NOMEMORY
11 BTD_ERR_BADDETECT
12 BTD_ERR_BADADDRMAP
13 BTD_ERR_BADEXTMEM
14 BTD_ERR_BADBOARDTYPE
15 BTD_ERR_BADWCS
16 BTD_NO_PCI_SERVICES
17 BTD_NO_PCI_CARRIER
18 BTD_NO_PLATFORM
19 BTD_BAD_MANUFACTURER
20 BTD_BAD_MODEL
21 BTD_BAD_SERIAL_PROM

50 BTD_ERR_NOWINRT
51 BTD_ERR_BADREGISTER
52 BTD_ERR_BADOPEN
53 BTD_NO_SUPPORT

APPENDIX B

Data Structures

This sections describes the data structures and other definitions used throughout the BusTools API. For each structure or topic, a high level description is provided 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
- Interrupt Enable / Message Status Bits
- RT Address Control Block (API_RT_ABUF)
- RT Control Buffer (API_RT_CBUF)
- RT Control Buffer (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)

1553 Command Word (BT1553_COMMAND)

Code Definition

```
typedef struct bt1553_command
        {
        unsigned short wcount:5;        // word count or mode code field
        unsigned short subaddr:5;       // subaddress field
        unsigned short tran_rec:1;      // transmit/receive bit
        unsigned short rtaddr:5;        // rt address field
        }
BT1553_COMMAND;
```

Description

This structure is used to define a 1553 Command Word as it would appear on the bus. The various 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 modecode) field of the Command Word. It should be a value from 0 to 31. Note that, 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 means that the "wcount" field should be interpreted as a modecode.

tran_rec: This is the "transmit" / "receive" bit of the Command Word. It should be a value of 0 or 1. 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. Note that a value of 31 typically means that the command is a "broadcast" command.

1553 Status Word (BT1553_STATUS)

Code Definition

```
typedef struct bt1553_status
        {
        unsigned short tf:1;            // terminal flag bit
        unsigned short dba:1;           // dynamic bus acceptance flag bit
        unsigned short sf:1;            // subsystem flag bit
        unsigned short busy:1;          // busy flag bit
        unsigned short bcr:1;           // broadcast received bit
        unsigned short res:3;           // unused bits
        unsigned short sr:1;            // service request
        unsigned short inst:1;          // instrumentation bit
        unsigned short me:1;            // message error
        unsigned short rtaddr:5;        // rt address field
        }
BT1553_STATUS;
```

Description

This structure is used to define a 1553 Status Word as it would appear on the bus. The various 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.

BC Retry Parameters (BusTools_BC_Init() argument)

Code Definition

```
#define BC_RETRY_ALTB      0x0001  /* Register 78 Bit 9, Alternate Bus      */
#define BC_RETRY_NRSP      0x0002  /* Register 78 Bit 10, No Response       */
#define BC_RETRY_ME        0x0004  /* Register 45 Bit 10, Message Error     */
#define BC_RETRY_BUSY      0x0008  /* Register 45 Bit 3, Busy Bit Set       */
#define BC_RETRY_TF        0x0010  /* Register 45 Bit 0, Terminal Flag      */
#define BC_RETRY_SSF       0x0020  /* Register 45 Bit 2, SubSystem Flag     */
#define BC_RETRY_INSTR     0x0040  /* Register 45 Bit 9, Instrumentation    */
#define BC_RETRY_SRQ       0x0080  /* Register 45 Bit 8, Service            */
```

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 Message Buffer (API_BC_MBUF)

Code Definition

```
typedef struct api_bc_mbuf
    {
    WORD messno;                    // BC message number ("0" based)
    WORD control;                   // API message control word
    WORD messno_next;               // BC message number for next logical message,
                                    //  set to 0xFFFF if end of aperiodic message list.
    WORD messno_prev;               // Unused

// this section is for non-conditional BC messages

    BT1553_COMMAND mess_command1;   // 1553 command word
    BT1553_COMMAND mess_command2;   // command word #2 (for RT-RT msgs)
    WORD errorid;                   // error injection buffer id number
    WORD gap_time;                  // time to delay before next message
    BT1553_STATUS mess_status1;     // 1553 status word
    BT1553_STATUS mess_status2;     // 1553 status word #2 (for RT-RT msgs)
    DWORD status;                   // message status from hardware
    WORD data[2][BT1553_BUFCOUNT];  // 1553 data buffer A and B

// this section is for conditional messages

    WORD data_value;                // Data value to compare
    WORD data_mask;                 // Bit mask for compare
    WORD address;                   // Address for comparison or word number
    WORD messno_branch;             // If '=' branch to this message
    WORD messno_compare;            // Message number containing data word in "address"
    }
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. There are six types of BC message which can be defined by this structure, only one of which may be specified for any given message:

- a 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, the BC_CONTROL_MESSAGE bit of the "control" word must be set.
- a 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.
- a conditional branch message (#1): 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. Note that for IP-1553 microcode versions below 2.0, and

all PC-1553 microcode versions this function does not always work correctly. Depending on the previous message, the branch might be executed before the specified word has been received by the firmware, causing the branch to be incorrect.

- a conditional branch message (#2): This message transfers control to another message in the BC message list if a specified location in BusTools memory meets the specified conditions. To specify this type of message, the BC_CONTROL_CONDITION2 bit of the "control" word must be set.
- a conditional branch message (#3). 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. To avoid firmware problems which exist in all versions of the PC-1553, and versions of the IP-1553 before V2.00, do not specify the previous message as the source of the test word.
- a 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 being defined. It is composed of two parts: (1) a message type field and (2) message control bits. The first six entries below are message type definitions. 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 "address" parameter. Note that the specified location must be in the SEG1 memory partition.

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 to indicates that this BC message structure defines a standard BC message. It instructs the BC simulation to generate a message on the bus (either transmit or receive).

BC_CONTROL_NOP: This code indicates that this is a noop message (this message does nothing).

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()** routine). This bit is valid only for messages of type BC_CONTROL_MESSAGE.

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()** routine). This bit is valid only for messages of type BC_CONTROL_MESSAGE.

BC_CONTROL_CHANNELA: This bit is set to indicate that this message should be transmitted on Channel A. This bit is valid only for microcode revision 1.04 or later (see the **BusTools_GetRevision()** routine). This bit is valid only for messages of type BC_CONTROL_MESSAGE.

BC_CONTROL_CHANNELB: This bit is set to indicate that this message should be transmitted on Channel B. This bit is valid only for microcode revision 1.04 or later (see **BusTools_GetRevision()** routine). This bit is valid only for messages of type BC_CONTROL_MESSAGE.

BC_CONTROL_INTERRUPT: This bit is set if the BusTools hardware should 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 in the BusTools memory. A message will be sent to the Window message handler defined in the call to the **BusTools_InterruptRegister()** routine.

BC_CONTROL_LAST: This bit is set in the last message to be executed in the BC message list. If the BC list is designed to be executed once (a "oneshot" list), the last message should be a branch message with the destination address equal to the current message number. To repeat the BC list, the last message should be a branch to the first message in the BC message list. This bit cannot be set in a message of type BC_CONTROL_MESSAGE.

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. Note that the last message of a minor frame cannot be a conditional or branch message.

BC_CONTROL_PINGPONG: This bit must be set if the dual data buffer is to be used for this BC message. If this bit is set, then the BusTools hardware will use both data buffers defined in the message, alternating between the two each time the BC message list is executed. This bit is valid only if the BC_CONTROL_MESSAGE bit is set. This bit is valid only for microcode revision 1.03 or earlier (see the **BusTools_GetRevision()** routine).

BC_CONTROL_RETRY: This bit is set to indicate that retries are enabled for this message. This bit is valid only if the BC_CONTROL_MESSAGE bit is set.

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. On reading a BC message from the BusTools hardware, both "mess_status1" and "mess_status2" will be valid. This bit is valid only if the BC_CONTROL_MESSAGE bit is set.

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 only used if the BC_CONTROL_MESSAGE bit is set. Note that if Broadcast is enabled (see the **BusTools_SetBroadcast()** routine), 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 only used if the BC_CONTROL_MESSAGE bit is set. Note that an RT subaddress of "0" is automatically a mode code command. Also, if SA31 is enabled (see the **BusTools_SetSa31()** routine), 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 only used if the BC_CONTROL_MESSAGE bit is set.

mess_command1.wcount: This is the word count (or modecode, 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 only used by RT to RT messages, and 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 only used if the BC_CONTROL_MESSAGE bit is set. Note that if Broadcast is enabled (see the **BusTools_SetBroadcast()** routine), 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 only used if the BC_CONTROL_MESSAGE bit is set. Note that a RT subaddress of "0" is automatically a mode code command. Also, if SA31 is enabled (see the **BusTools_SetSa31()** routine), 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 only used if the BC_CONTROL_MESSAGE bit is set.

mess_command2.wcount: This is the word count (or modecode, 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 only used if the BC_CONTROL_MESSAGE bit is set.

gap_time: For the PC-1553 this is the time to delay after the completion of this message before starting the next message in the BC message list. For the IP-1553 this is the time from the mid-parity bit of the last word of the previous message to the mid-sync bit of the current message. The delay is specified in microseconds. For the PC-1553 it must be in the range 8 - 65535 microseconds for point-to-point messages. The IP-1553 supports a range of 5 – 65535 microseconds for point-to-point messages. This entry is only used if the BC_CONTROL_MESSAGE bit is set.

mess_status1: This is the 1553 status word returned by the RT as a result of executing this message. It is only valid after reading the BC message (using the **BusTools_BC_ReadMessage()** routine). This entry is only used if the BC_CONTROL_MESSAGE bit is set.

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 will be indeterminate. It is only valid after reading the BC message (using the **BusTools_BC_ReadMessage()** routine). This entry is only used if the BC_CONTROL_MESSAGE bit is set.

status: This is the 32-bit message status from the BusTools board. The definition of the individual bits in this parameter can be found elsewhere in this section ("Interrupt Enable/Status bits"). This entry is only used if the BC_CONTROL_MESSAGE bit is set.

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 the BusTools memory. On reading a message, both buffers are read from the BusTools memory. However, only the buffer indicated by the BT1553_INT_ALT_BUFFER bit of the "status" parameter contains the data which was actually on the 1553 bus.

data_value: This is the 16-bit data value to be used in the comparison in this conditional message. This entry is only used if the BC_CONTROL_CONDITION bit is set.

data_mask: This is the 16-bit mask applied to the bus data prior to the comparison for this conditional message. This entry is only used if the BC_CONTROL_CONDITION bit is 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 specifies an address in the SEG1 memory partition which will be used in the comparison. For type #1 messages, the 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.

messno_branch: This is the destination message number in the BC message list to be 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" will be executed next. This entry is only used if the BC_CONTROL_CONDITION bit is 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.

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
#define BM_FILTER_MC_SYNC           0x00000002L
#define BM_FILTER_MC_TSWS           0x00000004L
#define BM_FILTER_MC_STST           0x00000008L
#define BM_FILTER_MC_XSD            0x00000010L
#define BM_FILTER_MC_OXSD           0x00000020L
#define BM_FILTER_MC_ITF            0x00000040L
#define BM_FILTER_MC_OITF           0x00000080L
#define BM_FILTER_MC_RRT            0x00000100L

#define BM_FILTER_MC_TVWC           0x00010000L
#define BM_FILTER_MC_SWO            0x00020000L
#define BM_FILTER_MC_TLC            0x00040000L
#define BM_FILTER_MC_TBW            0x00080000L
#define BM_FILTER_MC_SXSD           0x00100000L
#define BM_FILTER_MC_OSXS           0x00200000L
```

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 BusTools memory.

The BM filter provides the ability to specify that only certain messages be recorded. The filter operates based on the command word of the message (which is a combination of the RT address, RT sub-address, transmit / receive flag and word count or modecode for the message) . When the command word is received, the BM filter determines if messages with that particular command word are to be recorded or ignored by the BM microcode. If the BM filter bit for the command word in a particular message is disabled, 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 particular command word is enabled, then the filter checks a filter counter for that command word. The counter is initialized by the application to indicate what fraction of BM messages with a particular command word should be recorded, where a counter set to "n" indicates that every "n"th 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 particular RT sub-unit (RT address, sub-address, and transmit / receive flag combination). If the specified bit is not set, then messages with the specified command word will be ignored by the BM simulation. If the specified bit is set, then messages will be recorded, subject to the specified counter. The application can independently specify the "BM filter" bits for each RT sub-unit using the **BusTools_BM_FilterWrite()** routine.

If the RT sub-address is a non-modecode sub-address, 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 particular word-count for the specified RT sub-unit. If the specified bit is "0", then a message with the specified RT address, sub-address and word count will be ignored by the BM simulation. If the specified bit is "1", then a message with the specified RT address, sub-address and word count will be stored in the BM Message Buffers by the BM simulation. 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 sub-address is a modecode sub-address, then the 32-bit "modecode" parameter is used. Bits are set in this parameter to enable / disable BM operations with the associated modecodes. The bit assignments for each modecode are provided in the "BM_FILTER_MC_XXX" parameters.

Data Elements

wcount: This is the 32-bit enable / disable flags for non-modecode sub-addresses. The bit assignments are described above. Note that this parameter is a union with "modecode". The BusTools hardware does not differentiate between modecode subaddresses and normal sub-addresses with respect to filter operations.

modecode: This is the 32-bit enable / disable flags for modecode sub-addresses. This bit assignments are specified in the "BM_FILTER_MC_XXX" parameters. Note that this parameter is a union with "wcount". The BusTools hardware does not differentiate

between modecode subaddresses and normal sub-addresses with respect to filter operations.

pass_count: This is the counter parameter used by the BM filter. If this value is set to "n", then every "n"th message with the specified command word will be recorded. If the application wants every message recorded, set this parameter to "1".

control_word: This parameter is not used at this time.

BM Message Buffer (API_BM_MBUF)

Code Definition

```
typedef struct api_bm_mbuf
        {
        DWORD                   messno;         // message number (filled in by API)
        DWORD                   int_status;     // interrupt status
        BT1553_TIME             time;           // time for message
        BT1553_COMMAND          command1;       // command word #1
        WORD                    status_c1;      // command word #1 status
        BT1553_COMMAND          command2;       // command word #2
        WORD                    status_c2;      // command word #2 status

        BT1553_BMRESPONSE       response1;      // response time #1
        BT1553_STATUS           status1;        // status word #1
        WORD                    status_s1;      // status word #1 status

        BT1553_BMRESPONSE       response2;      // response time #2
        BT1553_STATUS           status2;        // status word #2
        WORD                    status_s2;      // status for status word #2

        WORD    value[BT1553_MBUFCOUNT];        // data words
        BYTE    status[BT1553_MBUFCOUNT];       // status of data
        }
API_BM_MBUF;    // Note that this structure is exactly 128 bytes long!
```

Description

This structure contains all information provided by the BusTools 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 for the message are provided.

Errors information detected for the message is provided in two ways: (1) for each word appearing on the bus, a 16 bit error status is provided (see "BM Word Status Bits"), and (2) for the entire message, a 32 bit error status is provided (see "Interrupt Enable / Message Status Bits"). 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.

While the structure contains value and status information for up to 32 data words, only the number of data words specified in the command word are valid. The remaining value and status information is indeterminate.

Certain elements of this structure ("command2", "status_c2", "response2", "status2", and "status_s2") are only valid in the case of an "RT-to-RT" message. One 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 message number for this message. The message number is assigned by the API software, with the first message

monitored on the bus being message #1. Each time the BM simulation is re-started (using the **BusTools_BM_StartStop()** routine), the message number counting is re-started at "1".

int_status: This is the 32-bit message status from the BusTools board bits for this message (see "Interrupt Enable / Message Status Bits" elsewhere in this section).

time: This is the time at which this message was detected on the bus. It is measured from when the BM simulation was started (using the **BusTools_BM_StartStop()** routine). Each time the simulation 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 status word associated with the command word. For the bit definitions in this word, see "BM Word Status Bits" elsewhere in this section.

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, sub-address, 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 status word associated with the second command word. For the bit definitions in this word, see "BM Word Status Bits" elsewhere in this section.

response1: This is the time delay which 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 status word associated with the status word. For the bit definitions in this word, see "BM Word Status Bits" elsewhere in this section.

response2 (valid only for "RT-to-RT" messages): This is the time delay which 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 status word associated with the second status word. For the bit definitions in this word, see "BM Word Status Bits" elsewhere in this section.

value[i]: This is the "i"th data word as it appeared on the 1553 bus. Note that only "command1.wcount" data entries are valid in this message.

status[i]: This is the 8-bit error status associated with the "i"th data word. For the bit definitions in this word, see "BM Word Status Bits" elsewhere in this section.

BM Trigger Buffer (API_BM_TBUF)

Code Definition

```
/*****************************************************************************
* BM Trigger Buffer definition
*
*    control:    dialog box entries for order of events:
*                    START SIDE              STOP SIDE
*                    0 ->  unconditionally   never
*                    1 ->  if A              if C
*                    2 ->  if A AND B        if C AND D
*                    3 ->  if A OR B         if C OR D
*                    4 ->  if A then B       if C then D
*    type:       trigger type:
*                    0 ->  none
*                    1 ->  command
*                    2 ->  status
*                    3 ->  data
*                    4 ->  time (not working now)
*    mask:       16 bit mask (applied prior to compare)
*    value:      16 bit value to compare
*    word:       word number (only applicable for data entry)
*    time:       32 bit time (not working now)
*****************************************************************************/

typedef struct api_bm_tbuf
        {
        WORD    trig_ext;           // 1 -> external trigger enabled (rest is ignored)
        WORD    trig_err;           // 1 -> trigger on errors
        WORD    trig_ext_output;    // 1 -> external output on trigger
                                    // 2 -> repetitive external trigger output
        WORD control1;              // control of trigger start
        WORD control2;              // control of trigger stop
        struct
                {
                WORD type;
                WORD mask;
                WORD value;
                WORD word;
                WORD count;
                }
        capture[4];
        struct
                {
                WORD type;
                WORD mask;
                WORD value;
                WORD word;
                WORD count;
                }
        stop[4];
        }
API_BM_TBUF;
```

Description N/A

Data Elements N/A

BM Word Status Bits (8/16 bit)

Code Definition

```
#define BT1553_INT_HIGH_WORD            0x0001
#define BT1553_INT_BIT_COUNT_DATA       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_MID_SYNC             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_TWO_BUS              0x1000
#define BT1553_INT_WRONG_BUS            0x2000
#define BT1553_INT_BIT_COUNT            0x4000
#define BT1553_INT_NO_IMSG_GAP          0x8000
```

Description

The BM Word Status is a 16-bit parameter, with each bit indicating a different condition which can be detected and/or reported by the BusTools 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 each and 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.

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 will also set 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 will also set the BT1553_INT_INVALID_WORD bit.

BT1553_INT_CHANNEL: This bit indicates the channel on which the message was detected. If this bit is set, the message was detected on Channel B. If this bit is not set, the message was detected on Channel A.

BT1553_INT_EARLY_RESP: This bit indicates that the 1553 decoder detected a command sync with less than two microseconds 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 will also set 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()** routine.

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 150 nanoseconds of the expected time for any successive bits in any word of the message (except the sync bit -- see below). This condition will also set the BT1553_INT_INVALID_WORD bit.

BT1553_INT_MID_SYNC: This bit indicates that successive mid-zero crossings were not within 150 nanoseconds of the expected time for any mid-sync or end-sync bit time. This condition will also set the BT1553_INT_INVALID_WORD bit.

BT1553_INT_NO_IMSG_GAP: This bit indicates that the mid-sync zero crossing of the next command sync was detected prior to 4 microseconds 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 microsecond 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 will also set the BT1553_INT_INVALID_WORD bit.

BT1553_INT_TWO_BUS: This bit indicates that both buses (channel A and channel 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.

Error Injection Definition (API_EIBUF)

Code Definition

```
typedef struct api_eibuf
        {
        WORD buftype;                   // error injection buffer type
        struct
                {
                BYTE etype;             // error code (e.g. EI_NONE, EI_TWOBUS, etc.)
                BYTE edata;             // error data (if req'd)
                }
         error[EI_COUNT];
        }
API_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 errors are injected by the particular device (or simulation of a device) which is putting words out to the bus. Thus, in a typical message, some errors would be injected by the Bus Controller simulation (e.g. errors on the control word and any data transmitted by the BC) and some errors would be injected by the Remote Terminal simulation (e.g. errors on the status word and any data transmitted by the RT).

There are five types of Error Injection Buffers which have been defined. The different types are used in different situations as appropriate to the simulation. The types are:

- EI_BC_REC (BC Receive): This Error Injection Buffer type is used to define the errors to be injected by the BC simulation into a Receive Message. This buffer contains error specifications for 34 words which 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): This Error Injection Buffer type is used to define the errors to be injected by the BC simulation into a Transmit Message. Since the BC only generates 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): This Error Injection Buffer type is used to define the errors to be injected by an RT simulation 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): This Error Injection Buffer type is used to define the errors to be injected by an RT simulation into a Transmit Message. This buffer contains error specifications for 34

words which 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 an RT-to-RT message): This Error Injection Buffer type is used to define the errors to be injected by a BC simulation into a RT-to-RT Message. This buffer contains error specifications for 2 words which are transmitted by the BC in an 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).

error[i].etype: Error Injection specification for a single word to be transmitted onto the bus. The possible error specifications are as follows:

- EI_NONE (no error): This entry is used when there is to be no error injected for this particular word of the message.

- EI_BITCOUNT (bit count error): This entry is used to inject a bit count error into this particular word of the message. The hardware normally transmits a sync pulse followed by 16 data bits and ending with a parity bit for each word sent out on the 1553 bus. When this error is selected, the actual number of data bits to be transmitted in the word must be specified in the "error[i].data" 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 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.

- EI_SYNC (inverted sync error): This entry is used to inject an inverted sync error into this particular word of the message. This error can be specified for any entry in any EI buffer type.

- EI_PARITY (parity error): This entry is used to inject a parity error into this particular word of the message. This error can be specified for any entry in any EI buffer type.

- EI_BADADDR (respond with wrong address): Normally, the RT status word contains the address of the RT. This error will force the status word to contain a different address. The address to be used is specified in the "error[i].data" element of the structure. The address can be any number from 0 to 31. This error can be specified only for "RT Receive" and "RT Transmit" EI buffer types.

- EI_TWOBUS (respond on both buses): This error forces an RT to transmit a word (either status or data) on both channels, not just on the channel on which the command word was received. The hardware actually allows this error to be specified and injected on a word-by-word basis. However, this is not the intended use of this error condition. This error is intended to be a message-wide condition. When the BC sends a command word, the RT will respond with its entire response on both channels. Therefore, the user should ensure that if this error is desired, the error bit is set for every word of the Error Injection buffer. This error can be specified only for "RT Receive" and "RT Transmit" EI buffer types. This error has no meaning when set in any BC buffer type.

- EI_RESPWRONGBUS (response on wrong bus): This error forces an RT to transmit a word (either status or data) on the opposite channel from which the command word was received. The hardware actually allows this error to be specified and injected on a word-by-word basis. However, this is not the intended use of this error condition. This error is intended to be a message-wide condition. When the BC sends a command word on one channel, the RT will respond with its entire response on the opposite channel. Therefore, the user should ensure that if this error is desired, the error bit is set for every word of the Error Injection buffer. This error can be specified only for "RT Receive" and "RT Transmit" EI buffer types. This error has no meaning when set in any BC buffer type.

- EI_WORDCOUNT (word count error): Normally, the command word specifies the number of words which will be contained in a message. This error can force the actual number of words transmitted in the message to be different from that specified in the command word. The number of words to be sent is specified in the "error[i].data" entry of the structure. The hardware will accept any word count from 1 to 33 for this error. This error condition must be specified on the command word of a BC message (for "BC Receive" buffer types) or the status word of an RT message (for "RT Transmit buffer types).

error[i].data: If the error injection spec requires data, this is the data value.

Interrupt Enable / Message Status Bits (32 bit)

Code Definition

```
#define BT1553_INT_HIGH_WORD            0x00000001L
#define BT1553_INT_INVALID_WORD         0x00000002L
#define BT1553_INT_LOW_WORD             0x00000004L
#define BT1553_INT_INVERTED_SYNC        0x00000008L
#define BT1553_INT_MID_BIT              0x00000010L
#define BT1553_INT_MID_SYNC             0x00000020L
#define BT1553_INT_PARITY               0x00000040L
#define BT1553_INT_NON_CONT_DATA        0x00000080L
#define BT1553_INT_EARLY_RESP           0x00000100L
#define BT1553_INT_LATE_RESP            0x00000200L
#define BT1553_INT_BAD_RTADDR           0x00000400L
#define BT1553_INT_CHANNEL              0x00000800L
#define BT1553_INT_TWO_BUS              0x00001000L
#define BT1553_INT_WRONG_BUS            0x00002000L
#define BT1553_INT_BIT_COUNT            0x00004000L
#define BT1553_INT_NO_IMSG_GAP          0x00008000L

#define BT1553_INT_END_OF_MESS          0x00010000L
#define BT1553_INT_BROADCAST            0x00020000L
#define BT1553_INT_RT_RT_FORMAT         0x00040000L
#define BT1553_INT_RESET_RT             0x00080000L
#define BT1553_INT_SELF_TEST            0x00100000L
#define BT1553_INT_BIT_FAILURE          0x00200000L
#define BT1553_INT_INVALID_COMM         0x00400000L
#define BT1553_INT_RT_RT_TIMEOUT        0x00800000L
#define BT1553_NA1000000                0x01000000L
#define BT1553_INT_RETRY                0x02000000L
#define BT1553_INT_NO_RESP              0x04000000L
#define BT1553_INT_ERROR_BIT            0x08000000L
#define BT1553_INT_ALT_BUFFER           0x80000000L

// ***** SOFTWARE ONLY BITS *****
#define BT1553_INT_TRIG_BEGIN           0x10000000L
#define BT1553_INT_TRIG_END             0x20000000L
#define BT1553_INT_BM_OVERFLOW          0x40000000L
```

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 BusTools board. In the case of the Interrupt Enable parameter, each bit tells the BusTools 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 which was detected by the BusTools board.

There are three situations where the Interrupt Enable parameter is used: (1) BC Interrupt Enable (specified in the **BusTools_BC_Init()** routine and stored globally in one of the RAM registers), (2) BM Interrupt Enable (specified as one of the parameters in each BM message buffer), and (3) 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 which cannot 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: (1) the BC Message Buffer, (2) the BM Message Buffer, and (3) the 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 status bits set in the status word are not effected by the Interrupt Enable bits set separately. The status bits will be set for any and all conditions detected by the microcode for that message.

Data Elements (in alphabetical order)

BT1553_INT_ALT_BUFFER: This bit indicates that Buffer B (the alternate buffer of the BC message) was used for this message. If this bit is not set, Buffer A was used for this message.

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 will also set the BT1553_INT_INVALID_WORD bit.

BT1553_INT_BIT_FAILURE: This bit indicates that the RT ran a Built-In-Test and a failure was reported.

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 will be indicated 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 Channel B, otherwise the message was detected on Channel A.

BT1553_INT_EARLY_RESP: This bit indicates that the 1553 decoder detected a command sync with less than two microseconds 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_ERROR_BIT: This bit indicates that the status response had the message error bit set.

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_COMM: This bit indicates that an invalid 1553 command word was detected in the message. This is typically an invalid modecode command, or an illegal broadcast command.

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 will also set 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()** routine.

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 150 nanoseconds of the expected time for any successive bits in any word of the message (except the sync bit -- see below). This condition will also set the BT1553_INT_INVALID_WORD bit.

BT1553_INT_MID_SYNC: This bit indicates that successive mid-zero crossings were not within 150 nanoseconds of the expected time for any mid-sync or end-sync bit time in the message. This condition will also set the BT1553_INT_INVALID_WORD bit.

BT1553_INT_NO_IMSG_GAP: This bit indicates that the mid-sync zero crossing of the next command sync was detected prior to 4 microseconds 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()** routine.

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 microsecond 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 will also set the BT1553_INT_INVALID_WORD bit.

BT1553_INT_RESET_RT: This bit indicates that a valid "Reset Terminal" modecode command was received. The RT must reset to an initialized state.

BT1553_INT_RETRY: This bit indicates that an automatic retry was executed by the BC. This bit does not indicate whether the retry was successful. Failure of the retry results in the "No Response" error bit being set or in the "Message Error" bit in the RT's 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_INT_RT_RT_TIMEOUT: This bit indicates that the RT to RT message timed out.

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 will begin at this time. This bit is generated by the API interrupt service routine.

BT1553_INT_TRIG_END: This bit indicates that the BM trigger disable condition was met in this message. BM message gathering will terminate at this time. This bit is generated by the API interrupt service routine.

BT1553_INT_TWO_BUS: This bit indicates that both buses (channel A and channel 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.

Interrupt Register/Filter/FIFO Structure

Code Definition

```
#ifdef __WIN32__
#define MAX_FIFO_LEN        64 /* Size of the event FIFO (power of 2!) */
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 distroyed.
 // User-requested thread priority:
 int    iPriority;    // THREAD_PRIORITY_* as defined for Windows function
                // SetThreadPriority() or zero for normal priority.
 DWORD  dwMilliseconds;// Thread time-out interval in milliseconds 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 distruction of thread.  "OR" both
                // together to enable notification on both events.
 // 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   IThreadId;   // 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
      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.
    }
API_INT_FIFO;

#define CALL_STARTUP     0x0001  /* Thread created and initialized      */
#define CALL_EVENT       0x0002  /* Registered event occured            */
#define CALL_SHUTDOWN    0x0004  /* Thread shutdown has been requested  */

#define MAX_REGISTER_FUNCTION 128 /* Max number of registered functions  */
#endif

#define EVENT_TIMER_WRAP 0x0002 /* Timer has wrapped around (overflowed) */
#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_RECORDER   0x1000 /* BM recorder buffer has 64K or timeout */
#define EVENT_MF_OVERFLO 0x2000 /* Minor frame timing overflow           */
#define EVENT_API_OVERFLO 0x4000 /* API BM Recorder buffer overflowed    */
```

Description

This structure contains all of the information needed by the API to filter and deliver interrupt notifications to a user-specified function.

Use of this structure is only supported under 32-bit Windows.

Memory for this structure is allocated by the user, initialized and passed to the BusTools_RegisterFunction() API routine. The API creates a new thread and an event object, and links the new structure into its internal list of such structures.

Whenever an enabled event occurs, the API interrupt function scans each registered structure to determine if the associated thread should be signaled. If the event is enabled by the filter, 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's function, passing it the card number and a pointer to the registered structure. The user function can then perform whatever processing 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 around. 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 problems when updating the pointers.

For more information on the use of this structure see the **BusTools_RegisterFunction()** routine.

Data Elements

Further description to be supplied.

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 (read only)
        }
API_RT_ABUF;
```

Description

This structure contains control information which must be specified and can be changed for each RT defined on the BusTools board. The most important elements of this structure are the "enable" flags. These flags control whether or not the BusTools microcode will process 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 simulation should respond to messages on channel A. If this flag is off, the specified RT will ignore messages on channel A.

enable_b: This flag indicates whether the RT simulation should respond to messages on channel B. If this flag is off, the specified RT will ignore messages on channel B.

inhibit_term_flag: This flag controls the reporting of the "Terminal Flag" within the message status word. If this flag is set, then the "Terminal Flag" will not be set in the message status word. If this flag is not set, then the actual state of the "Terminal Flag" will be reported. This flag should be initialized by the application. Once the RT simulation 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.

status: This is the latest message status word. It should be initialized by the application. Once the RT simulation is operating, this entry is modified as required by the microcode. The microcode will modify 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.

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 sub-unit (a single RT address /sub-address combination for transmit or receive operations). There are two components of an RT Control Buffer: (1) a bit field enabling/disabling that RT sub-unit for each possible word count, and (2) a pointer to the first RT Message Buffer for this RT sub-unit. The bit field must be supplied by the application. The RT Message Buffer address is managed by the BusTools API routines (thus, it is not included in the RT Control Buffer structure definition).

This version of the RT Control Buffer is used to control normal RT sub-unit messages as well as RT modecode operations (sub-units with sub-address of "0" and optionally "31"). The structure is identical except that the bit assignments for word count 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 API_RT_CBUFBROAD).

Data Elements

legal_wordcount: This is a 32-bit value containing one bit for each possible word count for messages to/from this RT sub-unit. If the bit is set, then messages with the corresponding word count are allowed to/from this RT sub-unit. If the bit is not set, then messages with the corresponding word count are not allowed to/from this RT sub-unit. The bits in this value are assigned as follows (note that bit 0 is the low order or least significant bit, bit 31 is the high order bit):

- Bit 0 controls operations for a message with 32 words (which is actually a word count entry of "0").
- Bit 1 controls operations for a message with 1 word.
- Bit 2 controls operations for a message with 2 words.
- .....
- Bit 30 controls operations for a message with 30 words.
- Bit 31 controls operations for a message with 31 words.

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
        }
API_RT_CBUFBROAD;
```

Description

The RT Control Buffer for Broadcast is a structure that controls the operation of all RT address and word count combinations for a broadcast command to a specified subaddress. There are two components of an RT Control Buffer for Broadcast: (1) an array of bit fields which enable and/or disable broadcast operations to the specified RT sub-address, and (2) a pointer to the first RT Message Buffer used for broadcasting to this RT sub-address. The array of bit fields must be supplied by the application. The RT Message Buffer address is managed by the BusTools API routines (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 BusTools memory. However, RT sub-addresses which share a common set of enable/disable parameters can share a common Control Buffer. The BusTools API routines define a default Control Buffer with all enable bits set. Initially, all sub-addresses 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 0 controls operations for a broadcast message with 32 words (which is actually a word count entry of "0").
- Bit 1 controls operations for a broadcast message with 1 word.
- Bit 2 controls operations for a broadcast message with 2 words.
- ••••••
- Bit 30 controls operations for a broadcast message with 30 words.
- Bit 31 controls operations for a broadcast message with 31 words.

RT Message Buffer (read-only) (API_RT_MBUF_READ)

Code Definition

```
typedef struct api_rt_mbuf_read
        {
        DWORD           status;         // interrupt status
        BT1553_TIMETAG  time_tag;       // time tag
        BT1553_TIME     time;           // time

        // contents of real message to/from bus

        BT1553_COMMAND  mess_command;   // command word
        BT1553_STATUS   mess_status;    // status
        WORD            mess_data[32];  // data
        BT1553_STATUS   mess_status2;   // status word (RT-to-RT only)
        }
API_RT_MBUF_READ;
```

Description

This structure contains all information which can be read from an RT Message Buffer. All elements of this structure (except for data values) contain valid information only after a message has been processed for this RT sub-unit. For Message Buffers associated with "transmit" RT sub-units, the data values are those originally supplied by the application. For Message Buffers associated with "receive" RT sub-units, 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 the section describing "Interrupt Enable / Message Status Bits" elsewhere in this chapter.

time_tag: This is the raw time-tag information supplied by the BusTools micro-processor indicating the time at which this message was transmitted / received. This time-tag contains a counter indicating the number of 50 nanosecond clock cycles which have occurred since the start of the BusTools simulation. However, since this value can "wrap-around" in under 4 minutes, it is typically not an accurate measure of elapse time. Use the "time" entry (defined below) to get the actual elapsed time.

time: This is the time at which this message was transmitted / received. The time is measured from the start of the BusTools simulation (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.

mess_status2: This is the second status word transmitted / received in this command. It is only valid for RT-to-RT commands.

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 an RT Message Buffer. For "transmit" messages, all contents of this structure should be supplied. For "receive" messages, only the interrupt enable bits and the error injection pointer need be initialized.

Data Elements

enable: This is 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 will be added to the Interrupt Queue. For bit definitions, see the "Interrupt Enable / Message Status Bits" section elsewhere in this chapter.

error_inj_id: This is the Error Injection Buffer number to be used with this message. For "transmit" messages, this should be a buffer which contains an entry for the status word and each possible data word for this message. For "receive" messages, this should be a buffer which contains an entry for only the status word.

mess_data (only required for "transmit" messages): This is the actual data to be transmitted the command. The application is responsible for ensuring a match between the number of data elements supplied here and the word count which actually is requested by the BC during runtime.

Time Structure (BT1553_TIME)

Code Definition

```
typedef struct bt1553_time
        {
        unsigned long microseconds;        // microseconds since start of simulation
        unsigned short topuseconds;        // Most significant part of microseconds
        }
BT1553_TIME; // Note that this is a 48-bit value!
```

Description

This structure is used to describe time within the BusTools API. Time is measured from the beginning of the simulation in microseconds. For time calculations purposes, the simulation is considered started when the Bus Monitor is started.

Data Elements

microseconds: This is the number of microseconds since the simulation was started.

topuseconds: The most-significant portion of the number of microseconds since the start of the simulation.

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP