

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.

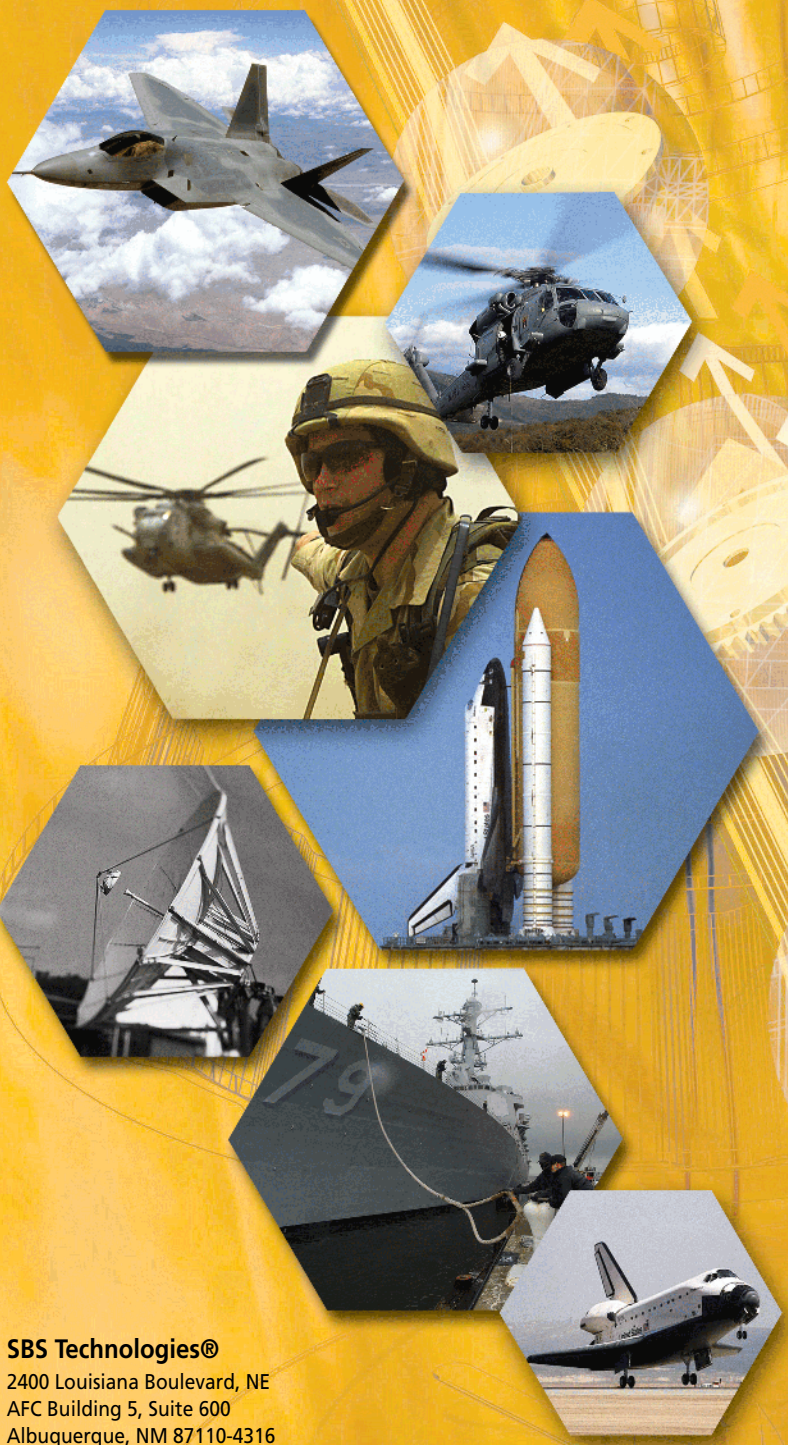
- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

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

Visit our website - *Click HERE*

MIL-STD-1553 ABI V-5 User's Manual



SBS Technologies®
2400 Louisiana Boulevard, NE
AFC Building 5, Suite 600
Albuquerque, NM 87110-4316
Sales: 800-SBS-1553 or 505-875-0600
Fax: 505-875-0400
Tech Support: 877-832-4727
Tech Support Email: support.government@sbs.com
Documentation Support Email: govdocs@sbs.com
http://www.sbs.com
http://www.resource.sbs.com

Doc PN: 501-553039-00

© 2003 SBS Technologies, Incorporated. All rights reserved.

MIL-STD-1533 ABI V-5 User's Manual

This document is the intellectual property of SBS Technologies, Inc. (SBS), and contains proprietary and confidential information. Use, disclosure, and reproduction is permitted only under the terms of an SBS software license agreement or explicit written permission of SBS. You are not authorized to use this document or its contents until you have read and agreed to the applicable software license agreement.

THIS DOCUMENT AND ITS CONTENTS ARE PROVIDED AS IS, WITH NO WARRANTIES OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF DESIGN, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM ANY COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

All computer code and software contained in this document is licensed to be used only in connection with an SBS product. Even if this code or software is merged with any other code or software program, it remains subject to the terms and conditions of this license. If you copy, or merge, this code or software, you must reproduce and include all SBS copyright notices and any other proprietary rights notices.

In no event will SBS be liable for any lost revenue or profits or other special, indirect, incidental and consequential damage, even if SBS has been advised of the possibility of such damages, as a result of the usage of this document and the software that this document describes. The entire liability of SBS shall be limited to the amount paid by you for this document and its contents.

SBS shall have no liability with respect to the infringement of copyrights, trade secrets, or any patents by this document of any part thereof. Please see the applicable software license agreement for full disclaimer or warranties and limitations of liability.

RESTRICTED RIGHTS LEGEND

Use, duplication, reproduction, release, performance, display or disclosure by the Government is subject to restrictions set forth in subparagraph (b)(3) of the Rights in Technical Data and Computer Software clause at 48 CFR 252.227-7013.

SBS Technologies, Inc., 2400 Louisiana Blvd. NE #5-600, Albuquerque, NM 87110

This manual, as well as the software described herein, is furnished under license and may only be used or copied in accordance with the terms of such license.

SBS Technologies and the SBS logo are trademarks of SBS Technologies, Inc. All other brand names and product names contained herein are trademarks, registered trademarks, or trade names of their respective holders.

Table of Contents

1: Introduction 1-1

Manual Organization 1-1
- Introduction 1-1
- Hardware Installation 1-1
- 1553 Operations 1-2
  - Module Overview 1-2
  - Module Startup/Test 1-2
  - Host Commands/Control Registers 1-2
  - BC, RT, Sequential Monitor, and Interrupts 1-2

Terminology and Symbols 1-3
- Terminology 1-3
- Symbols 1-6

Module Specifications 1-6
- General 1-6
- BC Simulation 1-7
- RT Simulation 1-7
- Monitoring 1-7
- Front Panel Connectors 1-8
- Power Consumption 1-8
- Customer Support Service 1-9
- FTP Site 1-9

2: ABI-V5 Hardware Installation 2-1

Unpacking and Inspecting the Equipment 2-1

VMEbus Characteristics 2-2
- V5 IRIG Hardware Configuration 2-2

Setting the ABI-V5 RAM Base Address in VMEbus Memory 2-3

Optional External Connections 2-5
- 1553 Signals and External Timing Controls5 2-5
- VMEbus Connections 2-5
- J1/J2 (J2/J3 for IRIG) Connections 2-8
- J3 (J4for IRIG) Connections 2-10
- Jumper Blocks JB1–JB4 2-11

Installing the ABI-V5 in a VMEbus System 2-12

Connecting the ABI-V5 to a 1553 Bus 2-12

Power Application 2-13

Testing the Board 2-15

3: 1553 Operations—Module Overview 3-1

Introduction 3-1
Module Overview 3-1
Module Startup/Test 3-1
Host Commands/Control Registers 3-1
BC Simulation 3-1
RT Simulation/RT Monitoring 3-2
Sequential Monitoring 3-2
Interrupt Processing 3-2
Architecture Overview 3-3
Design Review 3-4
Block (a): VMEbus Interface 3-5
Block (b): Memory Arbiter and Main Memory 3-6
Block (c): RISC Processor 3-6
Block (d): Local I/O Bus 3-6
Block (e): DSP 1553 Daughterboards 3-7
Block (f): Timer 3-7
48-Bit Timer for Standard V5 3-7
IRIG-B Timer for V5 IRIG 3-7
Block (g): 8-Bit I/O - External Interrupt (P2 Connections) 3-8
Memory Organization..... 3-9
Application Memory Organization 3-10
View Product Codes 3-10

4: Module Startup/Test 4-1

ABI-V5 Initialization Steps 4-1
ABI-V5 Firmware Download Instructions..... 4-2
Download Data Regions 4-2
Firmware Download and Startup Procedure 4-3
Download File Example 4-4
Built-In Tests..... 4-6
Performing All Tests 4-6
Performing Selected Tests 4-7
Module Operation 4-8
Firmware Configuration Register 4-8

5: Host Commands/Control Registers 5-1

Hardware Control Registers..... 5-2
Control/Status Register 5-3
Host DMA Registers 5-6
ABI ADDL, ABI ADDH 5-8
VME ADDL, VME ADDH 5-8
WORD CNT 5-8
INTV_DMA 5-8

Example: Steps to Initiate a Host DMA Transfer 5-9
Example: C Code for a Host DMA Transfer 5-9
Standard V5 Timing Hardware Registers 5-11
Internal Timer Control Register (ITCR) 5-11
Internal Timer Data Registers ITDRL, ITDRM, ITDRH 5-13
V5 IRIG Timing Hardware Registers 5-14
Internal Timer Control Register (ITCR) 5-14
Internal Timer Data Registers ITDRL, ITDRH ...5-14
Software Control Registers (1553 Processing Control)..... 5-16
RESP(RO) Register 5-19
DATABASE (RT, BC Data Buffer Base Address Control Register) 5-19
DBCTRL: Database Control Register 5-20
Host DMA Request Registers 5-20
HDMACTRL 5-20
HDMAABI 5-20
HDMAVME 5-20
HDMAWCV 5-22
Illegal Mode Codes 5-22
Illegal Receive Mode Code Mask (IRCVMD), Illegal Transmit Mode Code Mask (ITRXMD) 5-22
XDATPOL: External Data Polarity 5-22
BCXTSEL: BC External Trigger Out Select 5-23
Error Table..... 5-23
Last Error Message Buffer..... 5-25
RT Control and Definition Tables..... 5-26
Host Commands 5-26
Setting Host Command (With or Without Parameters) 5-27
Host Command Descriptions 5-28
Initialize Channel 5-28
Enable Channel 5-28
Disable Channel 5-28
Stop BC Chain 5-28
Start BC Chain 5-28
Update Global BC Control Information 5-29
Update RT Status Response Table Entry 5-29
Sequential Monitor Buffers 5-29
Host Processing Interrupt Queue 5-30
Change Data Base 5-30
1553 Stub Select 5-30
Host DMA Request 5-31
Update Illegal Mode Codes Host Command ..5-31

6: BC Simulation 6-1

Operational Steps for BC Simulation 6-1
Software Control Registers..... 6-2
BCCPTR 6-2
BCLPTR 6-2

BCIMGP 6-2
BCRETRY 6-3
RSPGPA 6-3
FRMTMR 6-4
BCSMSK 6-4
BCSCHDLY 6-5
BC Control Blocks 6-5
NEXT Pointer,
PREV Pointer 6-6
Schedule Time, Delay Time, or Schedule Delay 6-6
Transfer Types 6-7
Reserved 6-9
DMA CNTL, ABI ADDR, VME ADDR, WORD CNT/VECTOR 6-9
Receive RT Status/Transmit RT Status 6-9
Error Flags/Packet Type 6-10
Standard V5 TimeH,
TimeM/TimeL 6-11
V5 IRIG
TimeH
TimeL 6-12
Intermessage Gap 6-13
New Database 6-13
BC Control Word 6-14
Reserved 6-17
Transfer Type/Error Injection Control 6-17
Command 1/Command 2 6-18
Current Data Buffer Pointer/Control 6-18
Bits 0-23
Data Buffer Pointer/
Offset 6-18
Data Buffer Description 6-19

7: RT Simulation/RT Monitoring 7-1

RT Structure Overview..... 7-1
RT Status Response Table 7-2
RT Definition Table 7-2
RT Definition Pointer Table 7-2
Operational Steps for RT Simulation and RT Monitoring 7-2
RT Status Response Table 7-3
Half-Word (1) 7-3
Half-Word (2) 7-3
Half-Word (3) 7-3
Half-Word (4) 7-3
Half-Word Descriptions 7-5
(1) RT Global Control 7-5
(2) RT Status Transmit Control 7-6
(3) RT Status Response Word 7-6
RT Definition Table 7-9
Words in an RT Definition Table Entry 7-10
Reserved 7-10
RCV SA SM Filter,
TRX SA SM Filter 7-10
RCV SA SM

Swap on Msg,
TRX SA SM
Swap on Msg 7-11
RCV SA Int on Msg Received,
TRX SA Int on Msg Received 7-11
Mode Code Interrupt Mask 7-11
Last Status
Word 7-11
Vector Word (MON) 7-12
Synchronize with Data (MON) 7-12
Last Command (MON) 7-12
Bit Word (MON) 7-12
Selected TX Shutdown (MON) 7-12
Ovr. Selected TX Shutdown (MON) 7-12
Vector Word (TRX) 7-12
Reserved 7-12
Bit Word (TRX) 7-12
Reserved 7-13
Command Counter 7-13
RT Simulate/RT Monitor Summary 7-14
Simulate Only 7-14
Monitor Only 7-14
Simulate and Monitor 7-14
Illegal Subaddress 7-14
Update RT Illegal Subaddress Host Command 7-16
Control Buffer Pointers/Offsets 7-17
Monitor All BPT/Subaddresses Pointer/Offset 7-17
Monitor RCV SA Pointers/Offsets,
Monitor TRX SA Pointers/
Offsets 7-18
TRX SA Pointers/Offsets 7-18
Control Buffers 7-19
NEXT Pointer, PREV Pointer 7-19
Buffer Control 7-20
DMA CNTL,
ABI ADDR,
VME ADDR, WORD CNT/VECTOR 7-20
Standard V5 TimeH, TimeM, Time L 7-20
V5 IRIG
TimeH
Time L 7-20
Error Injection Control 7-21
Data Buffer Pointer 7-21
Data Buffer Options 7-22
Data Buffers 7-24
MONCTRL 7-24
Receive Data Buffers 7-24
Packet Type Word 7-26
Last Error Message Buffer 7-26
TimeH, TimeM, and TimeL 7-26
Monitor Buffer Offsets 1, 2 7-26
Gap Time 7-26
Complete 1553 Message Data 7-26
Last Error Message Buffer 7-27

Transmit Data Buffers 7-29
Data words 1 through n 7-30

8: 1553 Bus Monitoring 8-1

Sequential Monitoring 8-1
Mapped Monitoring 8-1
Sequential Monitoring 8-2
Operational Steps for Sequential Monitoring 8-3
Setting Sequential Monitor Filter and Swap Bits 8-4
RCV SA SM Filter, TRX SA SM Filter 8-4
RCV SA SM Swap on Msg, TRX SA SM Swap on Msg 8-4
Sequential Monitor Control Registers 8-6
SMFPTR 8-6
SMCPTR 8-6
SMLPTR 8-6
SMBCNT 8-6
MONCTRL 8-6
Sequential Monitor Buffers 8-7
NEXT Pointer, PREV Pointer 8-7
Buffer Control/Length 8-9
Number of Words 8-9
Number of Messages 8-9
Buffer Counter 8-10
DMA CNTL, ABI ADDR, VME ADDR, WORD CNT/VECTOR 8-10
Message Block Format (Updated by Firmware) 8-10
Type Word 8-10
TimeH, TimeM, TimeL 8-11
Message Block Offsets 1, 2 8-11
Gap Time 8-13
Data Words 8-13
Messages Received with Errors 8-13
Minimum Programming Requirements for Sequential Monitor Buffers 8-15
Buffer Swap Detection 8-16
Forcing Buffer Swaps 8-16
Resetting Sequential Monitoring 8-17
Mapped Monitoring 8-18
Mapped Monitoring Data Structures 8-20

9: Interrupts 9-1

Interrupt Queue Data Structures and Control Registers 9-2
Interrupt Bit Field 9-2
Time High 9-4
Time Low 9-4

IRIG Time 9-4
Interrupt Queue Offsets 1, 2 9-4
CMD2, CMD1 9-4
CMD1 RT Control Buffer Pointer 9-4
CMD2 RT Control Buffer Pointer 9-4
BC Control Block Pointer 9-4
SM Buffer Pointer 9-4
Database 9-4
Interrupt Types 9-4
Minimum Programming Requirements for Interrupt Queues 9-7
Hardware Interrupt Service Procedure..... 9-8
Software Polling Interrupt Service Procedure 9-9

1: Introduction

The SBS Technologies Advanced Bus Interface (ABI-V5) for VMEbus backplanes is an advanced, high density surface mount, multiprocessor design that provides one or two dual redundant MIL-STD-1553 interfaces to a VMEbus system. Each of the 1553 channels is capable of independent and concurrent simulation of BC and 31 RTs with dual function monitoring. The ABI-V5 also features a high speed VMEbus interface with D32/D64 slave and master capabilities.

1.1 Manual Organization

This section outlines the organization of this manual, which describes the 1553 processing capabilities of the ABI-V5.



Cross Reference: An accompanying manual, *MIL-STD-1553 Interface Libraries for the ABI-V5*, describes the C and Ada software libraries and device drivers provided with the module.

1.1.1 Introduction

This chapter, which you are now reading, provides general information about the ABI-V5, such as system specifications, mating connector information, and power requirements.

1.1.2 Hardware Installation

This chapter provides instructions for configuring and installing the ABI-V5 module in a VMEbus system. It includes instructions for setting the VMEbus address in A24 or A32 space, clock jumper selections, 1553 P2 signal routing selections, front panel pinouts, installation procedures for adding/removing 1553 processor daughter boards, and a 1553 bus connection/configuration overview.



Cross Reference: VMEbus interrupt vectors and levels are set by the host software; programming details are found in the *1553 Operations* section.

1.1.3 1553 Operations

This section has six chapters which detail the programming of the card, including hardware and software control registers, and BC, RT, and Monitor data structures. All the chapters except for "Module Overview" have identification tabs ("Module Overview" begins immediately after the large "1553 Operations" tab).

Module Overview This chapter explains the Architecture and Memory Organization of the module.

Module Startup/Test This chapter details how to load and boot-up the module. It also includes instructions and interpretations for the module's Built-in Tests (BITs).

Host Commands/ Control Registers This chapter reviews the host command set that controls major software operations. It also details the hardware and software control registers.

BC, RT, Sequential Monitor, and Interrupts These chapters detail the data structures and software operations for each respective function.

1.2 Terminology and Symbols

Terminology Table 1.2.1, defines some of the terms used throughout this manual.

Table 1.2.1: Basic Terminology

Term	Meaning
<CR>	Carriage return
ABI	Advanced bus interface
ADW	Associated Data Word (associated with mode commands)
Application Mode	Mode of operation for 1553 device operations. This mode is entered upon startup of the firmware.
BC	Bus controller
BC control chain	A series of BC control blocks in a linked list.
BIT	Built-in test
BM	Bus monitor
Built-in Test Mode	Mode of operation from which BITs can be run. The board is in this mode only after the firmware has been loaded, but before the firmware has been started.
Bus	A single 1553 bus connection (i.e., Bus A or Bus B).
Byte	8 bits
Chain	A series of blocks in a linked list.
Channel	One complete, dual-redundant 1553 bus interface.
Circular chain	A linked list of data buffers in which the last data buffer in the list points back to the first data buffer in the list.
Control Register	A predefined location in ABI/ASF memory that contains a value which the firmware uses for 1553 operations. Control registers define parameters for firmware structures and contain pointer values to the location of these structures.
CSR	I/O control and status register
Device	A logical entity that corresponds one-for-one with a 1553 channel and a device entry in the *sbs_dev.cfg* configuration file.
DLL	Dynamic link library
DMA	Direct memory access (same as VMEbus block)
DSP	Digital signal processor
Dual-Redundant	Includes both a primary and a secondary connection (i.e., Bus A and Bus B make up a dual-redundant bus).
Firmware	Program running in the ABI/ASF digital signal processors that controls all 1553 operations. The firmware must be loaded upon device initialization.

Term	Meaning
Firmware Structure	A table (or tables) defined in ABI/ASF memory containing values that the firmware uses for 1553 operations.
h	Hexadecimal
Hardware Control Register	A 16-bit register that controls key setup information for the SBS device. On PC platforms it is accessed through the I/O port; on PCI and VME platforms, it is stored in device RAM.
I/O	Input/output
IRIG	Inter range instrumentation group
IRQ	Interrupt request
ISR	Interrupt service routine
Link	A single block, in a series of blocks, that is pointed to by the previous block and has a pointer to the next block.
Longword	32 bits (double word)
LSW	Least significant word
Major Frame	A group of minor frames, usually determined by the slowest data rate in the system.
Mapped monitor	A BM function that provides fixed memory locations to store RT SA data in linked buffers. This function is ideal for real-time or current value applications such as real-time processing or system screen type displays.
Minor Frame	An increment of time usually determined by the fastest data rate in the system.
MSW	Most significant word
PC	Personal computer
RAM	Random access memory
Rcv	Receive
RISC	Reduced Instruction Set Computer
RO	Read Only register access
RT	Remote Terminal
RTOS	Real-time operating system
SA	Subaddress
Sequential monitor	A BM function that stores time tagged messages in programmable lengthy memory buffers. This function multiplexes all selected bus messages to large double buffered data structures and is ideal for archiving or storage applications.
Software Control Register	A 16-bit register that controls 1553 programming functions stored in device RAM.
T/R	Transmit/receive
TTL	Transistor-transistor logic

Term	Meaning
VME	VERSAmodule Eurocard
WO	Write only register access
Word	A 16-bit value; i.e., two bytes.
WR	Write/read
Xmt	Transmit
µ	Micro

All addresses are given in hexadecimal byte addressing. All words and long-words have bit 0 as the right-most bit. The ABI-V5 is a big endian machine.



Note: The Hardware Control Registers (located in memory below 40h) are accessed as 16-bit words. All other addresses, offsets, and data values described in this document refer to 32-bit longwords.

Symbols The following symbols appear throughout this manual:



Warning: Paragraphs next to this symbol contain information critical to module operation.



Note: Paragraphs next to this symbol contain information important to module operation.



Tip: Paragraphs next to this symbol contain useful tips.



Cross Reference: Paragraphs next to this symbol contain cross references to the *ABI/ASF User's Manual* or to a related page in this document

1.3 Module Specifications

1.3.1 General

- One or two dual redundant MIL-STD-1553 channels
- Each channel capable of concurrent BC and 31-RTs simulation, and dual function monitoring
- Full error injection and detection
- Capable of correlating and processing data between channels
- D32/D64 master/slave at 40+ MB, D16/D32 single cycle, A24/A32 addressing
- Three programmable interrupt vectors
- 8-bit I/O port for triggers
- 48-bit system clock: internal 1 microsecond or external TTL/RS-422
- 512 KB–2 MB SRAM models
- 128 KB instruction SRAM; third-party compilers available
- Programmable RISC/DSP and control logic
- Software selectable long or short stub connections
- Programmable, multicolor LEDs
- Shock requirements are to the highest commercial standards
- Operating Temperature: 0˚ to 60˚ Celsius, at less than or equal to 95% humidity

- Storage Temperature: –65˚ to 150˚ Celsius, at less than or equal to 95% humidity
- Mean Time Between Failures: >50,000 hours, based on MIL-HDBK-217F

1.3.2 BC Simulation

- Independent minor frame timers
- Doubly-linked-list BC block and transmit list control
- External triggers, auto retry

1.3.3 RT Simulation

- Simultaneous 31 remote terminals
- Double linked-list message control
- Full mode code control
- Programmable response time at RT level

1.3.4 Monitoring

- Chronological bus monitor with 48-bit time stamped messages. Filter any RT address or subaddress combination.
- Independent, real-time map monitoring with 48-bit time stamped, linked-list messages at RT address or subaddress level.

1.3.5 Front Panel Connectors

Details on the three front panel connectors, located at J1, J2, and J3, are shown in Table 1.3.1. A cable assembly is included with the ABI-V5 for connection to Bus A and Bus B. The assembly is available from the company listed below under part number CA-2078.

MilesTek
1 Lake Trail Drive
Argyle, Texas 76226
Attn: Frank or Jeanette Miles

800-524-7444
800-683-7444

Also,
940-484-9400 or FAX 940-484-9402

Table 1.3.1: Front Panel and Mating Connectors

Connectors	Type	Part Number	Manufacturer
Front Panel Connectors	HD-22 Receptacle, 26-pin	748481-6	AMP
Mating Connectors	HD-22 Plug, 26-pin	748365-1	AMP

1.3.6 Power Consumption

Single Channel

- +5 V at 2.6 amps
- +12 V at 250 milliamps

Dual Channel

- +5 V at 2.8 amps
- +12 V at 500 milliamps

1.3.7 Customer Support Service

SBS Technologies is dedicated to providing technically superior products and the best customer support possible. Full support for the ABI-V5 product is provided, including any reasonable assistance with the entire integration effort. For assistance, please contact SBS Technologies via phone, fax, regular mail, or e-mail.

SBS Technologies, Inc.
2400 Louisiana Boulevard, NE, Building 5, Suite 600
Albuquerque, New Mexico 87110
1-800-SBS-1553 • 505-875-0600
FAX (505) 875-0400
e-mail: *sbshelp@sbs1553.com*

1.3.8 FTP Site

SBS has an FTP site from which files may be retrieved. The site is

ftp.ni.net

and the SBS home directory is

cust/sbstech

The login name for this site is *anonymous* and the password is your internet name, i.e., *user@somewhere.com*. This is the only combination of user login name and password that will work on this site.

From the home directory there are directories for each major device in the SBS family of products, such as ABI-V5, ABI-V6, ABI-PC3, etc. Under each of these directories, you will find subdirectories for firmware, interface libraries source code, drivers, and other useful files.

2: ABI-V5 Hardware Installation

The ABI-V5 is a single board, 1553 interface for single or dual VMEbus systems. This chapter describes unpacking, configuring, and installing an ABI-V5 into a VMEbus system. Because proper installation is critical to successful operation of the ABI-V5, SBS Technologies recommends that the steps described here be followed explicitly. The steps involved in the installation include:

- Unpacking and inspecting the equipment
- Setting the ABI-V5 RAM base address in the VMEbus memory
- Reviewing the System Timer Configuration
- Selecting long- or short-stub connection
- Installing the configured board in the VMEbus system
- Testing the installed board

This manual chapter describes the above steps in detail.



Warning: This is an electronic product sensitive to electrostatic discharge. Observe normal precautions in handling the board to prevent damage.

2.1 Unpacking and Inspecting the Equipment

The ABI-V5 shipment should include the ABI-V5 board, hardware and software manuals, a DOS disk or TAR tape containing the ABI-V5 Application Code, and a DOS disk or TAR tape containing the Interface Libraries. Contact SBS Technologies at 1-800-SBS-1553 if any of these items is missing.

Carefully unpack the ABI-V5 board and inspect it for any visible damage that could have occurred during shipping. If there is damage, contact SBS Technologies with the board and board serial number available.

Figure 2.2.1 and Table 2.2.1 show a layout and description of the switch and jumper block positions on the ABI-V5 board. These switches and jumper blocks are detailed following the illustrations.

2.2 VMEbus Characteristics

The ABI-V5 board supports both A32 and A24 VMEbus address modes. The desired address mode is switch-configured. Both Supervisor and Nonprivileged address modes are supported. The ABI-V5 board supports slave data transfers in D64, D32, and D16 modes. Single transfers are supported in all D32 and D16 modes, and block transfers are supported in either D64 or D32. All Interrupt vectors are type D08.

The ABI-V5 is also capable of being a VME master device. While in this mode, D64 and D32 block transfers are supported. VME master also supports Supervisor and Nonprivileged transfers, in either A24 or A32 address mode.

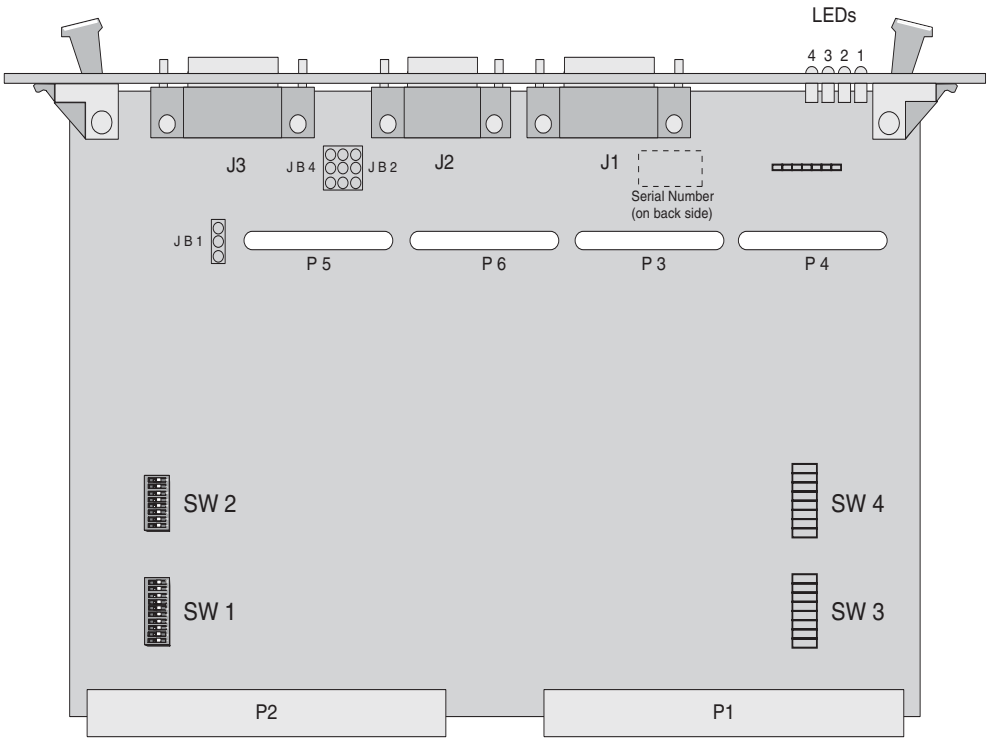


Figure 2.2.1: ABI-V5 Board Layout

Test blocks (TBs) are for factory test purposes only and are omitted from Figure 2.2.1.

V5 IRIG Hardware Configuration

An ABI-V5 with the IRIG time option contains just two jumper blocks (JB2 and JB3) and three switches (SW1, SW2, and SW3). Figure 2.2.1 shows the layout of the V5 board, and Table 2.2.1 describes the jumper blocks, connectors, and switches.

Table 2.2.1: Jumper Blocks, Connectors, and Switches

V5 STD Jumper Block/ Connector	V5 IRIG Jumper Block/ Connector	Description	Default Jumper Setting
JB1	N/A	External Clock Type Configuration	None
JB2	JB2	External Data 1 Type Configuration	None
JB3	JB3	External Data 2 Type Configuration	None
JB4	N/A	External Clock Clear Type Selection	None
P1, P2	P1, P2	VMEbus Interface	None
P3, P4	P3, P4	Transceiver Daughterboard #1 Connectors	None
P5, P6	P5, P6	Transceiver Daughterboard #2 Connectors	None
J1	J2	1553 Bus #1 (Primary and Secondary)	None
J2	J3	1553 Bus #2 (Primary and Secondary)	None
J3	N/A	External Connector. Assorted signals for external input/output connections	None
N/A	J4	Auxiliary Connector. Assorted signals for external input/output connections	None
J4	J1	System Timer Download Test Connector	None
SW1, SW2	SW1, SW3	Base Address Configuration	4800h (see Table 2.3.1)
SW3, SW4	SW2	Optional Connections for 1553 and external clock	None (see Table 2.4.1)

2.3 Setting the ABI-V5 RAM Base Address in VMEbus Memory

The base ABI-V5 appears as 512 KB or 2 MB (depending on module options) of RAM mapped into the local address space as either A32 or A24 addressable memory.

1. Set the desired offset address of this memory in the local VMEbus system to map the ABI-V5 RAM to a 512K word boundary.

2. Select address type A32 or A24 and select a base address offset by modifying SIP switches SW1 and SW2.

Figure 2.2.1 shows the locations of the jumper blocks and SIP switches on the ABI-V5 board. Set a switch to "on" to set the value of the corresponding address bit to "0"; set a switch to "off" to set the value of the corresponding address bit to "1". Table 2.3.1 and Figure 2.3.1 illustrate an offset base address of 4800 0000h and a selection of A32 address mode (the default setting of the board when shipped).

Table 2.3.1: Base Address Selection

Address Bit	Desired Value	Switch Location	Switch Setting
A31	0	SW1-1	ON
A30	1	SW1-2	OFF
A29	0	SW1-3	ON
A28	0	SW1-4	ON
A27	1	SW1-5	OFF
A26	0	SW1-6	ON
A25	0	SW1-7	ON
A24	0	SW1-8	ON
A23	0	SW2-1	ON
A22	0	SW2-2	ON
A21	0	SW2-3	ON
A20	0	SW2-4	ON
A19	0	SW2-5	ON
A18	0	SW2-6	ON
A17	0	SW2-7	ON
A32/A24 Select	0**	SW2-8	ON

Notes:
** A32 address mode is selected by setting this switch ON; A24 address mode is selected by setting this switch OFF.

When using A24 addressing, the positions of the switches in SW1 are "Don't Care" values.

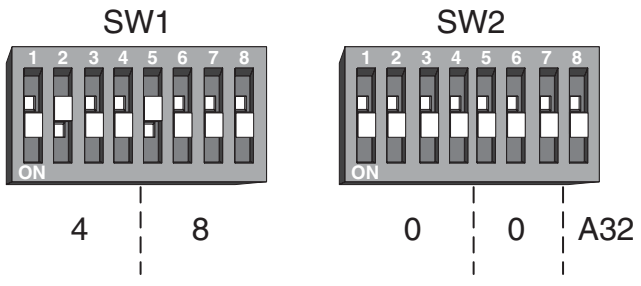


Figure 2.3.1: Switch Positions for Default Base Address

Determine the base address to be used for the ABI-V5 and set SW1 and SW2 accordingly.

2.4 Optional External Connections

2.4.1 1553 Signals and External Timing Controls5

Switches SW3 and SW4 (SW2 for IRIG) allow for 1553 signals and external timing controls to be connected to user I/O pins on the VME P2 connector. The board is shipped with these signals disconnected. Table 2.4.1 provides the signal description with the switch number and corresponding VME P2 pin designation.

Table 2.4.1: P2 Connections

STD V5 Switch #	V5 IRIG Switch #	Description	Default Switch Setting	P2 pin #
SW3-8	N/A	External Clock In +	OFF	C8
SW3-7	N/A	External Clock In –	OFF	C9
SW3-6	N/A	External Clock Clear In +	OFF	C10
SW3-5	N/A	External Clock Clear In –	OFF	C11
SW3-4	N/A	External Clock Out +	OFF	C14
SW3-3	N/A	External Clock Out –	OFF	C15
SW3-2	N/A	External Clock Clear Out +	OFF	C16
SW3-1	N/A	External Clock Clear Out –	OFF	C17
SW4-10	SW2-10	Bus 1 / Primary +	OFF	A8
SW4-9	SW2-9	Bus 1 / Primary –	OFF	A9
SW4-8	SW2-8	Bus 1 / Secondary +	OFF	A10
SW4-7	SW2-7	Bus 1 / Secondary –	OFF	A11
SW4-6	SW2-6	Bus 2 / Primary +	OFF	A14
SW4-5	SW2-5	Bus 2 / Primary –	OFF	A15
SW4-4	SW2-4	Bus 2 / Secondary +	OFF	A16
SW4-3	SW2-3	Bus 2 / Secondary –	OFF	A17
SW4-2	SW2-2	External Clock Enable	OFF	C20
SW4-1	SW2-1	Reserved	–	–

2.4.2 VMEbus Connections

Table 2.4.2 and Table 2.4.3 provide the VME connections to the ABI-V5 as well as the available SW3 and SW4 connections. (The SW3 and SW4 connections are shaded in gray in Table 2.4.3.)

Table 2.4.2: VMEbus P1 Connections

P1 Pin #	Row A	Row B	Row C
1	D00	BBSY*	D08
2	D01	BCLR*	D09
3	D02	N/C	D10
4	D03	BGIN0*	D11
5	D04	BGOUT0*	D12
6	D05	BGIN1*	D13
7	D06	BGOUT1*	D14
8	D07	BGIN2*	D15
9	GND	BGOUT2*	GND
1	N/C	BGIN3*	N/C
11	GND	BGOUT3*	BERR*
12	DS1*	BR0*	RESET*
13	DS0*	BR1*	LWORD*
14	WR*	BR2*	AM5
15	GND	BR3*	A23
16	DTACK*	AM0	A22
17	GND	AM1	A21
18	AS*	AM2	A20
19	GND	AM3	A19
20	IACK*	GND	A18
21	IACKIN*	N/C	A17
22	IACKOUT*	N/C	A16
23	AM4	GND	A15
24	A07	IRQ7*	A14
25	A06	IRQ6*	A13
26	A05	IRQ5*	A12
27	A04	IRQ4*	A11
28	A03	IRQ3*	A10
29	A02	IRQ2*	A09
30	A01	IRQ1*	A08
31	N/C	N/C	+12 VDC
32	+5 VDC	+5 VDC	+5 VDC

Table 2.4.3: Standard V5 VMEbus P2 Connections (SW3 and SW4 connections shaded in gray)

P2 Pin #	Row A	Row B	Row C
1	N/C	+5 VDC	N/C
2	N/C	GND	N/C
3	N/C	N/C	N/C
4	N/C	A24	N/C
5	N/C	A25	N/C
6	N/C	A26	N/C
7	N/C	A27	N/C
8	BUS 1 PRI +	A28	EXT CLK IN+
9	BUS 1 PRI–	A29	EXT CLK IN–
10	BUS 1 SEC +	A30	EXT CLR IN+
11	BUS 1 SEC–	A31	EXT CLR IN–
12	N/C	GND	N/C
13	N/C	+5 VDC	N/C
14	BUS 2 PRI +	D16	EXT CLK OUT+
15	BUS 2 PRI–	D17	EXT CLK OUT–
16	BUS 2 SEC +	D18	EXT CLR OUT+
17	BUS 2 SEC–	D19	EXT CLR OUT–
18	N/C	D20	N/C
19	N/C	D21	N/C
20	N/C	D22	EXT CLK ENA
21	N/C	D23	N/C
22	N/C	GND	N/C
23	N/C	D24	N/C
24	N/C	D25	N/C
25	N/C	D26	N/C
26	N/C	D27	N/C
27	N/C	D28	N/C
28	N/C	D29	N/C
29	N/C	D30	N/C
30	N/C	D31	N/C
31	N/C	GND	N/C
32	N/C	+5 VDC	N/C

Table 2.4.4: V5 IRIG VMEbus P2 Connections (SW2 Connections are Shaded in Gray)

P2-PIN #	Row A	Row B	Row C
1	N/C	+5 VDC	N/C
1	N/C	GND	N/C
3	N/C	N/C	N/C
4	N/C	A24	N/C
5	N/C	A25	N/C
6	N/C	A26	N/C
7	N/C	A27	N/C
8	BUS 1 PRI +	A28	N/C
9	BUS 1 PRI –	A29	N/C
10	BUS 1 SEC +	A30	N/C
11	BUS 1 SEC –	A31	N/C
12	N/C	GND	N/C
13	N/C	+5 VDC	N/C
14	BUS 2 PRI +	D16	N/C
15	BUS 2 PRI –	D17	N/C
16	BUS 2 SEC +	D18	N/C
17	BUS 2 SEC –	D19	N/C
18	N/C	D20	N/C
19	N/C	D21	N/C
20	N/C	D22	EXT CLK ENA
21	N/C	D23	N/C
22	N/C	GND	N/C
23	N/C	D24	N/C
24	N/C	D25	N/C
25	N/C	D26	N/C
26	N/C	D27	N/C
27	N/C	D28	N/C
28	N/C	D29	N/C
29	N/C	D30	N/C
30	N/C	D31	N/C
31	N/C	GND	N/C
32	N/C	+5 VDC	N/C

2.4.3 J1/J2 (J2/J3 for IRIG) Connections

Table 2.4.5 provides the pinout for the J1 and J2 (J2 and J3 for IRIG) connectors. J1 (J2 for IRIG) is for channel 1 and J2 (J3 for IRIG) is for channel 2. The

29K interrupt, an active low signal, is shared by both connectors on the Standard V5 and is not applicable to the V5 IRIG. All reserved pins are for future use and should remain open.

Table 2.4.5: J1/J2 (J2/J3 for IRIG) Connections

Pin #	Description
1	Primary Bus +
2	Primary Bus –
3	Secondary Bus +
4	Secondary Bus –
5	Reserved
6	Reserved
7	Reserved
8	Reserved
9	Reserved
10	Reserved
11	Reserved
12	Reserved
13	External Trigger In
14	Reserved
15	Reserved
16	Reserved
17	+5 VDC
18	GND
19	
20	
21	
22	29K Interrupt (Standard V5 ONLY)
23	
24	
25	
26	

2.4.4 J3 (J4for IRIG) Connections

The external clock for the V5 Standard may be connected to either the VME P2 connector or to the auxiliary connector, J3 (J4 for IRIG), on the front panel. However, only one of these connections should be made at a given time. Table 2.4.6 lists the pin assignments for the auxiliary connector. Ten pins are reserved for external data signals to be used for future implementation of special user requirements.

Table 2.4.6: J3 (J4 for IRIG) Connections

Pin #	V5 Standard Description	V5 IRIG Description
1	External Clock In +	
2	Reserved (External Data 1+)	Reserved (External Data 1+)
3	External Clock In –	
4	Reserved (External Data 1–)	Reserved (External Data 1–)
5	External Clock Clear +	IRIG In
6	Reserved (External Data 2+)	Reserved (External Data 2+)
7	External Clock Clear In –	
8	Reserved (External Data 2–)	Reserved (External Data 2–)
9	External Clock Out +	
10	Reserved (External Data 3)	Reserved (External Data 3)
11	External Clock Out –	GROUND
12	Reserved (External Data 4)	Reserved (External Data 4)
13	External Clock Clear Out +	
14	Reserved (External Data 5)	Reserved (External Data 5)
15	External Clock Clear Out –	
16	Reserved (External Data 6)	Reserved (External Data 6)
17		
18	Reserved (External Data 7)	Reserved (External Data 7)
19		
20	Reserved (External Data 8)	Reserved (External Data 8)
21		
22		
23		
24		
25	External Clock Enable	External INT *

Pin #	V5 Standard Description	V5 IRIG Description
26	GROUND	GROUND

2.4.5 Jumper Blocks JB1–JB4

Jumper blocks JB1–JB4 allow for the selection of single-ended or differential external data signals. A single-ended signal consists of a single TTL line. A differential signal consists of two TTL lines being driven with opposite levels, as shown in Table 2.4.7.

Table 2.4.7: TTL Lines in Differential Signal

TTL 1	TTL 2
+ = high	+ = low
– = low	– = high
Signal = high	Signal = low

Jumper blocks JB1–JB4 affect four external signals: External Clock In, External Data 1, External Data 2, and External Clock Clear In. Table 2.4.8 lists each jumper block with its associated data signal and available options. (Pin 1 for each jumper block is designated by a square.)

Table 2.4.8: Jumper Block Selections

JB #	Signal	Single-Ended (Place Jumper across Pins:)	Differential (Place Jumper across Pins:)
JB1	External Clock In	1–2	2–3
JB2	External Data 1	1–2	2–3
JB3	External Data 2	1–2	2–3
JB4	External Clock Clear In	1–2	2–3



Cross Reference: External Clock In and External Clock Clear In are explained in the section *Standard V5 Timing Hardware Registers* on 5-11.

There are a total of eight External Data lines available on the ABI-V5 (see Table 2.4.6). External Data 1 and 2 may be configured to output either single-ended or differential signals using JB2 and JB3. The remaining External Data lines are single-ended only. Software control register BCXTSEL (see Table 5.2.1 on page 16) is used to select the desired External Data line.

2.5 Installing the ABI-V5 in a VMEbus System

Install the ABI-V5 board into the desired slot in the host system. Follow the instructions supplied by the system manufacturer.

2.6 Connecting the ABI-V5 to a 1553 Bus

For operation, the ABI-V5 must be connected to a 1553 bus. Make these 1553 bus connections very carefully. *The ABI-V5 will not operate properly if the bus connections are not correct*. (For test purposes it is sufficient to simply connect standard bus terminators to the A and B bus pigtails on each channel's cable adapter.)

The 1553 bus consists of a twisted-pair cable, ranging in length from a few inches to several hundred feet (maximum of 100 meters, 330 feet is recommended.) The physical connection of a subsystem to this bus is called a stub. The configuration used for the connection is referred to as long-stub coupling and requires a transformer and two resistors, used at the exact point where the connection is made to the bus. These components are typically enclosed in a small module called a bus coupler. These couplers may provide for connection of only one stub or several stubs. The coupler has two connectors for the bus (usually at opposite ends of the coupler) and a connector for each stub (typically mounted perpendicular to the bus connectors).

Subsystem stubs will not function properly if connected directly together. Proper coupling devices must be used for each stub and the bus must be properly terminated. Figure 2.6.1 and Figure 2.6.2 illustrate the proper and improper connection schemes, respectively, for long-stub coupling for one channel and the terminology used for the various components associated with the bus. A three-stub coupler is shown in Figure 2.6.2. In this case, the bus is contained entirely within the coupler. If other devices are to be connected to the bus, one of the terminators must be removed and replaced with a cable connecting this coupler to another coupler. The other coupler could then be terminated or the bus extended further.

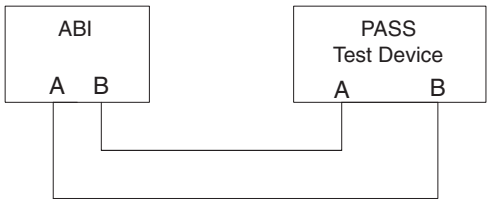


Figure 2.6.1: Incorrect Method of Bus Coupling (Long Stubs)

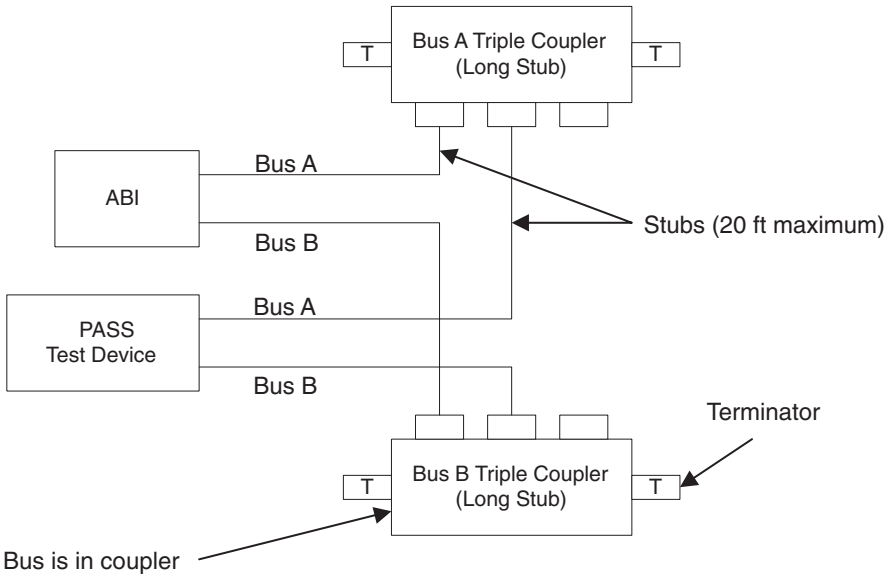


Figure 2.6.2: Correct Method of Bus Coupling (Long Stubs)

2.7 Power Application

Power up the host computer system to apply power to the ABI-V5. Figure 2.7.1 shows four LEDs visible on the ABI-V5's front panel. The LED labeled PROG indicates proper hardware initialization of the board and should be green immediately following power-up. The TEST LED should be red, and the CH1 and CH2 LEDs should be off. The PROG LED is also a user programmable LED. If the PROG LED is not green, contact SBS Technologies.



Cross Reference: See the following section, *Testing the Board*, on 2-15 for a description of the PROG LED.

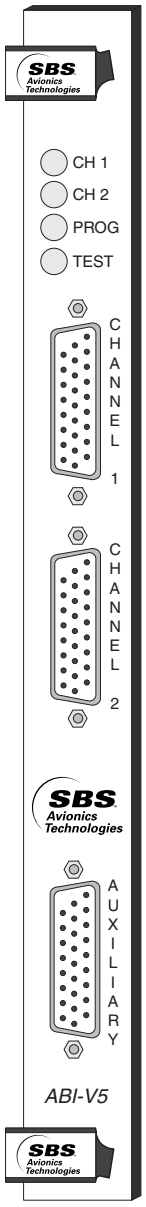


Figure 2.7.1: ABI-V5 Front Panel

2.8 Testing the Board

When the PROG LED is green, the ABI-V5 can be accessed by the host at the address selected during configuration. Determine a suitable method for accessing the board from host software. The method used depends on the operating system and the language used for programming.

Once the method has been determined, write a program to test the board. SBS supplies diagnostic functions which may be used to test the operation of the board.



Cross Reference: *Chapter 4: Module Startup/Test* discusses the procedures to be followed to utilize these diagnostic functions.

There is a simple test that may be performed (prior to the complex diagnostic test series) to determine if the ABI-V5 is being properly accessed. To perform this test, complete the following steps:

1. Turn the PROG LED green by writing a 0800h into the Control/Status register (offset 0000h from the VMEbus base address).
2. Turn the PROG LED red by loading a 0400h into the Control/Status register.

This test is to be used only during initial testing of the board.



Warning: The Control/Status register is used by the ABI-V5 firmware. Writing to this register during actual operation of the board could cause the ABI-V5 to malfunction.

3: 1553 Operations—Module Overview

3.1 Introduction

The multichannel Advanced Bus Interface, version five (ABI-V5), provides concurrent BC and 31-RTs simulation, dual function monitoring, advanced interrupt services, and high-speed VMEbus operations. This multiprocessor design incorporates the latest in RISC and DSP processor technologies to provide the most flexible, open designed 1553 module in the industry.

This manual section provides detailed programming instructions for 1553 and VMEbus DMA operations of the ABI-V5. The section is divided into six chapters which describe module operations. Read the first three chapters, *Chapter 3: 1553 Operations—Module Overview*, *Chapter 4: Module Startup/Test*, and *Chapter 5: Host Commands/Control Registers*, before attempting to operate the ABI-V5. Read the remaining chapters, *Chapter 6: BC Simulation*, *Chapter 7: RT Simulation/RT Monitoring*, and *Chapter 8: 1553 Bus Monitoring*, as required. They are each stand-alone chapters containing programming information specific to one ABI-V5 function.

Module Overview The chapter you are reading now, it reviews the hardware architecture of the ABI-V5, highlighting the major features and operation of the module.

Module Startup/Test This chapter details the startup requirements of the module which include downloading the instruction/program data and booting-up the processor. This chapter also includes instructions and interpretations for Built-In Tests (BITs).

Host Commands/ Control Registers This chapter details the processes (host commands) and control registers which allow the host application program to control hardware and data structures for 1553 processing. The host command set syntax and key software control registers for managing 1553 processing are described, and hardware registers that are directly accessible from the VMEbus (i.e., CSR, 48-bit clock, and VMEbus host registers for DMA control) are detailed in this chapter.

BC Simulation This chapter details the data structures associated with simulating BC messages. It explains the definition and management of BC control blocks and their associated data buffers. Control blocks are linked to allow for flexible and accurate BC simulation.

RT Simulation/ RT Monitoring

The ABI-V5 has a sophisticated data structures for simulation control and real-time monitoring of 1553 traffic at the RT address and subaddress levels. This chapter details how to set up status word responses, transmit buffers and mode code responses, and well as individual monitor buffers (the ABI-V5 can monitor its own RT or BC transmissions in real-time buffers). These monitor buffers are ideal for real-time host processing, user or GUI display tasks, or for logging and archiving individual RT monitor buffers. All transmit and monitor buffers are time-stamped with a 48-bit, 1-µsec clock value.

Sequential Monitoring

This chapter details the second type of monitoring available on the ABI-V5 - Sequential Monitoring. This independent monitoring feature provides correlation of 1553 messages filtered to the RT subaddress level (messages are time stamped with a 48-bit value). The host's application program can define the length of individual monitor buffers to store 1 to N messages, and buffers may be linked indefinitely through ABI-V5 memory. Monitoring can be set up to loop through buffers or stop for a snapshot mode.

Interrupt Processing

This chapter details servicing of the ABI-V5 VMEbus interrupts. The ABI-V5 provides very flexible interrupt selections for all operations to allow notification of 1553 processing events. Examples of such events include BC control block execution, 1553 message detected, error detected, and sequential monitor buffer swap. ABI-V5 interrupts have associated data packets which describe the type and time of the interrupt event to the host service routine (ISR).

BC, RT, and Sequential Monitor data structures also feature D64/D32 VMEbus master control words which allow for automated transfer of 1553 data to and from the host memory model. Setup and control of these words are detailed in the respective chapter.

The accompanying manual, *MIL-STD-1553 Interface Libraries for the ABI-V5*, provides detailed information for setting up and programming the ABI-V5 through SBS provided device drivers and high-level C functions or Ada procedures. SBS recommends reading the "1553 Operations" section prior to reading the *MIL-STD-1553 Interface Libraries for the ABI-V5* manual.

This manual assumes that the reader is familiar with 1553 terminology, especially the terms *bus controller* (BC) and *remote terminal* (RT). The term *host* or *host application program* refers to the user-written controlling program of the ABI-V5 and the ABI-V5 data structures associated with 1553 processing.



Note: The Hardware Control Registers (located in memory below 40h D16) are accessed as 16-bit words. *All other addresses, offsets, and data values described in this document refer to 32-bit longwords.*

3.2 Architecture Overview

The ABI-V5 is a multiprocessor, single card (standard 6U, B-Size) module for interfacing VMEbus processors (host systems) to one or two MIL-STD-1553 buses. The ABI-V5 has the unique characteristics of being able to fully simulate and monitor two 1553 buses while performing full VMEbus master and D64 operations.

The ABI-V5 is a multiprocessing architecture where 1553 simulation and monitoring tasks are divided into low and high levels of processing. Low-level processes control simulated BC messages and RT responses, and monitor 1553 messages for protocol verification. Low-level processing is performed independently on each daughter board by a dedicated DSP processor. High-level processing is performed by a RISC processor and is responsible for moving 1553 message packets in real-time to and from the DSP daughter boards (to/from user-defined data buffers.) The user's application program defines and manages BC, RT, and Monitor data structures in a main memory area where the RISC reads and stores 1553 messages.

BC, RT, Monitor data structures, and host command programming are controlled by the host's application program. Through memory data structures and command sets, the host controlling program may direct 1553 data for real-time processing.



Cross Reference: *Chapter 5: Host Commands/Control Registers*, *Chapter 6: BC Simulation*, *Chapter 7: RT Simulation/RT Monitoring*, *Chapter 8: 1553 Bus Monitoring*, and *Chapter 9: Interrupts* detail the host interface to the ABI-V5 application control structures.

3.2.1 Design Review

The design of the ABI-V5 incorporates an open systems philosophy. The RISC/DSP ABI-V5 is a generic processing engine (the RISC, DSP processors, and most of the hardware clock and control circuits are software programmable) that can be configured through various application programs. This manual explains the application program (firmware) SBS has designed for optimal 1553 bus processing and simulation. In this application, processing for either 1553 channel is performed independently through host-defined BC, RT, and Monitoring data structures (detailed in later chapters). The board's design allows for custom programs (written by the customer or SBS) to correlate data between buses or provide advanced processing to off-load the host system.

When installed into a VMEbus system, the ABI-V5 appears to the host processor as a 512KB or 2MB region of memory (memory mapped). The user needs to set address switches on the module in order for the host to access the board's memory.



Cross Reference: Please see the *Chapter 2: ABI-V5 Hardware Installation* for details on setting the A24/A32 address of the board. Other VMEbus settings for interrupt level and vectors are software programmable and are detailed later in *Chapter 5: Host Commands/Control Registers*.

The first 40h short words (20h longwords) of ABI-V5 memory are accessible only as D16 single cycle (this area is reserved for hardware control registers such as the CSR, clock, etc.). The upper memory region above the 80h byte address (100-1FFFFF or 100-7FFFFF byte) is accessible by D16/D32 single cycle slave, or D32/D64 Block Slave/Master. This upper memory region contains the software control registers.

The following paragraphs provide an architectural overview of the ABI-V5/RISC/DSP. The small blocked letter corresponds to the respective paragraph.

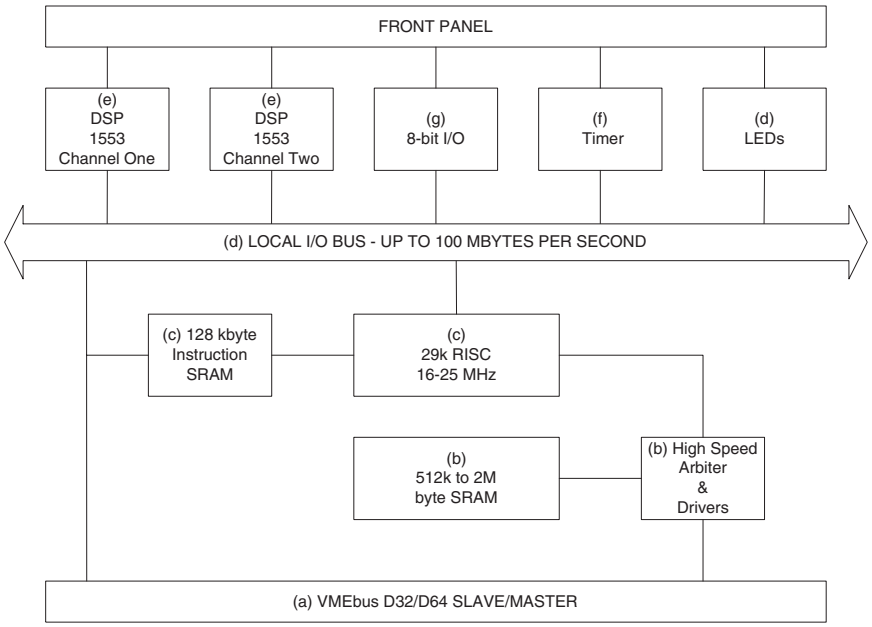


Figure 3.2.1: Multichannel ABI-V5 Block Diagram

3.2.2 Block (a): VMEbus Interface

Figure 3.2.1 shows the basic block diagram of the ABI-V5 module. From bottom to top, the first box shows the VMEbus interface. This circuitry includes the D16/D32/D64 slave and master interface, A24/A32 addressing, and interrupt logic.



Cross Reference: These signals are assigned to P2 per the VMEbus Rev C.1 standard and may be found in *Chapter 2: ABI-V5 Hardware Installation*.

The D32 and D64 interface is capable of bus master and slave block operations; D16 and D32 single-cycle slave is also supported. The bus master operations provide test and simulation systems with an advanced method of modeling 1553 data from local host memory. D64 Block transfers have a best-case transfer rate of 45 megabytes (MB) per second and a typical rate of 40 MB per second (tested between two ABI-V5 cards). D32 Block transfers are typically half that of D64 rates. Best-case D32 and D16 single-cycle rates are 150 ns with typical transfers at 250 ns. These rates are dependent on the master or slave device's VMEbus handshake.

3.2.3 Block (b): Memory Arbiter and Main Memory

The ABI-V5 provides a high-speed bus arbiter for memory access between the ABI-V5 RISC and VMEbus. In most cases, the VMEbus has priority for memory access resulting in a memory data access rate of less than 250 ns. The arbiter assures that neither system, the RISC or VMEbus, is interrupted for more than 40 ns for memory access.

The main memory area can be configured as 512KB or 2MB. There are no configuration switches or jumpers required—the RISC automatically powers up and detects the memory configuration. VMEbus access outside the module's memory area will result in a VMEbus error. The SRAM is high-speed 20-ns memory where the VMEbus host data path is 64 bits wide and the RISC data path is 32 bits wide.

3.2.4 Block (c): RISC Processor

The RISC processor is a 25-MHz 29005 by AMD. This processor is ideal for data and I/O movements due to its separate data buses for instruction and data and its capability for large data block movement on single cycles. The RISC can block move up to 127 32-bit words in single-cycle bursts, providing an access rate of 100 MB per second. The processor has 192 general-purpose registers, and SBS has implemented a 2:1 (two cycles for the first transfer and single-cycle transfers for subsequent words) local and memory bus access scheme. The 128 KB instruction RAM for the RISC is downloaded by the user at module startup, and instructions are executed in single, 40-ns cycles.



Cross Reference: Please see the *Chapter 4: Module Startup/Test* for details on instruction RAM loading.

3.2.5 Block (d): Local I/O Bus

The local I/O bus of the board is a general-purpose local bus. The bus interconnects the two 1553 daughter boards, hardware clock, 8-bit I/O, and LED logic. The bus and the RISC processor controlling this bus have direct access to the VMEbus for setup/control of key hardware registers such as the CSR and hardware clock. Cycle times for the local bus is the same as for the RISC: 40 ns. The RISC is capable of moving data to/from the 1553 daughters boards at 100 MB per second over the local bus.

3.2.6 Block (e): DSP 1553 Daughterboards

The 1553 daughter board is a separate computer system with a TI TMS320C40 DSP processor running at 20 MIPS. The DSP has on-board RAM (2 KB) and PROM (10 KB) for self-contained execution. DSP I/O port interconnect to an SBS-designed 1553 encoder/decoder (1553 UART designed by SBS) and FIFO control logic (for communication to the RISC). The DSP is downloaded and booted by the RISC at power-up.

Upon boot-up, the DSP immediately begins monitoring 1553 bus traffic from the transceiver and UART circuitry. The DSP simulates RT responses or begins BC transmissions as directed by the user's application program through data structures and host commands in main memory (host commands and data structures are reviewed in later chapters). The RISC processor communicates simulation commands and data to the DSP via the local bus and FIFO logic on the DSP. The DSP processor packs 1553 messages in FIFO logic for the RISC to read and process into user data structures in main memory.

3.2.7 Block (f): Timer

48-Bit Timer for Standard V5

This circuit provides a dual latching 48-bit, 1-µs timer (8 years in time), and two independent 23-bit 1-µs interrupt timers. The latching of a 48-bit time stamp (for each 1553 message) is controlled by the DSP through discrete logic to ensure accurate and consistent time marks. The 48-bit timer may be preset or reset by the host system, and may be driven from an external TTL or RS-422 signal (see Table 2.4.1 on page 5).

IRIG-B Timer for V5 IRIG

This circuit provides a dual latching 64-bit, 1-µs timer (based on an IRIG-B input), and two independent 23-bit 1-µs interrupt timers. The latching of an IRIG time stamp (for each 1553 message) is controlled by the DSP through discrete logic to ensure accurate and consistent time marks. The IRIG timer may be preset or reset by the host system when free running, and may be driven from an external IRIG signal (see Table 2.4.6 for pinout of J3).

Both the Standard V5 and the V5 IRIG cards contain 23-bit timers. Each 23-bit timer (8.3 seconds) may be programmed by the host computer. These timers are generally used for BC minor frame timing and for providing a hardware interrupt to the RISC. They may also be used for custom applications.

3.2.8 Block (g): 8-Bit I/O - External Interrupt (P2 Connections)

This 8-bit, bidirectional, TTL I/O port provides trigger functionality for 1553 processing. The RISC processor typically reads or writes to these ports when programmed 1553 events are to occur (as defined by the BC, RT, Monitor, or Interrupt data structures). For custom applications, these ports could be used as desired.

3.3 Memory Organization

As shown in Figure 3.3.1, the ABI-V5 memory map contains Data RAM (main memory), Instruction RAM, I/O space, and two extended I/O pages. The extended I/O pages provide memory access to each of the DSP cards (the host's application program does not need direct access to the DSP cards).

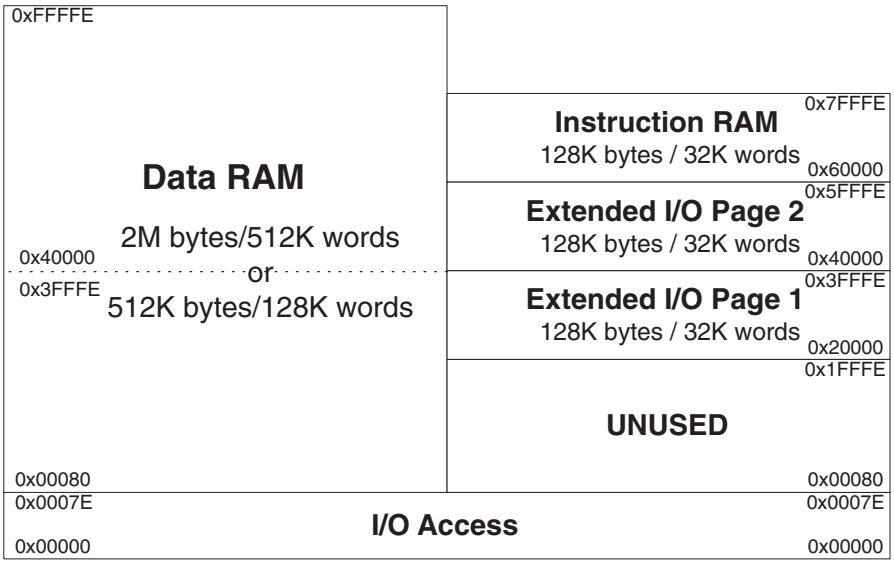


Figure 3.3.1: ABI-V5 Memory Map

The locations shown in Figure 3.3.1 are given in byte addressing. The region from 00000h to 0007Eh (I/O space) is accessible in 16-bit words. The region from 00080h and above is accessible in 32-bit longwords.

The host is able to select between main memory and Instruction RAM, Extended I/O Page 1, and Extended I/O Page 2 through a mode bit in the Control/Status Register (CSR).



Cross Reference: See *Chapter 5: Host Commands/Control Registers* for a detailed discussion of the CSR.

The current mode of the CSR and the memory address of the access governs where data is read or written. I/O Access (the first 80h bytes) is available regardless of the state of the CSR mode bit. The Disable I/O bit in the CSR restricts access to the I/O space. This may be preferable after startup to prevent miscellaneous accesses that can cause unpredictable results. Because this document deals primarily with data structures managed by the library in user memory, this memory map is provided for reference only.

3.3.1 Application Memory Organization

The 1553 application program controls key segments of main memory containing the pointer table, RT definition table, and RT status response tables for each of the two channels. Each channel contains these elements in separate areas of main memory, as shown in Table 3.3.1.

Table 3.3.1: General Main Memory Organization

Word Address	Byte Address	Description
0–07FFh	0–1FFCh	Reserved
0800h–0FFFh	2000h–3FFCh	Channel 1 control
1000h–17FFh	4000h–5FFCh	Channel 2 control
1800h–3FFFFh	6000h–FFFFCh	User space Data buffers BC command blocks Sequential monitor buffers

Separate control areas for each channel permit the ABI-V5 to operate two independent channels simultaneously. The control table for channel 1 starts at word address 800h, and the control table for channel 2 starts at word address 1000h. Each of these addresses is called the Control Base Address of the channel. The pointer table, RT response table, and RT definition table are located at offsets from the control base address of the channel.



Cross Reference: The pointer table and hardware control registers are detailed in *Chapter 5: Host Commands/Control Registers*. Module startup and test functions are detailed in *Chapter 4: Module Startup/Test*, and data structure control for BC command blocks, RT tables, Sequential Monitor, and Interrupt functions are detailed in Chapter 6, Chapter 7, Chapter 8, and Chapter 9, respectively.

3.3.2 View Product Codes

Memory addresses 100h, 101h, and 102h (32-bit longword addresses) contain the product codes and version numbers for the ABI-V5 mother board, daughter card 1, and daughter card 2 firmware loads, respectively. Memory addresses 108h, 109h, and 10Ah contain the system codes and version numbers for the ABI motherboard, daughter card1, and daughter card2 firmware loads respectively. During startup, the host must wait for memory location 100h to become nonzero, indicating that the ABI-V5 application firmware has loaded and initialized properly. Location 100h becoming nonzero is the last action to take place before the ABI-V5 firmware enters the main loop of the application.

4: Module Startup/Test

4.1 ABI-V5 Initialization Steps

Prior to startup, the ABI-V5 module must be properly installed in the host computer system.



Cross Reference: See *Chapter 2: ABI-V5 Hardware Installation* for instructions on how to install the ABI-V5 module.

Once this is accomplished, the ABI-V5 is ready to be configured and initialized for 1553 operations. The following startup procedure consists of several steps that must be completed in the proper sequence. The first four steps are common to both ABI-V5 channels.



Cross Reference: For more information on steps 1–4, see later sections in this chapter. For more information on steps 5–6, see *Chapter 5: Host Commands/Control Registers*.

1. Download the firmware files.
2. Start up the module and wait for self tests to complete.
3. Run additional BIT tests if desired.
4. Switch to application mode.
5. Program control registers *in each channel*.
6. Initialize and enable *the channel*.

The Control/Status Register may also be programmed anytime during ABI-V5 operation since it is always accessible to the host.

4.2 ABI-V5 Firmware Download Instructions

The ABI-V5 requires all firmware code and PLD data to be downloaded prior to initializing application data structures. This code and data is contained in two ABI-V5 download files which may be provided in three formats: ASCII, big endian binary, and little endian binary.

All three formats for these files contain data which are structured as 32-bit data pairs: a VMEbus longword offset address with the associated 32-bit data for that location. To download to the ABI-V5, the user:

1. Reads in a data pair from the file.
2. Calculates the target address by adding the board base address, as assigned by the user, to the longword offset address provided as the first 32-bit element of the data pair.
3. Writes the second 32-bit element of the data pair to the target address.
4. Repeat steps 1–3 until the end-of-file is reached.

4.2.1 Download Data Regions

The download data is loaded into two distinct areas of the ABI-V5: data RAM and 29-KB instruction RAM. Upon power-up or a board reset, the ABI-V5 is configured to address the data RAM area. To select instruction RAM for read and write operations, set the Mode Select bit and the Instruction Load bit in the Status/Control Register (bits 12 and 8, respectively) to “1”. To return to the data RAM area, clear the two bits either by writing zeros to CSR bits 8 and 12 or by resetting the board.

To properly download each of the data files, complete the following procedure:



Note: The Hardware Control Registers (located in memory below 40h D16) are accessed as 16-bit words. *All other addresses, offsets, and data values described in this document refer to 32-bit longwords.*

4.2.2 Firmware Download and Startup Procedure

1. Reset the ABI-V5 by writing a "1" to the CSR (offset 0h).

2. Download data RAM by doing the following:

   - ➢ Calculate the target offset by adding the longword offset value from the data RAM file to the board base address.
   - ➢ Write the associated data to the target offset.
   - ➢ Read the next data pair.
   - ➢ Repeat these steps until the end-of-file is reached.

3. Set the mode select and instruction load bits (bits 12 and 8) of the CSR to "1" (write 1100h to the CSR).

4. Download instruction RAM as follows:

   - ➢ Calculate the target offset by adding the longword offset value from the instruction RAM file to the board base address.
   - ➢ Write the associated data to the target offset.
   - ➢ Read the next data pair.
   - ➢ Repeat these steps until the end-of-file is reached.

5. Reset the ABI-V5 by writing a "1" to the CSR.

6. Clear the power-up/test status register by writing "0" to offset 120h.

7. Start the ABI-V5 firmware by writing a "2" to the CSR.

8. Go into a software loop, checking the value contained in the power-up/test status register. Exit the loop when the value is "1", indicating the power-up tests have completed.

9. Read the power-up/test result register (offset 121h). The value will be nonzero if errors were detected.

After this procedure has been completed, the ABI-V5 is in Built-in Test (BIT) mode awaiting a command. You can select BITs to be performed or you can initialize the board for Full Function 1553B operations.

4.2.3 Download File Example

An example of a data file in ASCII format is shown in Table 4.2.1.

Table 4.2.1: Data File in ASCII Format

Offset	Data
00000000	12643746
00000001	23447892
00000002	3A5BEE0
00000003	394F34D5

These four data pairs are to be written to the data RAM region starting at the board base address plus zero:

- 12643746 will be written to the base address of the board.
- 23447892 will be written to the base address plus 1.
- 3A5BEE0 will be written to the base address plus 2.
- 394F34D5 will be written to the base address plus 3.



Note: Data pairs are stored in hexadecimal, and the address is a longword offset.

The following example illustrates software code based on a LynxOS host system used to download data to the ABI-V5:

```
/*    This is an example of how to load the ABI-V5.  It is based on    */
/*    a LynxOs host system.  All specific Os calls are indicated.  The  */
/*    file type for the data is ASCII.                                  */

#include <stdio.h>
#include <stdlib.h>
/* Lynx specific */
#include <smem.h>
#include <errno.h>

FILE *f_ptr;
unsigned short *abiv5;
unsigned long  *load_ptr;

void set_pointer();

void main()
{
  unsigned long addr,data;

  /* create Lynx specific VME pointer */
  set_pointer();
```

```
  /* reset board */
  *abiv5 = 1;

  /* open data RAM file - version number would replace XXXX */

  if ((f_ptr = fopen("3001XXXX.dat","r")) == NULL)
  {
 puts("\nFile error");
 exit(0);
  }

  while(feof(f_ptr) == 0)
  {
 /* get data pair */
 fscanf(f_ptr,"%lX %lX\n",&addr,&data);
 /* calculate VME byte address based on longword offset */
 addr *= 4;
 load_ptr = (unsigned long *)((unsigned long)abiv5 + addr);

 /* write data */
 *load_ptr = data;
  }

  fclose(f_ptr);

  /* set board for instruction RAM access */
  *abiv5 = 0x1100;

  /* perform same sequence for instruction RAM */

  if ((f_ptr = fopen("1001XXXX.dat","r")) == NULL)
  {
 puts("\nFile error");
 exit(0);
  }



  while(feof(f_ptr) == 0)
  {
 fscanf(f_ptr,"%lX %lX\n",&addr,&data);
 addr *= 4;
 load_ptr = (unsigned long *)((unsigned long)abiv5 + addr);
 *load_ptr = data;
  }

  fclose(f_ptr);


 /* reset board to restore normal access and reload 29K pipeline */
  *abiv5 = 1;
  /* Lynx specific removal of VME pointer */
  smem_remove("XBOARD_PTR");
}

void set_pointer()
{
  smem_remove("XBOARD_PTR");
  abiv5 = (unsigned short *)
smem_create("XBOARD_PTR",0x48000000,0x70000,SM_READ | SM_WRITE);
  if (abiv5 == NULL)
  {
 puts("\nPointer error");
 smem_remove("XBOARD_PTR");
 exit(0);
  }
```

4.3 Built-In Tests

Table 4.3.1 contains a summary of the registers used to perform BITs.

Table 4.3.1: BIT Operational Registers

Byte Offset (Hex)	32-Bit Register Offset (Hex)	Register Description
480	120	Power Up /Test Status
484	121	Power Up / Test Result
488	122	Reserved
48C	123	Mode Command
490	124	Test Control
494	125	Selected Test Value
498	126	Reserved
49C	127	Error Count
4A0	128	Test Loop Count



Note: Approximately 85% of the ABI-V5 module is tested during power-up. If "all tests" are performed in BIT mode, 90% of the board is tested. The host interface is not fully tested due to user constraints (i.e., RAM tests). False alarm rate is estimated to be less than 1%. We have not had a false alarm from a properly performed test and have many boards in the field.

4.3.1 Performing All Tests

To perform all of the BIT tests, complete the following steps:

1. Write "0000" to the power up/test result register (offset 0121h).
2. Write "0000" to the error count register (offset 0127h).
3. Write the number of test loops to be performed to the test control register (offset 0124h).
4. Write "000B" to the mode command register (offset 0123h).
5. Read the power-up/test status register (offset 0120h) and wait for a test complete indication (value=00A5h).
6. Read the power-up/test result register (offset 0121h). If the value is nonzero, errors were detected. In addition, the PROG LED will turn red if a failure is detected.

4.3.2 Performing Selected Tests

Specific BIT tests can be run using the selected test command. This command uses the bit field values defined in Table 4.3.2 to determine which tests to run.

Table 4.3.2: Selected Test Bit Field

Bit Position	Self Test Description
0	29K Sanity Test
1	I/O Port Test
2	Data RAM Test
3	29K Burst Mode Test
4	System Timer Test
5	DSP/1553B Test
6	Reserved
7	Reserved

For example, to perform the data RAM and system timer tests, load 14h in the selected test register.



Note: The DSP/1553B test requires that the bus channels are terminated. The self-test code automatically detects the number of installed daughter boards.

To perform selected BIT tests, complete the following steps:

1. Write "0000" to the power up/test result register (offset 0121h).
2. Write "0000" to the error count register (offset 0127h).
3. Write the number of test loops to be performed to the test control register (offset 0124h).
4. Determine the bit field value for the selected tests using Table 4.3.2 and write this value to the selected test register (offset 0125h).
5. Write "0008" to the mode command register (offset 0123h).
6. Read the power-up/test status register (offset 0120h) and wait for a test complete indication (value=00A5h).
7. Read the power-up/test result register (offset 0121h). If the value is nonzero, errors were detected. If there were errors, the type of failure is indicated by the bit positions which are set to "1". The bit descriptions are the same as those in the selected test register.

4.4 Module Operation

The BC, RT, Monitor, and Interrupt functions of this interface are executed from application mode. To enter this mode from the self-test mode, perform the following operation:

1. Write "0" to product code registers 100h, 101h, and 102h.
2. Write "000D" to the mode command register (offset 0123h).

   This places the ABI-V5 in application mode, at which time the instruction RAM and data RAM configuration words are compared. If they are not identical, an error is generated and processing ceases. If they are identical, the 16-bit value is copied to word address 103h and processing continues.



Cross Reference: For more information, see the section *Firmware Configuration Register* on 4-8.)

3. Go into a software loop, checking the product code register (offset 0100h) for a nonzero value. Exit the loop when the register contains a nonzero value.
4. Load data structures and control registers for each channel into ABI-V5 memory.
5. Issue the Initialize Channel Host Command for each channel to initialize internal registers. Once the channel has been properly initialized, the firmware writes a 01h to the RESP control register (offset 03h from the Control Base Address of the channel).
6. Issue the Enable Channel Host Command for each channel to enable 1553 bus processing.
7. Verify that the RESP control register is incrementing. This ensures that the firmware has successfully performed startup and is currently enabled for processing 1553 data.

After this is completed, the ABI-V5 operates as described in *Chapter 5: Host Commands/Control Registers*.

4.5 Firmware Configuration Register

The instruction RAM and data RAM data files each contain a configuration word which details the specific firmware version being downloaded to the ABI-V5. When the ABI-V5 is placed in application mode, it compares the configuration word in the instruction RAM with that in the data RAM. If the two configuration words are identical, the value of the matching word is written to bits 0–15 of word address 103h on the ABI-V5. If the two words do not match, the data RAM value is written to bits 0–15 of word address 103h of ABI-V5 memory, and the instruction RAM value is written to bits 16–31 of word address 103h of ABI-V5 mem-

ory. In addition, the firmware issues a startup error to offset 100h from the Control Base Address of the offending channel (with bit 3 set to "1"), the PROG LED turns red, and processing halts. The contents on address 103h are illustrated in Figure 4.5.1.

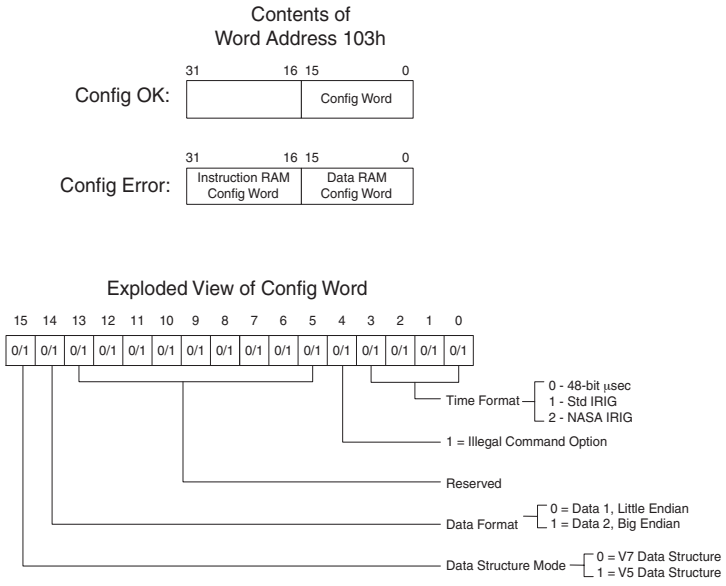


Figure 4.5.1: Contents of Configuration Register

- ***Bits 0–3: Time Format.*** This portion of the configuration word indicates which time format is supported by the firmware. If these bits contain a value of "0", the standard 48-bit clock is supported; if these bits contain a "1", the standard IRIG time format is supported; if these bits contain a "2", the NASA IRIG time format is supported. Only one of these time formats is supported in a given load of firmware.
- ***Bit 4: Illegal Command Option.*** If this bit is set to "1", the Illegal Command option is supported by the firmware.
- ***Bit 15: Data Format.*** This bit specifies which data format is supported by the firmware: if this bit is "0", Data 1 (little endian) format is supported; if this bit is "1", Data 2 (big endian) format is supported.

5: Host Commands/Control Registers

This chapter reviews control registers and host commands. Control registers provide key setup information used by the host and ABI-V5 system (i.e., setting of BC intermessage gap time, host controlled DMA registers, and the board's control and status register.) Host commands are control or request commands issued by the host program to invoke a specific ABI-V5 action—typically a 1553 related process, such as Start BC execution, Update RT Status Word information, etc. In general, control registers are programmed by the host with 1553 processing information that does not change upon activation of ABI-V5 processing for the channel. Host commands are typically used for activating or controlling those ABI-V5 processes that the host desires to manage "on-the-fly" during real-time 1553 execution.

The first part of this chapter describes hardware and software registers. Hardware control registers provide the following module functions to the host:

- Control/Status Register (CSR)
  - Module reset and operation control
  - VMEbus interrupt control
  - Front panel LED control
- Host DMA Control Register (HOST DMA CNTL)
  - Host control of VMEbus master transfers
  - VMEbus interrupt vector setting for optional DMA interrupts
  - 48-bit module timer setup and control
  - VMEbus interrupt vector settings for 1553 channels 1 and 2

The latter part of this chapter details the "global" software control registers for controlling each 1553 channel's host commands, interrupt queues, BC execution, allowed RT response time (monitoring), sequential monitor control, and database address offset.



Note: The terms offset, pointer, and address are used interchangeably when referring to SBS device memory. The device memory is accessed by an offset from the device's base memory address.

Note: All addresses and data values are in hexadecimal.

5.1 Hardware Control Registers

Table 5.1.1 summarizes the hardware control registers. Detailed discussions of the registers are found following the table.The registers in Table 5.1.1 apply to operations of both 1553 channels.

Table 5.1.1: Hardware Control Registers - D16 Access Only

Byte Addr (Hex)	16-Bit Word Addr	Type	Name	Description	Page Ref.
00h	00h	RW	CSR (from host only)	Control/Status register. Key register for proper initialization and operation of the ABI-V5. This is the only register always accessible to the host. See the description following this table.	5-3
02h	01h	RW	HOST DMA CNTL	Key register for host DMA transfer operations. See the description following this table.	5-6
04h	02h	WO	ABI ADDL	This register contains the least significant 16 bits of the ABI source address for a host DMA transfer.	5-8
06h	03h	WO	ABI ADDH	This register contains the most significant 6 bits of the ABI source address for a host DMA transfer.	5-8
08h	04h	WO	VME ADDL	This register contains the least significant 16 bits of the VMEbus destination address for a host DMA transfer.	5-8
0Ah	05h	WO	VME ADDH	This register contains the most significant 16 bits of the VMEbus destination address for a host DMA transfer.	5-8
0Ch	06h	WO	WORD CNT	This register contains the number of words to transfer regardless of the transfer width.	5-8
0Eh	07h	RW	INTV_DMA	Bits 0–7 of this register set the VMEbus Interrupt Vector for DMA transfer operations. Bit 0 is the right most, least significant bit.	5-8
1Ah	0Dh	RW	INTV_1553 Channel 1	Bits 0–7 of this register set the VMEbus Interrupt Vector for 1553 user-selected interrupts for ABI-V5 channel 1. Bit 0 is the least-significant bit.	—

Byte Addr (Hex)	16-Bit Word Addr	Type	Name	Description	Page Ref.
1Ch	0Eh	RW	INTV_1553 Channel 2	Bits 0–7 of this register set the VMEbus Interrupt Vector for 1553 user-selected interrupts for ABI-V5 channel 2. Bit 0 is the least-significant bit.	—
				Note: Accessing addresses not described in this table (0F–7Fh bytes) may adversely affect ABI-V5 operation and will produce unpredictable results. Do not access these registers!	
			STANDARD V5 TIME		
10h	08h	RW	ITCR	Internal Timer Control Register. Defines the mode of operation for the timer.	5-11
12h	09h	RW	ITDRL	Internal Timer Data Register. Lower 16 bits of the system timer.	5-11
14h	0Ah	RW	ITDRM	Internal Timer Data Register. Middle 16 bits of the system timer.	5-11
16h	0Bh	RW	ITDRH	Internal Timer Data Register. Upper 16 bits of the system timer.	5-11
			V5 IRIG TIME		
10h	04h	RW	ITDRL	Internal Timer Data Register. Lower 32 bits of the system timer. D32 Access Only: The longword address is 04h.	—
14h	05h	RW	ITDRH	Internal Timer Data Register. Upper 32 bits of the system timer. D32 Access Only: The longword address is 05h.	—
18h	0Ch	RW	ITCR	Internal Timer Control Register. Defines the mode of operation for the timer.	—

5.1.1 Control/Status Register

The Control/Status Register (CSR) contains bits which control module operation, VMEbus interrupts, and front panel LEDs. This register may be accessed at any time by the host. *Except where otherwise noted, the host has both read and write access to the CSR bits.*

CSR bits 8 and 12 are set during the software download procedure as described in Chapter 4: Module Startup/Test. Bits 0 and 1 are set during module startup. During normal operation, bits 0, 2, 8, and 12 should be cleared.

Table 5.1.2 describes the CSR bits in detail.

Table 5.1.2: Control/Status Register Bits (Address offset 0000 from VMEbus address - D16 access only)

Bit	Function	Bit Values	Description
0	Reset (WO)	0 = normal operation 1 = reset	Reset clears all state machines for host and firmware access. The firmware is halted after being reset and the instruction pipeline is loaded with four instructions. If new firmware code is loaded, the reset bit must be asserted.
1	Run	0 = firmware stop 1 = firmware run	
2	Step	0 = firmware normal mode 1 = firmware step mode	Toggling this bit causes the firmware to execute a single instruction. A reset must be issued to return to normal operation.
3	VMEbus Interrupt Enable 1	0 = clear 1 = enable interrupt	After enabling this bit, an interrupt can be generated by writing to register word offset 11h with the corresponding vector in register word offset 0Dh.
4	Reserved		
5	Running (RO)	0 = firmware halted 1 = firmware running	
6	VMEbus Interrupt Enable 2	0 = clear 1 = enable interrupt	After enabling this bit, an interrupt can be generated by writing to register word offset 12h with the corresponding vector in register word offset 0Eh.
7	Interrupt Pending 1 Interrupt Clear 1	0 = no interrupt 1 = interrupt pending (read)/clear interrupt (write)	Reading this bit indicates whether VMEbus interrupt 1 is pending. Setting this bit will clear the interrupt.
8	Instruction Load Enable	0 = normal operation 1 = load firmware instruction enable	Setting this bit allows host access to the firmware instruction RAM. This bit is set along with the access mode select bit. This bit should be cleared prior to operation of the board.
9	Interrupt Pending 2 Interrupt Clear 2	0 = no interrupt 1 = interrupt pending (read)/clear interrupt (write)	Reading this bit indicates whether VMEbus interrupt 2 is pending. Setting this bit will clear the interrupt.
10	LED - Red On	0 = off (power-up condition) 1 = on	
11	LED - Green On	0 = off 1 = on (power-up condition)	

Bit	Function	Bit Values	Description
12	Access Mode Select	0 = normal operation 1 = extended I/O access	Setting this bit allows host access to the extended I/O space for the instruction RAM and the daughter board interface. This bit must be reset prior to normal operation of the board.
13	Interrupt Level Select 0		These bits determine the VMEbus interrupt priority level to be used by the ABI-V5 board. This level will be used by all interrupts.
14	Interrupt Level Select 1		
15	Interrupt Level Select 2		

5.1.2 Host DMA Registers

The ABI-V5 is capable of being a DMA master on the VME bus and can perform DMA transfers to and from the host in one of two ways:

- Through the BC, RT, or Sequential Monitor data structures
- Directly from the host

To initiate an ABI controlled DMA transfer from the host, program the following hardware control registers at any time: HOST DMA CNTL, ABI ADDL, ABI ADDH, VME ADDL, VME ADDH, WORD CNT, and INTV_DMA. If both host and firmware initiated DMA transfers are being used, use the Host Request DMA Host Command to initiate the Host DMA. Table 5.1.3 defines the bits in the HOST DMA CNTL register; descriptions for the remaining six registers follow the table.

Table 5.1.3: HOST DMA Control Register (Address Offset 0001 from VMEbus Address: D16 Access Only)

Bit	Function	Bit Values	Description
0	Host DMA Status/ Host DMA Start (M_STAT/ M_START)	As read: 1 = DMA Transfer in Progress As written: 1 = Start DMA Transfer	Write a "1" to this bit to start a Host DMA transfer. The bit value will remain "1" until the DMA transfer is complete.
1	Host DMA Transfer Direction (M_DIR)	0 = Transfer from ABI-V5 to VMEbus 1 = Transfer from VMEbus to ABI-V5	This bit controls the direction of master transfers by the ABI-V5. Set to "0" to cause the ABI-V5 to perform a master transfer from ABI-V5 memory to VMEbus. Set to "1" to cause the ABI-V5 to transfer data from VMEbus to ABI-V5.
2,3	Host DMA Bus Request Level (M_BR)	00 = Bus Request Level 0 01 = Bus Request Level 1 10 = Bus Request Level 2 11 = Bus Request Level 3	These bits control the bus level for the VMEbus transfer. "11" is typically the highest, but this setting is dependent on the bus control settings/protocol of the slot 0 VMEbus arbiter.
4	Host DMA Address Width (M_A24)	0 = A32 1 = A24	This bit selects either a 32 or 24 bit address width for the VMEbus transfer. The VMEbus address corresponds to the 32 or 24 least significant bits of VME ADDH/L, respectively.

Bit	Function	Bit Values	Description
5	Host DMA Data Width (M_D64)	0 = D32 1 = D64	This bit controls the width of data transfers to/from VME-bus memory. The Length Counter value is in increments of this bit setting—i.e., if bit 5 is set to "0" and Length Counter is 16, 16 32-bit transfers will occur.
6	Host DMA Spv/ Non-Priv Access (M_SPV)	0 = Non Privileged (NORMAL) 1 = Supervisory	This bit controls memory address modifiers. The ABI-V5 responds to either mode.
7	Host DMA Address Increment Disable (M_INCDIS)	0 = Increment VMEbus Address 1 = No Address Increment	This bit controls automatic VMEbus address increments for ABI-V5 block transfers. In short, this bit allows for transfers to/from regular memory or FIFOs. The amount the address is incremented depends on the setting of bit 5; a D32 transfer increments the VME-bus address by 4 and a D64 transfer increments the address by 8.
8,9,10	DMA Interrupt Level (M_ISEL)	000 = Int. Level 0 (no interrupts) | 111 = Int. Level 7	These bits control the interrupt level for both Host and ABI initiated DMA transfers. "111" is the highest level.
11	Host DMA Interrupt Enable (M_IENB)	0 = Disable Interrupt after DMA 1 = Enable Interrupt after DMA	This bit enables/disables an interrupt by the ABI-V5 to the VMEbus after a block (DMA) transfer has completed. This feature allows the ABI-V5 to notify the host processor that a DMA transfer has completed for synchronization.
12	Host DMA Complete (M_COMP)	READ ONLY 1 = Host DMA Transfer has completed	When read, this bit contains the state of the Host DMA Complete bit. This bit is automatically cleared after being read.
13	Host DMA Error (M_ERR)	0 = Host DMA Transfer completed successfully 1 = Error occurred during Host DMA Transfer	When read, this bit indicates whether an error occurred during a Host DMA Transfer. This bit is automatically cleared after being read.
14,15	Master ID Number (M_ID)	Determined by user	Use these bits as desired to identify a Host DMA Transfer. These are read/ write bits which are not affected by the transfer.

ABI ADDL, ABI ADDH Address 0002h, Address 0003h. ABI ADDL (ABI-V5 source address low) and ABI ADDH (ABI-V5 source address high) define the ABI-V5 source address for DMA operation to the VMEbus. These two words combine to make up a 22-bit ABI-V5 byte address which includes all the bits of ABI ADDL and bits 0–5 of ABI ADDH (bits 6-15 of ABI ADDH are not used and should be set to zero). The source address can be any ABI-V5 local address.

VME ADDL, VME ADDH Address 0004h, Address 0005h. VME ADDL (VMEbus address low) and VME ADDH (VMEbus address high) combine to make the VMEbus destination address words (24 or 32 bits as defined by bit 4 of the Master CSR). VMEbus memory, as with ABI-V5 memory, is big-endian oriented. Most VMEbus processor and memory modules are byte oriented; thus, D32 or D64 transfers must start on a four-byte or eight-byte boundary.

WORD CNT Address 0006h. WORD CNT is the number of transfers to be performed over the VMEbus. The count value is for either D64 or D32 words, as designated by bit 5 of the Master CSR word. For example, if WORD CNT is 128 and bit 5 is set to "1" (selecting D64), 128 64-bit words will be transferred. Thus, the length count is not a byte count, but a VMEbus transfer count. The minimum transfer value is zero and the maximum is 65535.

INTV_DMA Address 0007h. This register contains the interrupt vector used when the master interface interrupts the VMEbus.

When DMA operations are initiated from BC, RT, or Monitor data structures, or from the use of the Host DMA Request Host Command, the values are copied to these master registers from the software data structures. In the software data structures, these values are stored as 32-bit longwords. The firmware breaks each 32-bit longword into two 16-bit words, copying the values into the master DMA registers. As a result, the 32-bit values for the ABI-V5 and VME addresses are copied into the high (ABI ADDH, VME ADDH) and low (ABI ADDL, VME ADDL) master registers. The Word Count/Vector data word is copied to WORD CNT and INTV_DMA.

The vector portion of the Word Count/Vector word allows the user to set a unique DMA Interrupt Vector for each DMA transfer initiated by the firmware, provided DMA Interrupts are enabled (bit 11 of the DMA Control Word). Access this function by setting an 8-bit interrupt vector value in bits 16–23 of a DMA Word Count/Vector word. If these bits are nonzero, the firmware will write the value they contain to the INTV_DMA master register during the startup process of a DMA transfer. If these bits are zero, the current value contained in INTV_DMA is used as the interrupt vector.



Warning: If using this feature, be certain that the interrupt vector supplied to INTV_DMA is correct. For example, if a unique vector is used for sequential monitor buffer DMAs, that value will be written to INTV_DMA during the sequential monitor buffer DMA process. If this is then followed by an RT buffer DMA where DMA interrupts are enabled (and the Interrupt Vector in the RT buffer's DMA Word Count/Vector word is zero), the interrupt vector supplied to the host will be that of the sequential monitor buffer. Processing may not be the same as it is for an RT buffer. It is the programmer's responsibility to ensure that DMA transfers and Interrupt Vectors are coordinated properly.

Example: Steps to Initiate a Host DMA Transfer

Complete the following steps to initiate a host DMA transfer:

1. Write a control value to HOST DMA CNTL (address 0001h), setting bit 0 to "0".
2. Write the lower 16 bits of the ABI-V5 internal address to ABI ADDL (address 0002h).
3. Write the upper 6 bits of the ABI-V5 internal address to ABI ADDH (address 0003h).
4. Write the lower 16 bits of the VMEbus address to VME ADDL (address 0004h).
5. Write the upper 16 bits of the VMEbus address to VME ADDH (address 0005h).
6. Write the length of the transfer to WORD CNT (address 0006h).
7. If using interrupts, write a VMEbus Interrupt Vector to INTV_DMA (address 0007h).
8. Write the control value written in step 1 logically OR'd with "1" to HOST DMA CNTL. This sets bit 0 to "1", causing the Master interface to start the Bus Request cycle and transfer data to or from the VMEbus.

Example: C Code for a Host DMA Transfer

The following is a pseudo-C code example of an ABI-V5 to VMEbus Master transfer.

```
/* HEADER FILE  */
#define BASE_ADDR 0x48000000
/* Master Register Address Definitions */
#define M_CSR0x01
#define M_IAL0x02
#define M_IAH0x03
#define M_VAL0x04
#define M_VAH0x05
#define M_LEN0x06
#define M_VEC0x07
/* Master - Control/Status Register Bit Definitions */
#define M_START0x01
#define M_READ0x02
#define M_WRITE0x00
#define M_BR00x00
#define M_BR10x04
```

```
#define M_BR20x08
#define M_BR30x0C
#define M_A320x00
#define M_A240x10
#define M_D320x00
#define M_D640x20
#define M_NPRIV0x00
#define M_SPV0x40
#define M_INCDIS0x80
#define M_INTR10x900
#define M_INTR20xA00
#define M_INTR30xB00
#define M_INTR40xC00
#define M_INTR50xD00
#define M_INTR60xE00
#define M_INTR70xF00

#define M_STAT0x0001
#define M_ERR0x2000
#define M_COMP0x1000
#define M_INPROG0x0001

unsigned short *abi = BASE_ADDR;

/*********************************************************
master_abi_read()

This routine causes the ABI to request the vmebus and read length D32
words from VMEbus address vme_addr and write them to ABI internal
address int_addr.
*********************************************************/
master_abi_read(int_addr, vme_addr, length)
unsigned long int_addr;
unsigned long vme_addr;
unsigned short length;
{
/* this routine assumes the following:
Bus Request Level 2
A32 Addressing
D32 Accesses
Non-Privileged Accesses
No Interrupts
*/
unsigned short control_value;

control_value = M_READ | M_BR2 | M_A32 | M_D32 | M_NPRIV;
*(abi+M_CSR) =  control_value;
*(abi+M_IAL) = int_addr & 0xffff;
*(abi+M_IAH) = int_addr >>16;
*(abi+M_VAL) = vme_addr & 0xffff;
*(abi+M_VAH) = vme_addr >>16;
*(abi+M_LEN) = length;
*(abi+M_CSR) = control_value | M_START;

while((*(abi+M_CSR) & M_STAT) == M_INPROG)  {
check_for_timeout();
}

check_for_errors();
}
```

5.1.3 Standard V5 Timing Hardware Registers

The ITCR, ITDRL, ITDRM, and ITDRH registers configure the 48-bit hardware timer used for 1553 message time stamping in ABI-V5 Sequential Monitoring. This timer may be read, reset, or preset as desired by the host software. The ABI-V5 timer may be driven from an internal clock or from an external source (front panel or P2 connector). If multiple ABI-V5s are in use, one ABI-V5 may be configured as the master clock.

Internal Timer Control Register (ITCR)

Address Offset 0008. This register defines the mode of operation of the timer and contains sixteen bits, described in Table 5.1.4. The default value for this register is 4011h.

Table 5.1.4: Internal Timer Control Register (ITCR) - D16 Access Only (Address Offset 0008 from VMEbus Address)

Bit	Function	Description
0	Clock Enable/Disable (ENABLE)	Set this bit to “1” to enable the system timer when the external enable (bit 4) is active and the clear inputs (bit 2 and the external clear signal) are inactive. Default for this bit is “1” or enabled.
1	Reserved	
2	Timer Clear (TMRCLR)	The clear signal to the timer is the OR of this bit and the external clear signal. To clear the timer to a count of zero, set this bit to “1” and hold for the entire clock period (one microsecond). The timer will remain cleared until this bit is cleared and the timer is enabled.
3	External Clock Select (EXTSEL)	Set this bit to “1” to cause the system timer to run from the external clock input.
4	External Enable Output (EXTENB)	Set this bit to “1” to activate the external enable output signal. If this ABI-V5 is operating independently, this bit enables the internal system timer. If multiple ABI-V5s are in use and this ABI-V5 is designated as the master clock, this bit enables the timers on the slave devices. (If this ABI-V5 is a slave device, set this bit to “0”.)
5	External Clock Master Enable (EXTCEN)	When multiple ABI-V5s are in use, only one module may act as the master clock. Set this bit to “1” to configure this ABI-V5 as the master clock. The master clock signal is taken from the internal clock of this ABI-V5. See Table 5.1.5 for an illustration of the clock options which may be selected using bits 3 and 5.
6, 7	Reserved	
8, 9	Clear Type Selection (CLRSEL0,1)	These bits select the configuration of the external clear input signal, as detailed in Table 5.1.6. The default configuration is level, active low.
10–13	Reserved	
14	External Enable Input (EXTENBI)	This is a read only bit which indicates whether the external enable signal is activated (1 = enabled, 0 = disabled).
15	System Timer Clear Condition (CLRCOND)	This is a read only bit which indicates the OR’d combination of the system timer internal clear bit and the external clear signal.

Table 5.1.5: Clock Options

EXTCEN (Bit 5)	EXTSEL (Bit 3)	Clock Used	Clock Input	Operating Mode
0	0	Internal	None	Normal Mode (default)
0	1	External	None	Normal Mode using external clock
1	0	Internal	Internal	Clock Master Mode
1	1	External	External	Illegal Mode

Table 5.1.6: External Clear Configurations

CLRSEL1 (Bit 9)	CLRSEL0 (Bit 8)	External Clear Signal Configuration
0	0	Level, Active Low
0	1	Level, Active High
1	0	Edge, Rising
1	1	Edge, Falling



Note: Disable the timer before changing the state of the Timer Clear bit (bit 2).

Internal Timer Data Registers ITDRL, ITDRM, ITDRH

Address Offsets 0009, 000A, 000B. These registers are used to preset or read the 48-bit timer used for time stamping 1553 messages. ITDRL holds the lower 16 bits, ITDRM holds the middle 16 bits, and ITDRH holds the upper 16 bits of the 48-bit timer. Perform the following steps to write to these registers:

1. Clear bit 0 of the ITCR to disable the timer.
2. Write the desired values to ITDRH, ITDRM, and ITDRL, in that order.
3. Set bit 0 of the ITCR to enable the timer.

When bit 0 of the ITCR is enabled, the hardware will load the 48-bit internal timer with the programmed values. This function can be performed with internal or external timer configurations.

Perform the following steps to read these registers:

1. Read ITDRL, latching the upper 32 bits of the timer.
2. Read ITDRM and ITDRH. Because the timer is latched, it may be read anytime in the future using only ITDRM and ITDRH, as long as ITDRL is not reread.

5.1.4 V5 IRIG Timing Hardware Registers

For the ABI-V5 with IRIG option, the ITCR, ITDRL, and ITDRH registers configure the IRIG hardware timer used for 1553 message time stamping in ABI-V5 Sequential Monitoring. This timer may be read, reset, or preset as desired by the host software. The ABI-V5 timer may be driven from an internal clock or from an external source (front panel).

Internal Timer Control Register (ITCR)

Address Offset 000C. This register defines the mode of operation of the timer and contains sixteen bits, described in Table 5.1.7. The default value for this register is "0".

Table 5.1.7: Internal Timer Control Register (ITCR): D16 Access Only (Address Offset 000C from VMEbus Address)

Bit	Function	Description
0	Clock Enable/Disable (ENABLE)	Set this bit to "1" to enable the clock; set this bit to "0" to disable the clock. Default for this bit is "1" or enabled.
1–9		Reserved
10	IRIG In (RO)	Read Only: when read, this bit indicates whether an external IRIG signal is being used (if "1", external IRIG is present).
11–15		Reserved

Internal Timer Data Registers ITDRL, ITDRH

Word Address Offsets 0008, 000A/Longword Address Offsets 0004, 0005. These registers are used to preset or read the IRIG timer used for time stamping 1553 messages. ITDRL holds the lower 32 bits, and ITDRH holds the upper 32 bits of the IRIG timer.

The registers contain the IRIG days, hours, minutes, and seconds in BCD format, and the milliseconds and microseconds in hexadecimal, as shown in Figure 5.1.1.

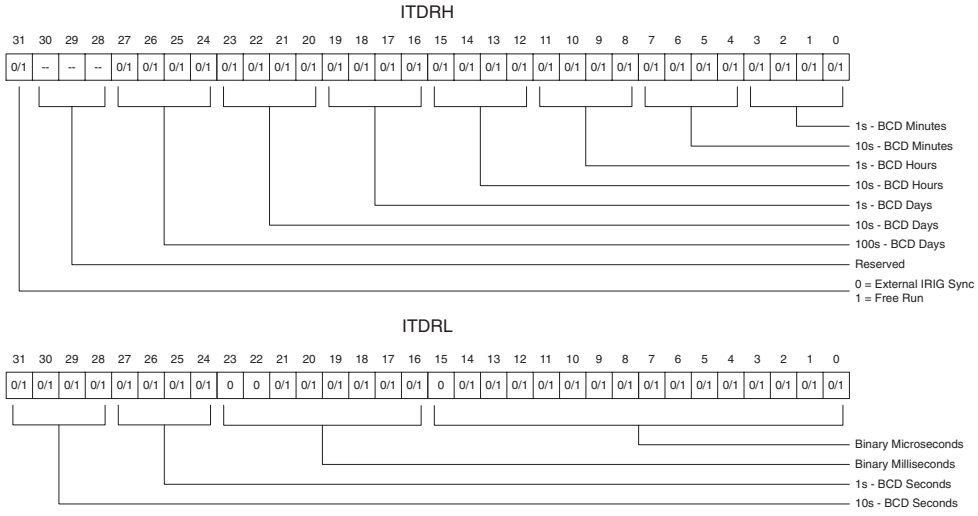


Figure 5.1.1: IRIG Time Registers

Bits 16–23 of ITDRL contain the binary milliseconds. The binary milliseconds contain the number of 20-millisecond counts within the current second. Bits 0–15 of ITDRL contain the binary microseconds. To calculate the total number of microseconds in a time value read from ITDRL and ITDRH, use the following formula:

```
(binary milliseconds * 20,000) + binary microseconds
```

If bit 31 of ITDRH is “0”, it indicates that the board is in sync with an external IRIG signal.



Note: If there is no external IRIG source, the internal timer will begin counting at zero upon module startup. It may be loaded with a desired value, to the nearest second, as long as no external signal exists.

Perform the following steps to write to these registers:

1. Clear bit 0 of the ITCR to disable the timer.
2. Write the desired values to ITDRH, and ITDRL, in that order. The loaded value will be accurate to the nearest second.
3. Set bit 0 of the ITCR to enable the timer. When bit 0 of the ITCR is enabled, the hardware will load the IRIG internal timer with the programmed values.

Perform the following steps to read these registers:

1. Read ITDRH, latching the lower 32 bits of the timer.
2. Read ITDRL. Because the timer is latched, it may be read anytime in the future using only ITDRL, as long as ITDRH is not re-read.

5.2 Software Control Registers (1553 Processing Control)

The Software Control Registers table contains parameters and pointers to various data structures necessary for successful operation of the ABI-V5's 1553 firmware. The definitions of these registers are identical for each channel; the only difference lies in the location and contents of the registers. Table 5.2.1 describes the Software Control Registers. The addresses are offsets from the Control Base Address of the channel (800h for Channel 1 and 1000h for Channel 2.) The actual physical address in the ABI-V5 is the sum of the Control Base Address of a channel and the offset. Each channel control area is 800h words in length, providing 2K words of control for each channel.

Table 5.2.1: Software Control Registers—D32 Access (Offsets Are from Word Address 800h for Channel 1 and 1000h for Channel 2)

32-Bit Word Offset (hex)	Byte Offset	Name	Description	Default	Page Ref.
			General System Registers		
00h	00	HOSTCMD1	Host Command Word 1. Holds the command word portion (parameter 1) of a host command.	0	5-28
01h	04	HOSTCMD2	Host Command Word 2. Holds the data word portion (parameter 2) of a host command.	0	5-30
02h	08	HCRESP	Host Command Response Register. If 01h, host command is complete; if 02h, host command is being processed; if 03h, command not valid or error has occurred.	0	5-27
03h	0C	RESP	Firmware Response word. Values are as follows: 0 = Channel Not Initialized, 1 = Channel Initialized. Value constantly changes while firmware is enabled for processing bus data (I/O enabled).	0	5-19
			Interrupt Processing Registers		
04h	10	IQPTR1	Offset to Interrupt Queue 1. Contains ten-word entries that describe the cause of an interrupt.	None	9-7
05h	14	IQPTR2	Offset to Interrupt Queue 2. Same as IQPTR1. This is the queue which is currently active. The host should *not* process this queue.	None	9-7

32-Bit Word Offset (hex)	Byte Offset	Name	Description	Default	Page Ref.
06h	18	IQCNT1	Contains the number of entries in Interrupt Queue 1. This is used by the host computer's ISR to process the IQPTR1 queue.	0	9-7
07h	1C	IQCNT2	Contains the number of entries in Interrupt Queue 2.  **Note:** Do not access this register while the ABI-V5 is processing 1553 data.	0	9-7
08h	20	IQNUM	Defines the maximum number of entries per interrupt queue. If the number of entries exceeds the value in IQNUM, an interrupt overflow will occur.	0	9-7
			BC Simulation Registers		
09h	24	BCCPTR	Current Bus Control Program Block. Contains the offset to the currently active BC command block.	0	6-2
0Ah	28	BCLPTR	Last Bus Control Program Block. Contains the offset to the Last BC Command Block that executed.	0	6-2
0Bh	2C	BCIMGP*	Use to change the BC intermessage gap time (in 250-ns tics): `((1553 Intermessage Gap - 2 ) * 4) - 3`	185h (1553B gap of 100 µs)	6-2
0Ch	30	BCRETRY	BC Retry count and bus pattern. Number of retries to be performed for a BC error condition. Defines bus (A and/or B) to be used for BC retries.	0	6-3
0Dh	34	RSPGPA*	Use to program the maximum allowable status response gap time (in 250-ns tics.) `((Minimum 1553B Status Gap - 2 ) * 4)`	30h (1553B gap of 14 µs)	6-3
0Eh	38	FRMTMR	Contains the Minor Frame Time for BC chain execution, in 1-µs tics.	0	6-4
1Dh	74	BCSMSK	Contains the 32-bit Status Word Mask. Bits 0–15 contain the Transmit RT Status Mask, and bits 16–31 contain the Receive RT Status Mask. The contents of this register are ANDed with the returned status word(s) from BC Control Block offset 8_{10} when bit 12 of a BC Control Word is "1". A nonzero result generates an interrupt.	0	6-4

32-Bit Word Offset (hex)	Byte Offset	Name	Description	Default	Page Ref.
1E	78h	BCSCHDLY	Contains the delay time for a scheduled BC delay block. Program bits 0–15 of this location with the delay time, in microseconds, to be used for a BC delay block (BC transfer type 4) which is scheduled. This value applies to all scheduled BC delay blocks and should be programmed prior to execution of a BC chain.	0	6-5
			Sequential Monitor Registers		
0F	3Ch	SMFPTR	Pointer to the first sequential monitor buffer.	0	8-6
10	40h	SMCPTR	Pointer to the current sequential monitor buffer - filled by ABI-V5. Updated by the firmware.	0	8-6
11	44h	SMLPTR	Pointer to the last sequential monitor buffer to be filled. Updated by the firmware.	0	8-6
12	48h	MONCTRL	Sequential Monitor Control Register: bit 0 disables the capture of spurious/error data, bit 1 enables sequential monitoring of all RTs and subaddresses, and bit 2 generates an interrupt on error.	0	8-6
13	4Ch	SMBCNT	Sequential Monitor Buffer Count. Preload from host to assign a number value to each buffer. Incremented on each buffer swap.	0	8-6
			General System Registers		
14	50h	DATABASE	RT, BC data buffer base address	0	5-19
15	54h	HDMACTRL	Host DMA Request Control Word	0	5-20
16	58h	HDMAABI	Host DMA Request ABI-V5	0	5-20
17	5Ch	HDMAVME	Host DMA Request VME Address	0	5-20
18	60h	HDMAWCV	Host DMA Request Word Count/Vector	0	5-22
19	64h	XDATPOL	External Data Polarity	0	5-22
1A	68h	BCXTSEL	BC External Trigger Out Select	0	5-23
1B	6Ch	IRCVMD	Illegal Receive Mode Code Mask. These registers are used to specify which receive mode codes are illegal.	FFCDFFFFh	5-22
1C	70h	ITRXMD	Illegal Transmit Mode Code Mask. These registers are used to specify which transmit mode codes are illegal.	FFF2FE00h	5-22

32-Bit Word Offset (hex)	Byte Offset	Name	Description	Default	Page Ref.
1Fh	7C	DBCTRL	Database Control Register (see the description following this table)	0	5-19
20–FFh	78–3FC		Reserved		

* Programmed with default value by firmware upon power-up.

5.2.1 RESP(RO) Register

The RESP or Firmware Response register provides an indication of channel status to the host. This register is read only and may assume the following values:

- ***0: Channel Not Initialized.*** A value of zero in the RESP register indicates that the channel has not been initialized. Channel initialization occurs with an Initialize Channel Host Command. A channel which has not been initialized is also disabled from processing 1553 bus traffic.
- ***1: Channel Initialized.*** A value of one in the RESP register indicates that the channel is initialized but disabled from processing 1553 bus traffic.
- ***Changing Value.*** A changing value in the RESP register indicates that the channel is initialized and enabled for processing 1553 bus traffic.



Cross Reference: Refer to the section *Host Commands* on 5-26 for details on initializing, enabling, and disabling a channel.

5.2.2 DATABASE (RT, BC Data Buffer Base Address Control Register)

The DATABASE control register contains the base address in ABI-V5 memory of RT and BC data buffers. This control register serves as the base address of a data pool. All pointers that are associated with BC and RT buffers serve as offsets from this data pool base address to the actual data buffer. The firmware will add all pointers in the RT Definition Table and BC Control Block pointer to the contents of this control register to determine the actual physical address of the data buffer or control block. The host may change the current Data Base by issuing a "Change Data Base" host command to effect a complete data structure change in ABI-V5 memory. Additionally, the firmware provides a provision for changing the Data Base address during BC operation.



Cross Reference: *Chapter 6: BC Simulation* contains details of this feature for changing the Data Base address.

If the user chooses not to use the Data Base feature, this control register will contain a value of zero. In this case, the pointers read by the firmware will point to actual physical memory locations. The firmware will adjust the addresses of DMA transfers for RT buffers and BC data structures by the value of this register.



Note: The firmware will add the contents of the DATABASE control register to pointers associated only with BC and RT operation. Interrupt queue and sequential monitor buffer pointers will not be affected by this register. Also, the DATABASE control register can be changed only through a "Change Data Base" host command or through the "New Base" provision of BC operation.

5.2.3 DBCTRL: Database Control Register

- ***Bit 0: Reserved.*** This bit is reserved for future use.
- ***Bit 1: Apply Database to All Pointers.*** The state of this bit determines whether the value in the DATABASE control register is applied to both control and data buffer pointer/offset values and BC control block pointers, or to just the data buffer pointer/offset values. Set this bit to "0" to apply the DATABASE value only to the data buffer pointer/offset values. This is the default setting. Set this bit to "1" to apply the DATABASE value to both the control buffer and the data buffer pointer/offset values.
- ***Bits 2–31: Reserved.*** Bits 2–31 are reserved for future use.

5.2.4 Host DMA Request Registers

The following registers are used only in conjunction with the Host DMA Request host command (0Ch). Program values in these registers prior to issuing the host command.

HDMACTRL Offset 15h. This is the key register for host DMA request operations. The bits of this register have the same function as those for the HOST DMA Control Register. See Table 5.1.3 for details.

HDMAABI Offset 16h. This register defines the ABI-V5 source address for DMA operation to the VMEbus. Only bits 0–23 are used; set bits 24–31 to zero. The source address can be any ABI-V5 local address. (ABI-V5 addresses are in words, not bytes.)

HDMAVME Offset 17h. This register defines the VMEbus destination address word (24 or 32 bits as defined by bit 4 of the Master CSR). VMEbus memory, as with ABI-V5

memory, is big-endian oriented. Most VMEbus processor or reflective memory modules are byte oriented; thus, D32 or D64 transfers must start on a four-byte or eight-byte boundary.

HDMAWCV Offset 18h.

- ***Bits 0–15: Word Count.*** Bits 0–15 of this register hold the word count, i.e., the number of transfers to be performed over the VMEbus. The count value is for either D64 or D32 words, as designated by bit 5 of the Master CSR word. For example, if the word count is 128 and bit 5 is set to "1" (selecting D64), 128 64-bit words will be transferred. Thus, the length count is not a byte count, but a VMEbus transfer count. The minimum transfer value is zero and the maximum is 65535.
- ***Bits 16–23: Interrupt Vector.*** Bits 16–23 of this register contain the interrupt vector used when the master interface interrupts the VMEbus. Programming this vector is optional; if these bits are zero, the value contained in INTV_DMA will be used as the interrupt vector.

5.2.5 Illegal Mode Codes

Illegal Receive Mode Code Mask (IRCVMD), Illegal Transmit Mode Code Mask (ITRXMD)

Offsets 1Bh, 1Ch.These registers are used to specify which receive and transmit mode codes are illegal. Each Illegal Mode Code Mask consists of a bit field organized by mode code (bit 0 = MD0, bit 1 = MD1, etc.) If a bit is "1", the corresponding mode code is illegal. When a mode code is illegal, the RT responds by setting the Message Error (ME) bit to "1" in the status response word. If the command is a Transmit Mode Code with Associated Data, the RT suppresses transmission of data.

During firmware startup, IRCVMD is defaulted to FFCD FFFFh, and ITRXMD is defaulted to FFF2 FE00h.

5.2.6 XDATPOL: External Data Polarity

This register is used to set the polarity for each of the external data lines in the auxiliary connector. The auxiliary data lines are used for generating external triggers during Bus Controller operation. Only bits 0–7 are used in this register. Bit 0 corresponds to External Data 1, bit 1 corresponds to External Data 2, etc. Refer to Table 2.4.6 on page 10 for auxiliary pin locations. The contents of this register must be established prior to issuing an "Initialize Channel" host command.

If a bit in this register is set to "0", the external data line corresponding to that bit will normally be held low. If a trigger condition arises and the external data line has been programmed as the output trigger, the external data line will be driven high, causing a positive-going edge. The line will remain high for a period of one message and will then go low.

Likewise, if a bit in this register is set to "1", the corresponding external data line

will normally be held high. A trigger on this line will cause a negative going edge. The line will be held low for a period of one message and will then go high.



Note: When using a dual-channel ABI-V5, XDATPOL must be programmed identically in both channels. It is the programmer's responsibility to assure that this is the case.

5.2.7 BCXTSEL: BC External Trigger Out Select

This register is used to select the External Data line in the auxiliary connector for the BC Output Trigger to Front Panel function (bit 2 of a BC Control Word). Only the values 0–7 are valid. A value of "0" corresponds to External Data 1, a value of "1" corresponds to External Data 2, etc. When a BC Control block is encountered with the "Output Trigger to Front Panel" bit (bit 2) set to "1", the contents of this register will be read to determine the external data line on the auxiliary connector on which the trigger out will occur. The generated trigger will have the polarity programmed in XDATPOL.

5.3 Error Table

The ABI-V5 error table is located at offsets 100h–1DBh from the control Base Address of a channel. The firmware uses this table to reflect various system level errors to the host that have occurred during ABI-V5 operation. Each location in the error table may contain a single value which provides either additional information concerning the error or a pointer to a buffer where an error was detected. If a bit value is stored in a table location, it will reflect the OR'd sum of all the errors which occurred. Possible error codes are listed in Table 5.3.1.

Table 5.3.1: Error Table—D32 Access

32-bit Word Offset	Byte Offset	Contents	Description
ABI-V5 Startup Error			
100	400	0001	No daughter boards detected on startup
		0002	Channel #1 daughterboard present but not responding
		0004	Channel #2 daughterboard present but not responding
		0008	Firmware configuration error between instruction RAM and data RAM files
Host Command Error			
101	404	0001	Invalid host command
		0002	Invalid host command parameter
		0004	Channel not disabled; command cannot execute
		0008	Channel not initialized; command cannot execute
		0010	RT definition pointer null; command cannot execute
		**Note**: Bits 16–31 of a Host Command Error word contain the Host Command which caused the most recent error.	
Sequential Monitor Buffer Length Error			
102	408	SM Buffer Pointer	The indicated sequential monitor buffer became active but contains an invalid buffer length. The sequential monitor buffer must have a programmed length of no less than 36 words.
DMA Queue Full			
103	40C	Buffer Pointer	A DMA transfer was enabled in the indicated buffer but the DMA queue is full.
RT Definition Pointer Table Error			
104	410	RT Number	A message was received for the indicated RT which has been turned on in the RT Status Response Table but a null pointer was encountered in the RT Definition Pointer Table for the RT.

32-bit Word Offset	Byte Offset	Contents	Description
RT Data Buffer Pointer Error			
105	414	Bit / Description 0–15 Cmd word 16 Mon All SAs 17 Mon RCV SA 18 Mon TRX SA 19 TRX SA 20–31 Reserved	The indicated RT Control Buffer contains a NULL data buffer pointer/offset. Data cannot be written to or read from the data buffer associated with this RT Control Buffer. Bits 16–19 specify the data buffer type and are mutually exclusive.  Note: Bits 0–4 of the command word are always 0.
DMA Error			
106	418	Pointer to DMA CNTL word	An error occurred on the DMA operation pointed to by the value contained in this error table.
Interrupt Error			
107	41C	Interrupt Bit field	An interrupt needs to be generated for the indicated Interrupt bit field but Interrupt queues have not been allocated.
BC Data Buffer Pointer Error			
108	420	BC Control Block Pointer	The indicated BC Control Block contains a NULL data buffer pointer/offset. Data cannot be written to or read from the data buffer associated with this BC Control Block.

5.4 Last Error Message Buffer

Beginning at offset 1DCh from the control base address of a channel is the last error message buffer. The last error message buffer contains the last message which occurred with an error.

Table 5.4.1: RT Control and Definition Table Offsets

32-Bit Word Offset	Byte Offset	Name	Description
01DC–01FF	0770–07FC	Last Error Message Buffer	Contains the last message of the channel which contains an error.

5.5 RT Control and Definition Tables

Beginning at offset 200h from the control base address of a channel are the RT control and definition tables. The RT Status Response Table and RT Definition tables define how RTs will respond on the bus to command words and where to place data that has been received from the bus. Table 5.5.1 outlines this control area.

Table 5.5.1: RT Control and Definition Table Offsets

32-Bit Word Offset	Byte Offset	Name	Description
0200–023F	0800–08FC	RT Status Response Table	Defines the status response word and status response time for each simulated RT.
0240–025F	0900–097C	RT Definition Pointer Table	A 32-word table, arranged by RT, where each entry points to an RT definition table. An RT definition table provides pointers to RT data buffers for simulation and monitoring as well as SA filtering and interrupt generation.



Cross Reference: *Chapter 7: RT Simulation/RT Monitoring* details each of the RT Status Response and Definition table elements.

5.6 Host Commands

The host command registers permit the user to issue real-time, 1553 processing commands to an ABI-V5 channel. As shown in Table 5.2.1 on page 16, host command registers HOSTCMD1, HOSTCMD2, and HCRESP are located at offsets (from the control base address of a channel) of 0000h, 0001h, and 0002h respectively.

Host commands are required to load user-defined data structures into internal ABI-V5 registers, to enable 1553 bus processing, to start or stop BC simulation, or to process the interrupt queue. Whenever host data structures are programmed, one or more host commands must be issued to notify the ABI-V5 firmware that new values exist.

As shown in Table 5.6.1, host commands contain one or two parameters. Parameter 1 *always* contains the command word, identifying the type of action. If required by a command, parameter 2 contains the data word.

Note: If 16-bit accesses are implemented, bits 16–31 of the command must be written first, followed by bits 0–15.

Table 5.6.1: Host Command Register Commands

Parameter 1 Command Word	Parameter 2 Data Word	Description
00000001	N/A	Initialize Channel
00000002	N/A	Enable Channel - Enables 1553 bus processing
00000003	N/A	Disable Channel - Disables 1553 bus processing
00000004	N/A	Stop BC Chain
00000005	BC Pointer	Start BC Chain
00000006	N/A	Update Global BC Control Information
00000007	RT Number	Update Single RT Status Response Table Entry
00000008	Action	Sequential Monitor Buffers
00000009	Action	Host Processing Interrupt Queue
0000000A	Data Base	Change Data Base
0000000B	Select	1553 Stub Select
0000000C	N/A	Host DMA Request
0000000D	RT Number	Update RT Illegal Subaddress. This applies to the Illegal Command option only.
0000000E	N/A	Update Illegal Mode Code. This applies to the Illegal Command option only.

5.6.1 Setting Host Command (With or Without Parameters)

1. Clear HCRESP.
2. If applicable, write the parameter to HOSTCMD2.

Note: The host should not write a value to HOSTCMD2 for host commands without a parameter unless that value is 0. If a nonzero value is written to HOSTCMD2, unpredictable results may occur.

3. Write the command to HOSTCMD1.
4. The firmware reads HOSTCMD1 and writes a “02h” to HCRESP, indicating the command is being processed.
5. The command written in HOSTCMD1 will be validated. If the command value is invalid, the firmware writes a “03h” to HCRESP, the appropriate entry in the

error table is updated, and host command processing ceases at this point.

6. If applicable, the firmware reads HOSTCMD2, clears HOSTCMD2, and validates the value read from HOSTCMD2. If the parameter is invalid, a "03h" is written to HCRESP and the appropriate entry in the error table is updated. Host command processing ceases at this point.

7. The firmware takes the appropriate action as indicated by the host command in HOSTCMD1, and, if applicable, the parameter in HOSTCMD2.

8. The firmware indicates completion of the command by writing a "01h" to HCRESP. If an error occurs, the firmware writes a "03h" to HCRESP and an appropriate entry in the error table is updated.

5.7 Host Command Descriptions

Initialize Channel HOSTCMD1 = 00000001h. Issue this command after initial memory allocation and setup is complete. This command causes the ABI-V5 channel to initialize internal registers according to current memory allocation. This command is processed only if the channel is disabled. The firmware issues a host command error to offset 101h if the channel is enabled. If a pointer is placed in SMFPTR, the firmware validates the sequential monitor buffer length of the buffer pointed at by SMFPTR. If the buffer length is at least 36 words long, the minimum required buffer length, the contents of SMFPTR are copied to SMCPTR, and sequential monitoring is enabled. If the buffer length is invalid, the firmware writes the pointer to the buffer in the error table at location 102h, disabling sequential monitoring but completing initialization. If SMFPTR contains a zero, sequential monitoring will not be enabled. Once the channel has been properly initialized, the firmware writes a 01h to RESP, indicating an initialized channel.

Enable Channel HOSTCMD1 = 00000002h. Issue this command to enable 1553 bus processing by the ABI-V5 channel. When the channel is enable, the value in RESP will constantly change.

Disable Channel HOSTCMD1 = 00000003h. Issue this command to disable 1553 bus processing by the ABI-V5 channel. Once the channel is disabled, control register RESP stops changing and contains the value of 1, indicating that the channel is initialized but disabled.

Stop BC Chain HOSTCMD1 = 00000004h. Issue this command to stop execution of a currently running BC chain. This command causes the ABI-V5 channel to stop processing the BC list.

Start BC Chain HOSTCMD1 = 00000005h. Allocate and initialize a BC list in user memory of the ABI-V5 and then issue this command to start a BC list. After initialization, write the pointer to the beginning of the BC list to HOSTCMD2 and then write the Start BC Chain command to HOSTCMD1. The firmware reads the value from

HOSTCMD2, clears HOSTCMD2, and then writes the value contained in HOSTCMD2 to BCCPTR. If the BC command block pointed to by the value contained in HOSTCMD2 has either the Minor Frame Start designator or Wait for External Trigger bit set, the pointer is not written to BCCPTR until the corresponding event has occurred.

Update Global BC Control Information

HOSTCMD1 = 00000006h. Issue this command to change the BC intermessage gap value (BCIMGP), BC retry information (BCRETRY), the Maximum Allowable Status Response Time (RSPGPA), and the Frame Time (FRMTMR). First reprogram the registers and then issue this host command to inform the ABI-V5 firmware of the change.

Update RT Status Response Table Entry

HOSTCMD1 = 00000007h. Make any change necessary changes to a single RT's Status Response Table (RT Status Word, RT Status Gap, RT Protocol, etc.) and then issue this command to effect an update of the table. Upon completion of the changes, write the RT number to HOSTCMD2, then the command word 00000007h to HOSTCMD1.

Sequential Monitor Buffers

HOSTCMD1 = 00000008h. The host uses this command to externally control the sequential monitor buffers. Two different actions may occur with this command as outlined below:

- ***Swap Sequential Monitor Buffers (HOSTCMD2 = 0).*** Issuing this host command with HOSTCMD2 = 0 causes the sequential monitor buffers to swap. The firmware copies the contents of SMCPTR to SMLPTR. The firmware then reads the contents of the "next" pointer of the current buffer. If the "next" pointer is "0", sequential monitoring ceases. If not, the buffer length is read and validated that it meets the minimum required length of 36 words. If the length does not meet the minimum programming length, error table location 102h from the control base address of the channel will contain a pointer to the buffer in error. Otherwise, the pointer is copied to SMCPTR, making it the currently active buffer. Generation of interrupts on buffer swap is controlled through the sequential monitor buffer swap interrupt bit in the buffer itself.
- ***Sequential Monitor Reset (HOSTCMD2 = 1).*** Issuing this host command with HOSTCMD2 =1 causes a reset of sequential monitoring. This command can be issued to restart sequential monitoring if monitoring has halted due to encountering a NULL "next" pointer or an invalid sequential monitor buffer length. The actions taken for a sequential monitor reset are identical to those actions taken during channel initialization for the sequential monitor. Note that sequential monitoring will be reset and begin filling the buffers from the head of the chain of sequential monitor buffers (the buffer pointed at by SMFPTR).

Host Processing Interrupt Queue

HOSTCMD1 = 00000009h. Issue this command to initiate host processing of the interrupt queue. To begin host processing, write FFFFFFFFh to HOSTCMD2 followed by command 00000009h to HOSTCMD1. This command causes the ABI-V5 firmware to complete any interrupt queue entries in progress to IQPTR1. The firmware then acknowledges to the host by writing a "1" to HCRESP. The firmware clears the count contained in IQCNT2 and writes any subsequent interrupts to the queue pointed at by IQPTR2. After reading a "1" in HCRESP, process the interrupts contained in IQPTR1. When the queue is completely processed, write 00000001h to HOSTCMD2 followed by the command 00000009h to HOSTCMD1. The firmware then performs an interrupt queue buffer swap, copying the pointer in IQPTR1 to IQPTR2, and the pointer in IQPTR2 to IQPTR1. Upon completion, the firmware again acknowledges by writing a "1" to HCRESP.

Change Data Base

HOSTCMD1 = 0000000Ah. Issue this command to change the current data pool base address. This address serves as the base address from which all BC control blocks and RT pointers are added to locate the corresponding buffer. To change the Data Base address contained in the control register DATABASE, write the new base address to HOSTCMD2 followed by command 0000000Ah to HOSTCMD1.

1553 Stub Select

HOSTCMD1 = 0000000Bh. Issue this command to change the 1553 bus stub configuration. To select long stub (ABI-V5 default), first write a "0" to HOSTCMD2 followed by command 0000000Bh to HOSTCMD1. To select

short stub, write a nonzero value to HOSTCMD2 followed by command 0000000Bh to HOSTCMD1.

Host DMA Request

HOSTCMD1 = 0000000Ch. Issue this command to initiate a DMA transfer when firmware DMA transfers are also being used (initiated from the BC, RT, or Monitor data structures). Four software control registers must be programmed prior to issuing this command: HDMACTRL, HDMAABI, HDMAVME, and HDMAWCV. Refer to the descriptions of these registers earlier in this chapter. Note that the value written to HDMACTRL must have the start bit (bit 0) set to "1" to enable the DMA transfer to occur. After programming these registers, write a 0000000Ch to HOSTCMD1 to queue the host requested DMA transfer.



Note: This host command is provided for use when both firmware and host initiated DMA transfers are required. When a host DMA transfer is requested, the transfer is placed in the DMA transfer queue managed by the firmware. The DMA transfer will occur after all DMA transfers in the queue preceding this request have successfully completed.

Update Illegal Mode Codes Host Command

HOSTCMD1 = 0000000Eh. This host command is available only with ABI-V5 firmware that includes the Illegal Command option. This host command must be issued in order for any changes to the Illegal Mode Code masks to take effect.

1. Make any change necessary to the channel's Illegal Receive (IRCVMD) and Illegal Transmit (ITRXMD) mode code masks.

2. Upon completion of the changes, write the command word 0000000Eh to HOSTCMD1.

   The new illegal mode code masks will then become active.



Note: Only legal mode codes can be made illegal. Illegal mode codes, as specified by MIL-STD-1553B, cannot be made illegal.

6: BC Simulation

The ABI-V5 is capable of simultaneous simulation of a BC and 31 RTs while monitoring one or two dual redundant 1553 channels. BC simulation for a channel is configured through linked control blocks. This data structure provides powerful options for accurately simulating BC minor and major frame sequences. Also provided are two 23-bit, 1-µs minor frame timers (one for each channel).



Note: Prior to reading this chapter, be familiar with the information discussed in *Chapter 3: 1553 Operations—Module Overview*, *Chapter 4: Module Startup/Test*, and *Chapter 5: Host Commands/Control Registers*.

The first part of this chapter reviews minimum programming requirements and key control registers. The remaining pages detail the key data structure for BC Simulation: the BC Control Block.



Note: The terms offset, pointer, and address are used interchangeably when referring to SBS device memory. The device memory is accessed by an offset from the device's base memory address.



Note: Addresses listed in this chapter are 32-bit longwords.

6.1 Operational Steps for BC Simulation

The following steps give a summary of the procedures required to operate the ABI-V5 as a Bus Controller:

1. Load the Instruction Code per *Chapter 4: Module Startup/Test*.
2. If the default values are not satisfactory, program the following control registers: BCIMGP, BCRETRY, RSPGPA, and FRMTMR.
3. Load the BC control chain(s).
4. Issue the "Initialize Channel" command (Host Command 1). If the ABI-V5 is also performing RT simulation or sequential monitoring, define these data

structures prior to issuing Host Command 1. If interrupts are being used, allocate the interrupt queues.

5. Issue the "Enable Channel " command (Host Command 2).
6. Issue the "Start BC Chain " command (Host Command 5).
7. To change the values of BCIMGP, BCRETRY, RSPGPA, and FRMTMR during 1553 processing, simply program the registers with new values and then issue an "Update Global BC Control Information " command (Host Command 6).

6.2 Software Control Registers

Eight registers in the channel control table are associated with bus controller operation: BCCPTR, BCLPTR, BCIMGP, BCRETRY, RSPGPA, FRMTMR, BCSMSK, and BCSCHDLY.



Note: Control registers are given as offsets from the base control address of the channel; add 0800 to the VMEbus module address for channel 1, and 1000 to the address for channel 2.

BCCPTR Control Table Offset 09h, Updated by Firmware. Control register BCCPTR contains a pointer to the current BC command block in operation. This register always contains the offset of the current BC Control Block. Use this register to track BC chain program execution.

BCLPTR Control Table Offset 0Ah, Updated by Firmware. Control register BCLPTR contains a pointer to the last BC command block processed by the ABI-V5. This register always contains the offset of the last BC Control Block that was executed.

BCIMGP Control Table Offset 0Bh. Control register BCIMGP contains the BC intermessage gap used during bus controller operation which applies to all BC control blocks. Only bits 0–9 of this register are used. The register contains the number of tics (where each tic = 250 ns) that will yield the desired gap time. The ABI-V5 channel waits this amount of time after the completion of one message before issuing the first command word of the next BC control block.

The value entered into this register is obtained using the following formula:

$$BCIMGP = ((x - 2) * 4) - 3$$

where x is the desired 1553B intermessage gap in microseconds.

After initialization, the host must issue an "Update Global BC Information" host

command to effect a change in value of this register.



Cross Reference: Refer to *Chapter 5: Host Commands/Control Registers* for details on this host command.

BCRETRY Control Table Offset 0Ch. Control register BCRETRY contains the retry count and retry bus pattern for BC retries. If an error occurs in a BC block in which the Enable BC Retry bit is set to "1", the ABI-V5 channel uses the contents of this register as a count and bus pattern, reissuing the message until either the message completes without an error or the count is exhausted. Figure 6.2.1 shows the format of this control register.

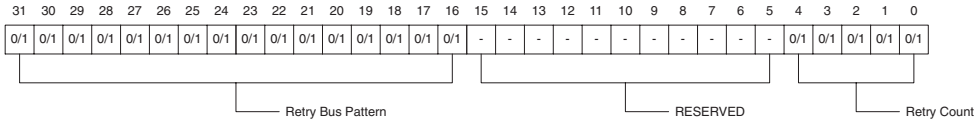


Figure 6.2.1: BCRETRY Control Register

- ***Bits 0–4:Retry Count.*** The retry count specifies the maximum number of retries the BC will attempt if there is an error (maximum of 16).
- ***Bits 16–31:Retry Bus Pattern.*** The Retry Bus Pattern defines which bus (A or B) the BC will use to retry the message. Each bit pertains to a particular retried message; bit 16 applies to the first retry of the message, bit 17 applies to the second retry of the message, etc. This pattern continues up to the bit that applies to the number of retry messages defined by Retry Count (bits 0–4). If a bit is "0", the associated retry of the message will be on the same bus as the original message. If a bit is "1", the associated retry of the message will occur on the alternate bus. For example, suppose the BC transmits a message on the primary bus (Bus A) and an error occurs. If the Enable BC Retries bit is "1" and the associated retry bus pattern bit is a "1", the retry of the message will be on the secondary bus (Bus B).

RSPGPA Control Table Offset 0Dh. This register contains the maximum allowable status response gap time (which is the maximum time the ABI-V5 will wait before indicating a status response time-out). Only bits 0–9 of this register are used. The register contains the number of tics (where each tic = 250 ns) that will yield the desired maximum allowable status response gap. The value entered into this register is obtained using the following formula:

$$(x - 2) * 4$$

where x is the maximum 1553B status gap in microseconds.

Note: The value programmed in this register can only be changed when the channel is disabled. To change the value, issue a Disable Channel Host Command, program RSPGPA with the desired value and issue an Enable Channel Host Command.

FRMTMR Control Table Offset 0Eh. This register contains the minor frame time for BC chain execution. The programmed time is in 1-µs clock tics and occupies bits 0–22 as shown in Figure 6.2.2.

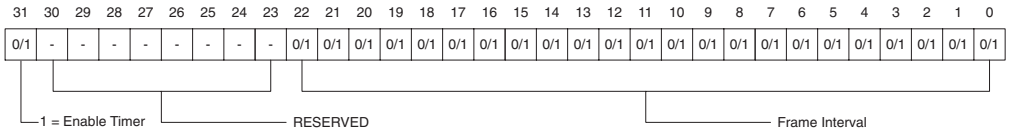


Figure 6.2.2: BC Frame Timer Register

Bit 31 serves as an enable: when bit 31 = 1, the frame timer is enabled with the programmed time; when bit 31 = 0, the frame timer is disabled. During BC execution, if the firmware encounters a Minor Frame Start Designator (see bit 4 of the BC Control Word) for the first time, it starts the Frame timer and transmits the BC message containing the Minor Frame Start Designator. If the firmware encounters subsequent BC control blocks with the Minor Frame Start Designator, it waits until the programmed time has expired before transmitting the BC block.

Note: If the first Minor Frame Start Designator encountered by the firmware occurs in a BC block after the first BC block in a chain and the programmed time has expired, a BC intermessage gap of 30–35 µs will occur between the preceding BC message and the message containing the Minor Frame Start Designator. This condition will only occur once as the Minor Frame timing initializes. Subsequent BC intermessage gaps will follow the programmed value.

Note: When updating BCIMGP, BCRETRY, RSPGPA, and FRMTMR, issue an "Update Global BC Control Information" host command. This host command *always* updates these four values. If changing more than one of these values, update as many of these four register locations as possible *before* issuing an "Update Global BC Control Information" host command. This will minimize the amount of firmware processing and maximize the efficiency of the host command.

BCSMSK Control Table Offset 1Dh. Control register BCSMSK contains the 32-bit Status Word Mask. Bits 0–15 contain the Transmit RT Status Mask and bits 16–31 contain the Receive RT Status Mask. The contents of this register are ANDed with

the returned status word(s) from BC Control Block offset 8 when bit 12 of a BC Control Word is “1”. A nonzero result generates an interrupt.

BCSCHDLY Control Table Offset 1Eh. Control register BCSCHDLY contains the delay time for a scheduled BC delay block. Program bits 0–15 of this location with the delay time, in microseconds, to be used for a BC delay block (BC transfer type 4) which is scheduled. This value applies to all scheduled BC delay blocks and should be programmed prior to execution of a BC chain.

6.3 BC Control Blocks

The BC Control Block, pictured in Figure 6.3.1, provides the backbone to the Bus Controller operation. A BC list consists of a possible linked list of BC Control Blocks. The BC Control Block consists of the following:

- Link pointer words
- DMA control words
- Database control word
- Status, time, and gap time words (updated when message received)
- BC control word
- Error injection/transfer type words
- Transmit command words
- Receive or transmit data words

Below are the definitions for each word in a BC Control Block.

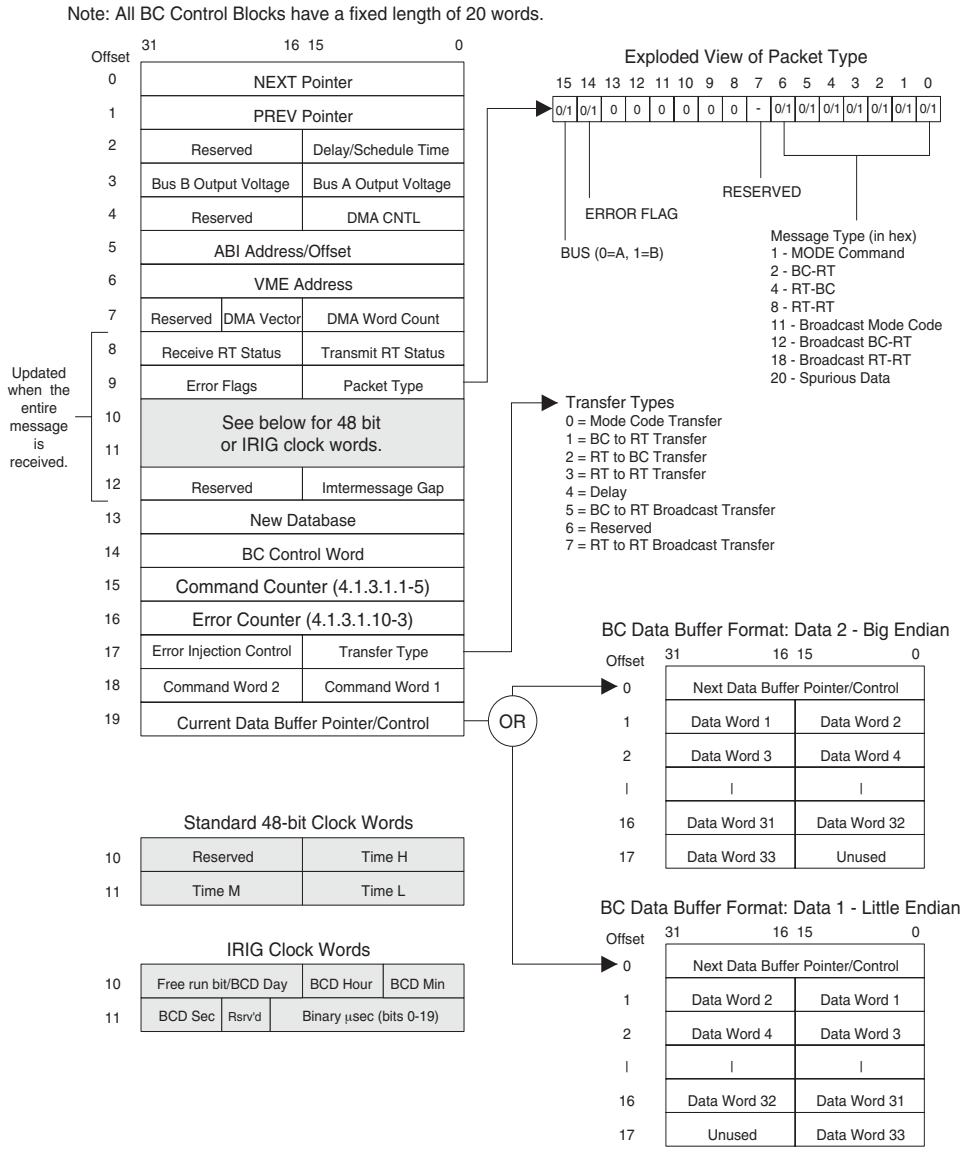


Figure 6.3.1: BC Control Block

NEXT Pointer, PREV Pointer

Control Block Offsets 0, 1. BC Control Block allocation consists of a linked list of control blocks. Allocate these blocks from the free memory region beginning at 1800h. As shown in Figure 6.3.1, each control block contains a "Next" (NEXT) pointer and an optional "Previous" (PREV) pointer. Use the PREV Pointer to construct a doubly-linked list of BC Control Blocks to facilitate host processing. The firmware never accesses the PREV Pointer location. A null value encountered in the NEXT Pointer location will cause BC execution to halt.

Schedule Time, Delay Time, or Schedule Delay

Control Block Offset 2. Use this control block offset to set:

- Delay Time

- Schedule Time
- Schedule Delay

Offset 2 will contain one of two values in the lower bits of the word. How the contents of offset 2 are interpreted is dependent on the contents of Bit 15 of control block offset 17. These two registers work in conjunction with each other and must be set accordingly.

There are eight transfer types that can be placed into Offset 17. These values comprise a 3 bit combination that are placed into bits 0, 1 and 2. A list of transfer types is below. If a Delay transfer type is used (4h), Bit 15, Offset 17 will be cleared and indicates that the lower 16 bits of Offset 2 contain a delay time in microseconds. The minimum delay time allowed is 25 µs. Any other type of transfer type will set Bit 15, Offset 17. When Bit 15,Offset 17 is set, the lower 19 bits of Offset 2 will contain the schedule time in µs. Additionally, a scheduled delay time can be implemented by using a Delay Transfer Type (4h) as above, but setting bit 15 instead of clearing it. This indicates that the value in the lower 19 bits of Offset 2 are the schedule time and the delay time is then stored in the register BCSCHDLY. See Figure 6.3.2.

Transfer Types

- 0MODE
- 1BCRT
- 2RTBC
- 3RTRT
- 4DELAY
- 5BCRT (Broadcast Transfer)
- 6RESERVED
- 7RTRT (Broadcast Transfer)

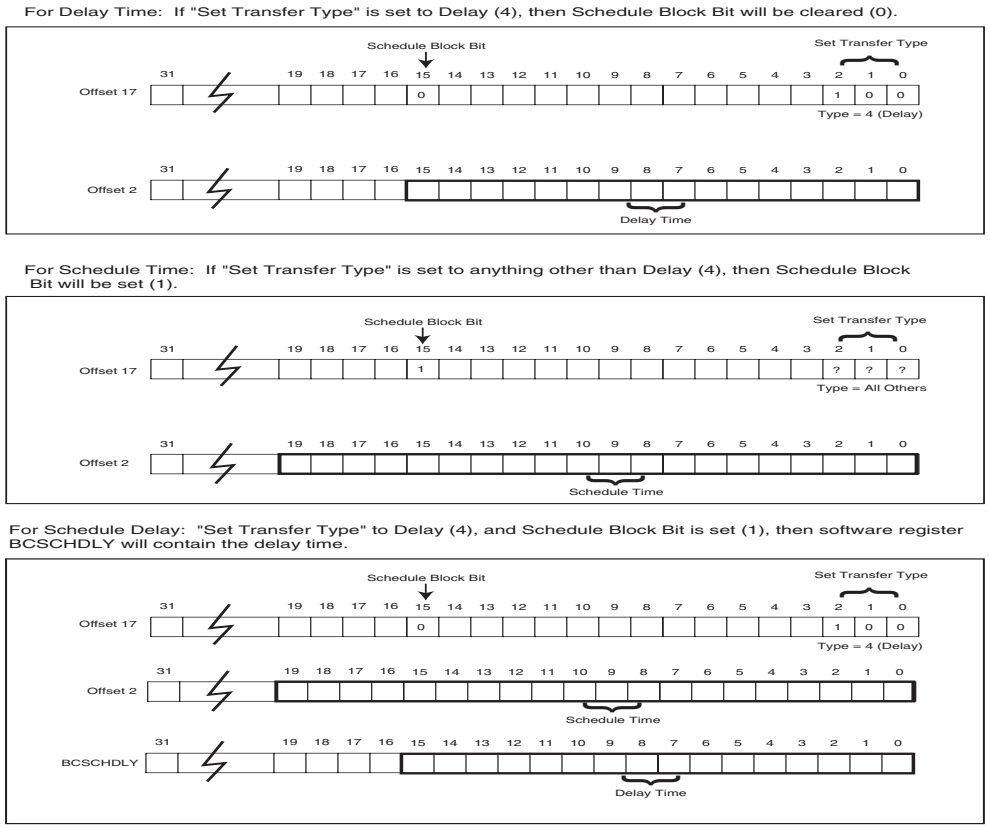


Figure 6.3.2: Set Delay Time, Schedule Time, or Schedule Delay

The 20-bit Schedule Time quantity is used to schedule a BC message in a minor frame. Scheduling a BC message causes the firmware to wait for the programmed time, from the beginning of the minor frame, to elapse before transmitting the message on the bus. BC Scheduling is available only if the Bus Controller is using the minor frame feature.



Cross Reference: Refer to the subsection *FRMTMR* on 6-4 for information on how to enable the BC Frame Timer.

To determine the value to place in this location for a desired schedule time, use the following formula:

```
BC Frame Time - Desired Schedule Time,
```

where *BC Frame Time* is the value contained in the FRMTMR control register and *Desired Schedule Time* is the time in microseconds from the beginning of the minor frame.



Note: BC Scheduling is available only with the standard 48-bit, 1-µs timer option.

Reserved

Control Block Offset 3. These two words are reserved for future use.

DMA CNTL, ABI ADDR, VME ADDR, WORD CNT/VECTOR

Control Block Offsets 4, 5, 6, 7. These four words control DMA operations for writing ABI-V5 memory to VMEbus memory at the end of a BC control block.



Cross Reference: These words are defined in *Chapter 5: Host Commands/Control Registers*.

Receive RT Status/Transmit RT Status

Control Block Offset 8. Offset 8 of the BC Control Block contains the status response word or words, depending on the transfer type, for the message. Bits 0–15 contain the Transmit RT Status word and bits 16–31 contain the Receive RT Status word. This location is updated with the received status regardless of whether an error occurs in the message. This update occurs upon completion of the message.

Error Flags/ Packet Type

Control Block Offset 9. Offset 9 of the BC Control Block contains the packet type word in bits 0–15 and the error flags word in bits 16–31.

- ***Bits 0–15: Packet Type Word.*** The packet type word identifies what type of transfer actually occurred on the bus for the current BC Control block. These 16 bits identify the transfer type and bus. If bit 14 is set to "1", an error occurred in the message. Bit 15 indicates the bus on which the message was received. If bit 15 is equal to "0", the message was received on the primary bus (bus A); if equal to "1", the message was received on the secondary bus (bus B).
- ***Bits 16–31: Error Flags Word.*** Each bit in the error flags word identifies a particular type of error which occurred. Errors can be broken into two categories:
  - Those which occur at a message level
  - Those which occur at a word level

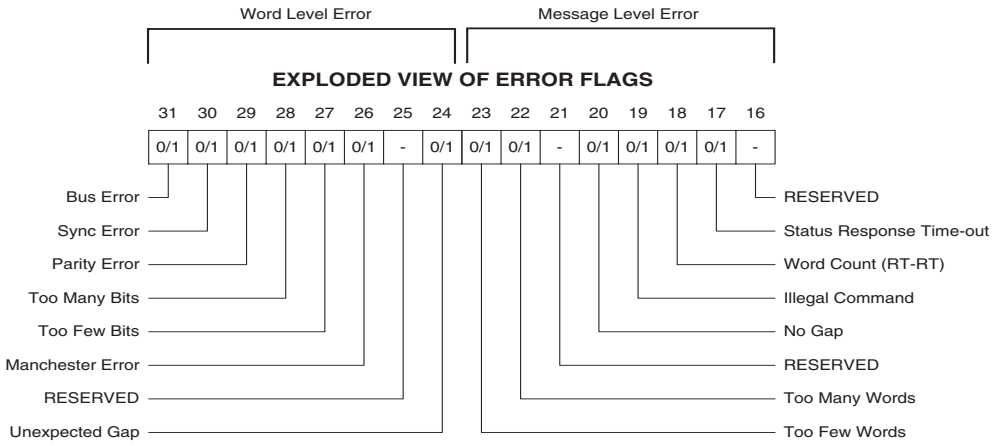


Figure 6.3.3: Error Flags Word

- Bits 16–23 pertain to errors which occur at a message level and bits 24–31 pertain to errors which occur at a word level (see Figure 6.3.3). When an error occurs at a word level, the word in which the error occurred is stored as the last word in the packet. This word is located by the host based on the message type and word count in the packet type word.



Note: When the firmware is copying data to a BC Control Block and buffer, the Packet Type/Error Flags word is the last word written to the buffer. The host may poll this location to determine when the firmware has completed writing data to the buffer. Delay blocks, however, do not include a packet type word. The host may detect completion of a delay block by examining the Message Success bit of the BC Control Word.

- ***Bit 16: Reserved.*** This bit is reserved for future use.
- ***Bit 17: Status Response Time-out.*** This bit indicates that a status response was not received before the time programmed in RSPGPA expired.
- ***Bit 18: Word Count (RT-RT).*** This bit indicates that the word count fields of the two command words in an RT-RT transfer did not match.
- ***Bit 19: Illegal Command.*** This bit indicates that a command word was received which appeared to be a mode code but contained an illegal value in the word count field.
- ***Bit 20: No Gap.*** This bit indicates that a gap was expected on the bus but a contiguous word was received.
- ***Bit 21: Reserved.*** This bit is reserved for future use.
- ***Bit 22: Too Many Words.*** This bit indicates that more data words were received than indicated in the associated command word.
- ***Bit 23: Too Few Words.*** This bit indicates that fewer data words were received than indicated in the associated command word.
- ***Bit 24: Unexpected Gap.*** This bit indicates that a gap was detected on the bus that was not expected according to the current command and transfer type.
- ***Bit 25: Reserved.*** This bit is reserved for future use.
- ***Bit 26: Manchester Error.*** This bit indicates that a Manchester error was detected on a word.
- ***Bit 27: Too Few Bits.*** This bit indicates that fewer than 19 bits were detected on a word before the next valid sync was detected.
- ***Bit 28: Too Many Bits.*** This bit indicates that more than 19 bits were detected on a word before the next valid sync was detected.
- ***Bit 29: Parity Error.*** This bit indicates that a parity error was detected on a word.
- ***Bit 30: Sync Error.*** This bit indicates that a sync error was detected on a word. This error can occur when a command sync is expected but a data sync is received, or when a data sync is expected but a command sync is received.
- ***Bit 31: Bus Error.*** This bit indicates that a word was received on the opposite bus from which the command word or words were received.

Standard V5 TimeH, TimeM/TimeL

Control Block Offsets 10, 11. For the Standard V5, TimeH, TimeM, and Time L are the 48-bit time stamp of the message and indicate the ABI-V5 time in which the first command word of the Control Block was received. Upon transmission and subsequent receipt of the first command word on the bus, the firmware latches the current ABI-V5 time.

BC Control Blocks having the standard IRIG format are structured as shown in Figure 6.3.4.

Note: BC Scheduling is not available with the IRIG time option.

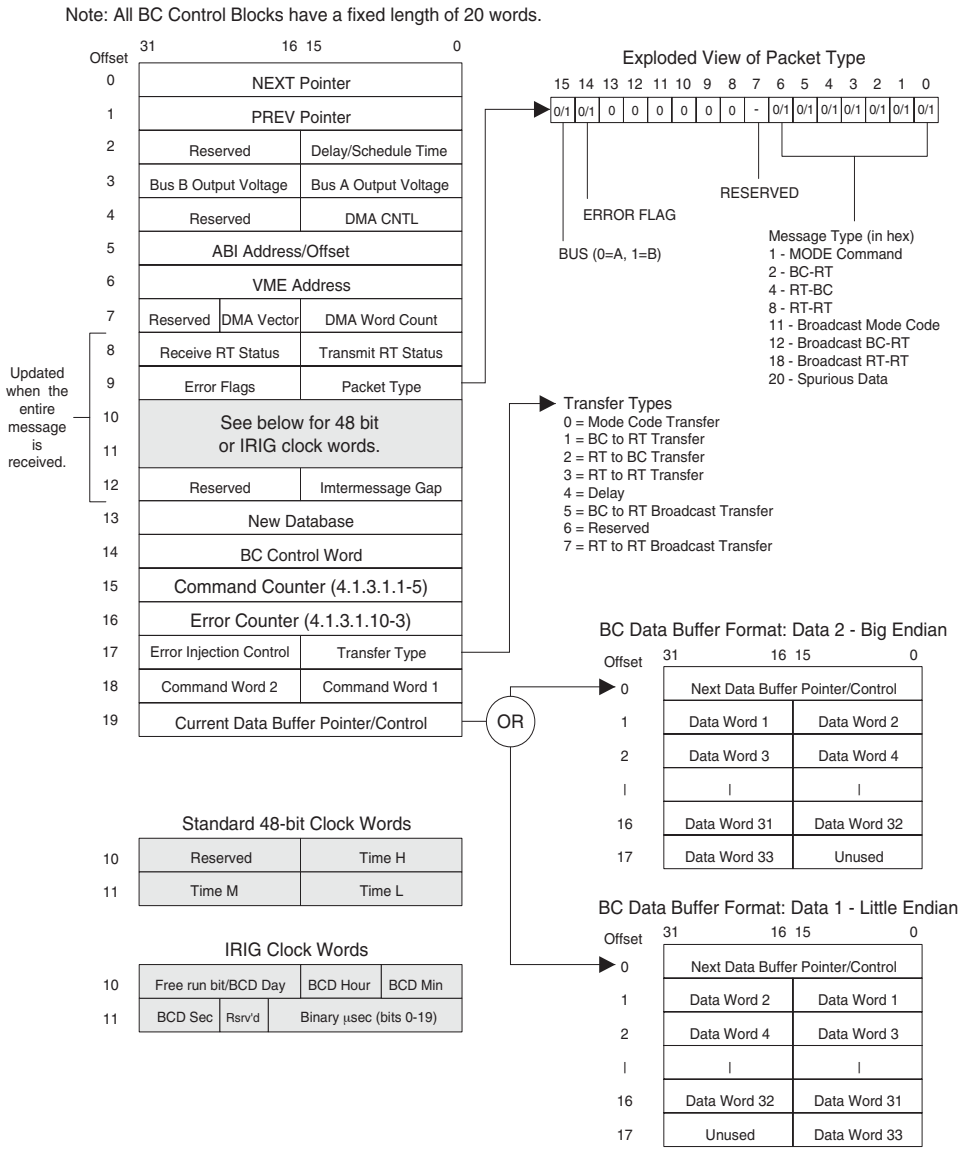


Figure 6.3.4: BC Control Block with Standard and IRIG Time Stamps

V5 IRIG TimeH TimeL

V5 IRIG Control Block Offsets 10, 11. For the V5 IRIG, Control block offsets 10 and 11 contain the time stamp of the message and indicate the IRIG time in which the first command word of the Control Block was received. Table 6.3.1 details each of these words.

Table 6.3.1: BC Control Block Offsets 10 and 11

Bit	Description
Offset 10	
0–3	1s position of BCD Minutes
4–7	10s position of BCD Minutes
8–11	1s position of BCD Hours
12–15	10s position of BCD Hours
16–19	1s position of BCD Days
20–23	10s position of BCD Days
24–27	100s position of BCD Days
28–30	Reserved
31	0 = board is in sync with an external IRIG source 1 = board is in "free run"
Offset 11	
0–19	Binary microseconds
20–23	Reserved
24–27	1s position of BCD Seconds
28–31	10s position of BCD Seconds

Intermessage Gap

Control Block Offset 12. The intermessage gap is a 16-bit quantity that contains the gap from the last word on the bus of the preceding message to the beginning of the first word in this message (generally the first command word).

New Database

Control Block Offset 13.

- ***Bits 0–19: New Database Value.*** Bits 0–19 of this word provide a synchronous means of changing the contents of the control register DATABASE. Upon completion of BC Control Block processing, the ABI-V5 firmware examines the value of these bits. If the value is nonzero and the "Enable Database Swap" bit in the buffer Pointer/Offset word is "1", the firmware copies the value of bits 0–19 to control register DATABASE. This permits the host application to completely change the location of the data buffers utilized by the RT and BC buffers for each channel. This option should only be used in Bus Controller processing if using the BC database offset for RT and BC buffers.
- ***Bits 20–30: Reserved.*** These bits are reserved for future use.
- ***Bit 31: Swap Database.*** Set this bit to "1" to swap the contents of control register DATABASE with the value contained in bits 0–19 of the New Database word. Set this bit to "0" to simply load DATABASE with the value contained in bits 0–19 of the New Database word. The state of this bit is

maintained in the event of a swap.

Two conditions must be satisfied in order for either a database swap or load to occur: (1) bits 0–19 must contain a nonzero value, and (2) bit 29 of offset 19 (Current Data Buffer Pointer/Control) must be set to "1".

BC Control Word

Control Block Offset 14. The BC Control word contains control information for the current BC message. The BC Control word specifies triggering, minor frame designation, whether retries will occur if there is an error, what interrupts to generate, and the bus on which to transmit the message. Figure 6.3.5 contains the BC Control word layout. If retries are attempted by the BC on the current BC message, this word will contain retry status information as well.

If an error occurs in a BC message, the message will be copied to the "The Last Error Message" buffer located at offset 01DCh from the Control Base Address of a channel. This buffer always contains the last message which contains an error. The updating of this buffer is not affected by the state of either the On/Off bit or Monitor/Simulating bit of an RT. This buffer will never contain spurious data. Spurious data can only appear in the Sequential Monitor buffers.

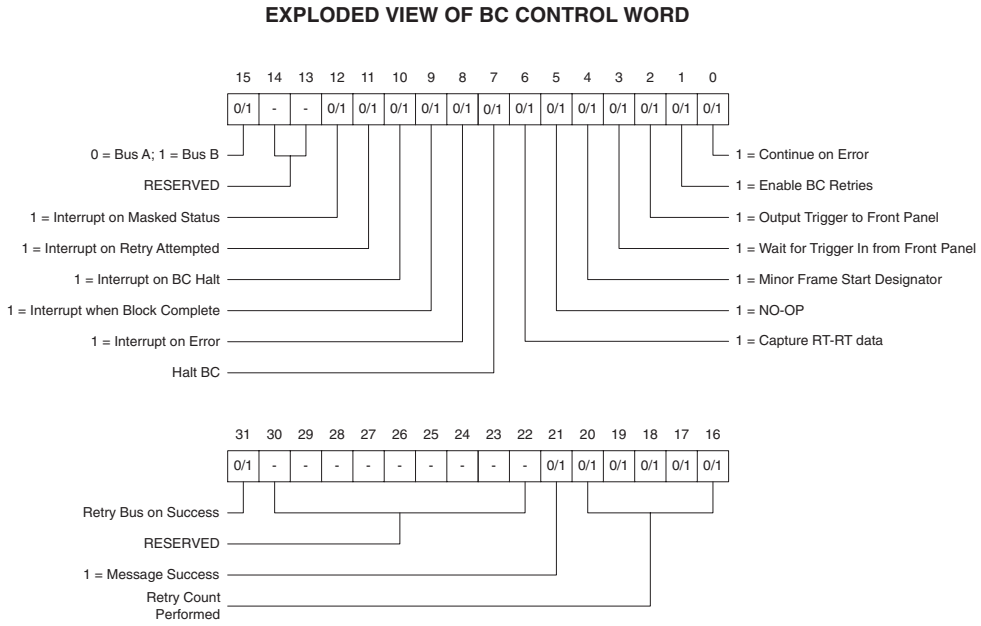


Figure 6.3.5: BC Control Word

- ***Bit 0: Continue on Error.*** This bit determines whether the firmware ceases execution of the chain program if an error occurs. Set it to "1" to cause the firmware to continue executing the chain program if a protocol error occurs. Set it to "0" to cause the firmware to terminate the chain program if an error occurs.
- ***Bit 1: Enable BC Retries.*** This bit enables BC retries for the current BC Control block.

➢ ***Bit 2: Output Trigger to Front Panel.*** Set this bit to "1" to cause the firmware to generate an external trigger to the front panel upon execution of the current BC Control Block. The external data line on which the trigger is generated is selected using the BC External Trigger Out Select Control Register (BCXTSEL) at offset 1Ah. The generated trigger has the polarity programmed in the External Data Polarity Control Register (XDATPOL) at offset 19h.



Cross Reference: Refer to *Chapter 2: ABI-V5 Hardware Installation* for trigger connections and to *Chapter 5: Host Commands/Control Registers* for details on BCXTSEL and XDATPOL.

➢ ***Bit 3: Wait for Trigger from Front Panel.*** Set this bit to "1" to cause the firmware to wait for an external trigger from the front panel before executing this BC Control block. The trigger is TTL and normally held high by the trigger source. The channel will trigger on a negative edge. The trigger must be held low for 300 ns. Upon receipt of the trigger from the front panel, the firmware begins transmission of the current BC Control Block.



Cross Reference: Refer to *Chapter 2: ABI-V5 Hardware Installation* for trigger connections.

➢ ***Bit 4: Minor Frame Start Designator.*** This bit marks the BC Control block as the first block in a minor frame. During BC execution the firmware will examine the state of this bit in each BC Control block it processes. If the firmware processes a BC Control block with this bit set to "1", it waits for the time programmed in the Frame Timer Control Register (FRMTMR) to expire before transmitting the current BC Control Block. If the time has already expired, the block is transmitted immediately. This is known as a "late start" BC block.



Note: A BC chain running under a minor frame may be started via an external trigger. Once a BC chain is running and the minor frame timer is in use, the "Wait for Trigger from Front Panel" bit will be ignored.

➢ ***Bit 5: NO-OP.*** Set this bit to "1" to "no-op" the BC Block, causing the firmware to ignore all actions for the block. When the firmware encounters a BC Block with the NO-OP bit set to "1", it immediately attempts to process the next block, pointed to by NEXT Pointer. If the next block also has the NO-OP bit set, the firmware proceeds to the following block until it reaches a BC Control Block which does not have the NO-OP bit set.



Warning: A looping BC chain consisting only of NO-OP blocks is an illegal condition. Do not program a BC list with this condition. If this condition is encountered during execution, the firmware will enter an invalid state.

- ***Bit 6: Capture RT-RT Data.*** Set this bit to "1" to cause data from an RT-RT transfer to be placed in the BC data buffer. This bit is only valid if the transfer type is RT-RT or RT-RT Broadcast.
- ***Bit 7: Reserved.*** This bit is reserved for future use.
- ***Bit 8: Interrupt on Error.*** Set this bit to "1" to generate an interrupt for any bus error that occurs during execution of this BC Control block. If an error occurs in the BC message, the resulting message packet will be copied to the "Last Error Message" buffer of the channel.
- ***Bit 9: Interrupt when Block Complete.*** Set this bit to "1" to generate an interrupt when execution of this BC Control block has completed.
- ***Bit 10: Interrupt on BC Halt.*** Set this bit to "1" to generate an interrupt when BC chain execution halts. Halting of a BC chain most commonly occurs due to a NULL value in the NEXT pointer of a BC Control block. BC chain execution may also halt if the firmware reads a NULL pointer value in the TRX Buffer Pointer of a BC Control block or if an error occurs in the message and the Continue on Error bit is not set.
- ***Bit 11: Interrupt on Retry Attempted.*** Set this bit to "1" to generate an interrupt if retries were attempted for this BC Control block. The firmware will attempt retries if bit 1 is "1" and an error occurs during processing of the current BC Control Block. This interrupt only indicates that retries were attempted. It does not indicate the success or failure of the retries.
- ***Bit 12: Interrupt on Masked Status.*** Set this bit to "1" to generate an interrupt on masked status. When this bit is "1", the status response word(s) from offset 8 in the BC control block are logically ANDed with the contents of the BCSMSK control register (offset 1Dh from the control base address of the channel). If the resulting value is nonzero, an interrupt is generated.
- ***Bits 13–14: Reserved.*** These bits are reserved for future use.
- ***Bit 15: Bus Select.*** This bit tells the firmware to send the commands of the current BC message on bus A or B. Set this bit to "0" (the default setting) to send commands on bus A. Set it to "1" to use bus B.
- ***Bits 16–20:Retry Count Performed.*** These bits contain a count of how many retry attempts were made for the BC message. The value in this field never exceeds the retry count programmed in the BCRETRY control register of the pointer table. If this field is nonzero, retries have been attempted. (Examine bit 21 to determine if they were successful and bit 31 to determine the bus on which they were successful). *Clear this bit after examining the BC Control block.*
- ***Bit 21: Message Success.*** The firmware sets this bit to "1" if the BC Control

Block was able to complete without error. If retries are enabled, examine the Retry Count Performed field; if it contains a nonzero value, retries were attempted. If retries were attempted and this bit is set, bit 31 will indicate the bus on which they were successful. *Clear this bit after examining the BC Control block.*

- ***Bits 22–30: Reserved.*** These bits are reserved for future use.
- ***Bit 31: Retry Bus on Success.*** This bit identifies the bus on which a retry was successful. This bit is valid only if bits 16–21 contain a nonzero value and bit 21 is set to "1". If bit 31 is "0", the retry was successful on the primary bus. If bit 31 is "1", the retry was successful on the secondary bus. For retries, the primary bus is the bus on which the message was originally transmitted (Bit 15 of the BC Control Word). *Clear this bit after examining the BC Control block.*

Reserved Control Block Offsets 15 and 16. These two words are reserved for future implementation of conditional branching.

Transfer Type/ Error Injection Control Control Block Offset 17.

- ***Bits 0–2: Transfer Type.*** The transfer type designates what type of 1553 transfer is being initiated by the BC. Transfer types are listed in Figure 6.3.1 on page 6.
- ***Bits 3–14: Reserved.*** These bits are reserved for future use.
- ***Bit 15: Schedule BC Block.*** Set this bit to "1" to schedule the BC transfer as described in the Schedule Time discussion.
- ***Bits 16–31: Error Injection Control.*** This half-word (shown in Figure 6.3.6) provides control information for injecting an error in the BC message.

  The Error Word field indicates the word on which the firmware injects an error. A value of "0" indicates that the firmware will not inject a word level error. A value of "1" indicates the error will be injected on the first transmitted word, "2" indicates it will be injected on the second, and so on. Note that the Error Word field includes command words in the word count. Therefore, a value of "1" causes the error to be injected on Command Word 1. A value of "2" results in the error being injected on Command Word 2 if the transfer type is RT-RT or RT-RT Broadcast. Otherwise, a value of "2" or more causes the error to be injected on a data word. It is your responsibility to program the correct value in this field.

  Long Message and Short Message inject an error at a message level rather than at a word level. These bits cause the BC to transmit one too many or one too few data words, respectively. The Insert Gap bit inserts a small gap in the data stream, prior to the word indicated in the Error Word field. The remaining bits indicate what type of error to inject on the word specified in the Error Word field. These bits apply only to BC-RT or Receive Mode

Codes with Associated Data Words. These bits, including the Insert Gap bit, have no meaning if the Error Word field is zero.

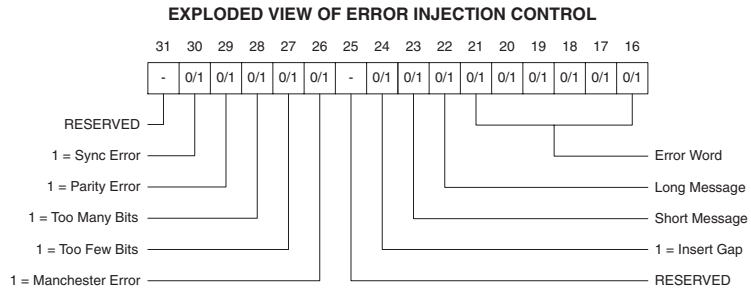


Figure 6.3.6: Error Injection Control Word

Command 1/ Command 2

Control Block Offset 18. Command 1 contains the receive command of an RT-RT or the command word of any other type transfer. Command 2 contains the transmit command of an RT-RT or RT-RT Broadcast. For all other transfer types, Command 2 has no meaning.

Current Data Buffer Pointer/ Control

Control Block Offset 19. The Current Data Buffer Pointer/Control contains control bits for the BC Control Block and a pointer/offset to a list of data buffers. The first word in each data buffer contains the NEXT Data Buffer Pointer/Control word which has the same format as Current Data Buffer Pointer/Control. This format is shown in Figure 6.3.7.

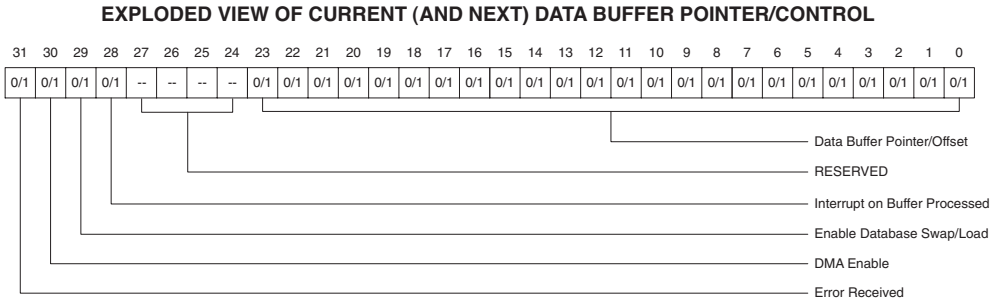


Figure 6.3.7: Current (and Next) Data Buffer Pointer/Control Word

Bits 0-23 Data Buffer Pointer/ Offset

The Data Buffer Pointer portion of this word specifies a pointer to a user allocated data buffer. This data buffer is 18 words long and is allocated from user space in ABI-V5 memory beginning at 1800h. The buffer is used by the firmware to hold data received by the BC or data to be transmitted by the BC.

The firmware adds the contents of the DATABASE control register to the contents of Data Buffer Pointer/Offset to obtain the actual data buffer address. Therefore, the interpretation of this pointer/offset is dependent on the contents of DATABASE. If DATABASE contains a zero, this location provides an actual physical memory pointer to the beginning of the data buffer. If DATABASE contains a nonzero value, this location serves as an offset from DATABASE to the

beginning of the data buffer. If the firmware encounters a null value in this field, BC execution will halt regardless of the state of the Continue on Error bit. If this occurs, the firmware will generate error code 108h. The error table entry will contain a pointer to the BC block in which a null data buffer pointer was encountered.

Data Buffer Description

If the transfer type is BC-RT, BC-RT Broadcast, or a Receive Mode Code with Associated Data Word, the firmware reads the buffer to retrieve the data which the BC transmits. Data words 1 through *n* contain 16-bit data words for transmission.

When the BC is receiving data (i.e., RT-BC, or transmit mode code with associated data word), the firmware copies the data to this buffer if the message is received without error. The data words written by the firmware to these locations are packed.



Note: If the word count is odd, bits 16–31 of the last data word in the packet will have no meaning.

Allocation of a data buffer is not required for RT-RT, RT-RT broadcast, or mode codes without associated data words.

The last word in the data buffer (offset 16) is provided for word count errors in the event that the BC receives more than 32 words. Any data received which exceeds 33 words is placed in a spurious data packet and placed in the sequential monitor.

If the transfer completes error free, the contents of the Next Data Buffer Pointer/Control are written to the Current Data Buffer Pointer/Control location. Figure 6.3.8 illustrates one possible BC configuration.

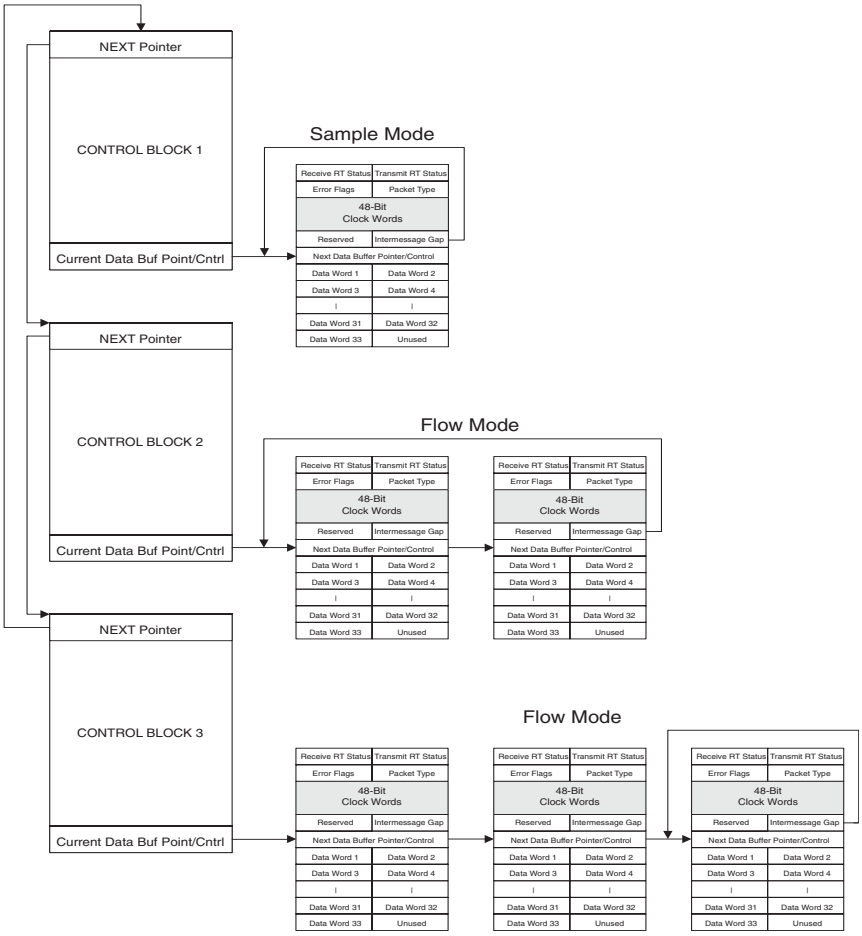


Figure 6.3.8: Sample BC Configuration

- ***Bits 24–27: Reserved.*** These bits are reserved for future use.
- ***Bit 28: Interrupt on Buffer Processed.*** Set this bit to "1" to generate an interrupt (code 00001000h) when the current data buffer has been processed and the message has completed error free.
- ***Bit 29: Enable Database Swap/Load.*** Set this bit to "1" to enable a Database swap or load. If the contents of New Database, control block offset 13 are nonzero and this bit is set to "1", the firmware swaps or loads the Database value as described under the subsection *New Database*. This swap or load occurs only if the message completes error free or if the Continue on Error bit is set to "1".
- ***Bit 30: DMA Enable.*** Set this bit to "1" to enable a DMA transfer upon message completion. If this bit is set to "1", the DMA Start bit of the DMA Control Word (BC Control Block offset 4) is set to "1", and the message completes error free, the DMA transfer specified by offsets 4–7 of the BC Control Block will occur. If the message does not complete error free but the Continue on Error bit is set to "1", the DMA transfer will still occur.

➢ ***Bit 31: Error Received.*** This bit is set by the firmware to indicate that an error occurred in the message and the data copied to the BC receive buffer may not be complete. This bit is set only for RT-BC or transmit mode codes with an associated data word where the BC is receiving data.

7: RT Simulation/RT Monitoring

The ABI-V5 can simultaneously simulate a BC and 31 RTs while performing real-time mapped monitoring and sequential monitoring on one or two dual redundant 1553 channels. This chapter reviews processes and data structures required for Remote Terminal simulation and real-time mapped monitoring. The options for linked RT transmit buffers and mode code configurations provide the ideal memory model for simulated RT responses. Similarly, linked buffers may be defined at the RT address or subaddress level for real-time mapped monitoring. All buffers provide for VMEbus master operations, and monitor buffers are time tagged with a 48-bit time value.



Note: Before you read this chapter, be familiar with the information discussed in *Chapter 3: 1553 Operations—Module Overview*, *Chapter 4: Module Startup/Test*, and *Chapter 5: Host Commands/Control Registers*.



Note: The terms offset, pointer, and address are used interchangeably when referring to SBS device memory. The device memory is accessed by an offset from the device's base memory address.



Cross Reference: For Sequential Monitor functions, refer to the paragraphs concerning Sequential Monitor Filter words discussed in the overview of the RT Definition Table on 7-9.



Cross Reference: See the *Interface Libraries* manual for C functions, Ada procedures, and device driver information necessary to program the ABI-V5 from the host.

7.1 RT Structure Overview

RT simulation and mapped monitoring is controlled by two data structures:

- RT Status Response Table
- RT Definition Table

RT Status Response Table

The RT Status Response Table defines which RTs the ABI-V5 is simulating and allows the user to program RT status response words, response times, and 1553 variant protocols (future protocol releases).

RT Definition Table

The RT Definition Table provides the user with a comprehensive table of control and pointer words for managing RT transmit data buffers, real-time monitor buffers, mode code information, command counters, and control bits for overall RT operations (including Sequential Monitor subaddress filter bits). There are 32 RT Definition Tables for each channel, one for each RT including broadcast message RT Address 31.

RT Definition Pointer Table

The offset to each of these tables is stored in the RT Definition Pointer Table, a structure containing 32 elements, or one element for each Definition Table.

This chapter reviews these data structures and the process of managing data buffers and RT responses for simulation and monitoring.



Cross Reference: The reader should be familiar with RT-related Host Commands (see the section *Host Commands* on 5-26).



Note: Addresses listed in this chapter are 32-bit longwords.

7.2 Operational Steps for RT Simulation and RT Monitoring

The typical steps for programming the ABI-V5 for RT functions are as follows:

1. Load the Instruction Code per *Chapter 4: Module Startup/Test*.
2. Define the mode of operation and the status responses for the desired RTs in the RT Status Response Table.
3. Allocate the RT definitions in the RT Definition Tables and place the pointers in the RT Definition Pointer Table.
4. Issue the "Initialize Channel" command (Host Command 1). If the ABI-V5 is also performing BC or Sequential Monitor functions, define these data structures before issuing Host Command 1.
5. Issue the "Enable Channel" command (Host Command 2).
6. To redefine the RT Status Response Table during 1553 processing, simply program the new value and then issue the "Update Single RT Status Response Table Entry" command (Host Command 7).



Note: Data structure tables apply to each channel and are offset from the base address of each channel—add 0800h to the table offset for channel 1 and 1000h to the table offset for channel 2.

7.3 RT Status Response Table

The RT Status Response Table defines which RTs the ABI-V5 is simulating and how an RT responds to commands received on the 1553 data bus. The RT Status Response Table is arranged by RT number and begins at offset 0200h from the Control Base Address of a channel. Each entry in the table consists of two 32-bit words, as shown in Figure 7.3.1.

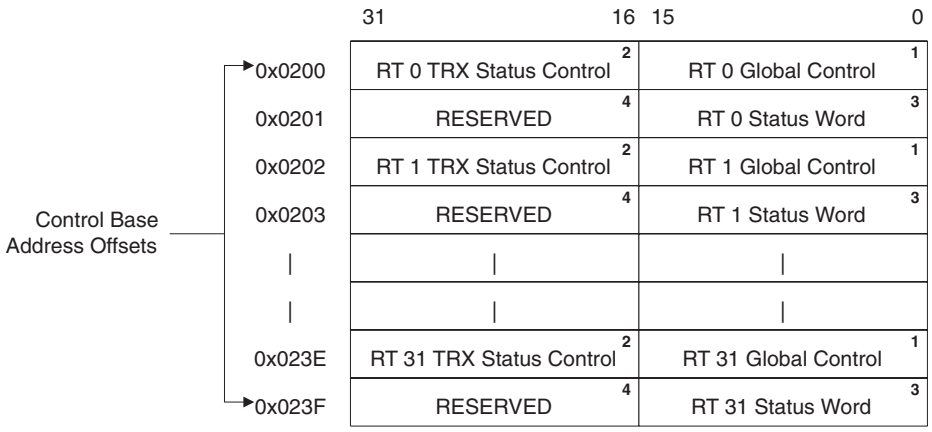


Figure 7.3.1: RT Status Response Table

Half-Word (1) The first half-word (1) of an RT Status Response Table entry contains global control information concerning the RT. This global control information includes RT on or off, RT simulation or monitor, RT protocol, Dynamic Bus Control Acceptance information, and bus fail A/B control bits.

Half-Word (2) The second half-word (2) contains the status response gap and error injection type in formation. The status response gap is the gap that precedes the status word. The status response gap time is defaulted by the firmware at startup to 6.0µsecs. The error injection type indicates the type of error to be inserted on the status response word.

Half-Word (3) The third half-word (3) contains the actual status response word for transmission on the 1553 bus. The status response word for each RT is defaulted by the firmware at startup to contain the RT number in bits 11–15.

Half-Word (4) The fourth half-word is reserved for future expansion.

An exploded view of half-words (1)–(3) appears in Figure 7.3.2 followed by a detailed discussion.



Note: Any changes made to the RT Status Response Table will not take effect until the host issues either an "Initialize Channel" or "Update Single RT Status Response Table Entry" host command.

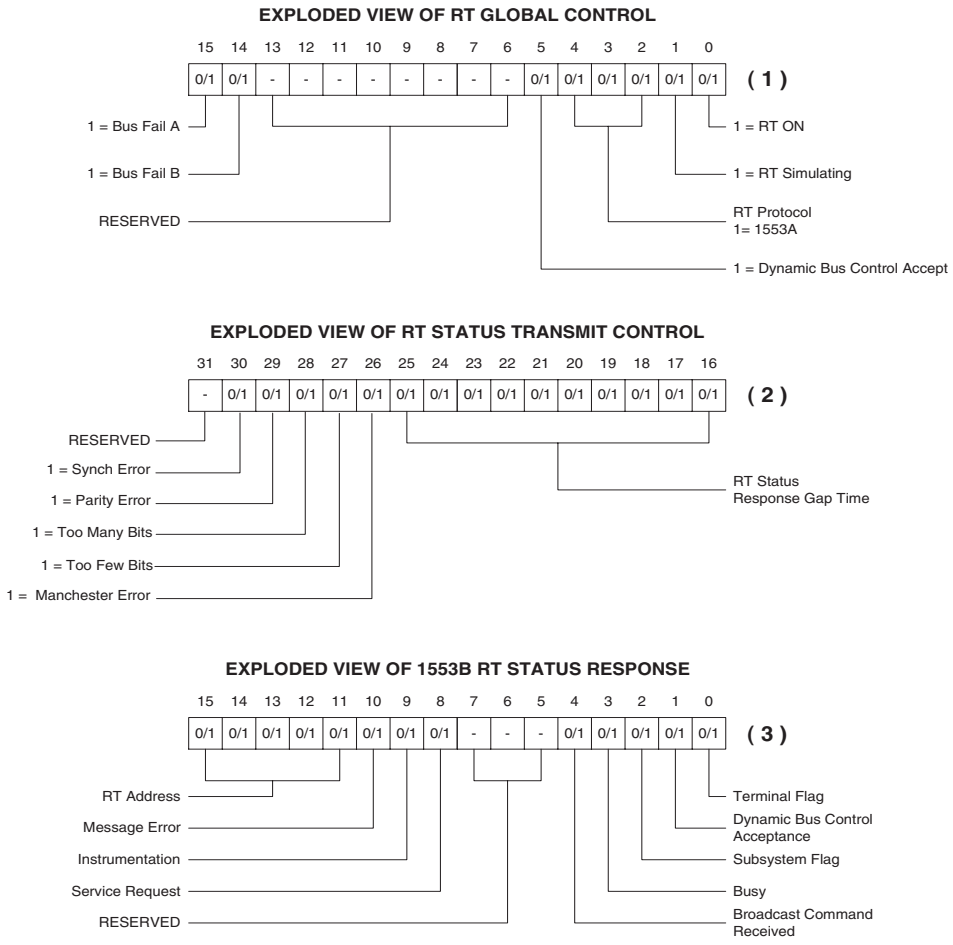


Figure 7.3.2: RT Status Response Table Half-Words

7.4 Half-Word Descriptions

(1) RT Global Control

This half-word contains global control information for the RT which determines function and protocol.

- ***Bit 0: RT On/Off.*** This bit controls whether the RT is on or off. A value of "1" indicates that the RT is on, and a value of "0" indicates that the RT is off. If the RT is on, bit 1 defines the function of the RT (Simulation or Map Monitor). When an RT is off, it ignores any messages received on the bus. Regardless of whether the RT is on or off, sequential monitoring proceeds as programmed in the RT Definition Table. If a control buffer pointer is defined in the RT definition table, then data buffers must be defined and the RT will be mapped monitored.

  This bit is useful for "freezing" an RT data buffer, i.e., following an interrupt. To freeze the buffer, turn the RT off by setting this bit to "0" and then issue an "Update Single RT Status Response Table Entry" command (Host Command 7).

- ***Bit 1: RT Simulating.*** This bit controls whether an RT responds to commands received on the bus. A value of "1" indicates that the ABI-V5 channel is simulating the RT. Simulation of specific subaddresses for the RT is controlled by programming the appropriate data buffers in the RT Definition Table. The ABI-V5 channel receives or transmits data only for RT subaddresses that contains a nonzero control buffer pointer value in the RT Definition Table.

  A value of "0" indicates that the ABI-V5 is not simulating the RT. When the RT is not simulating, it will not respond to commands received on the bus. When operating in this mode, the RT performs like a receiving RT, regardless of whether the command word is for a receive or transmit subaddress. The firmware places data in the appropriate MON RCV SA or MON TRX SA that contains pointers to buffers.

- ***Bits 2–4: RT Protocol.*** The ABI-V5 supports both 1553A and 1553B protocols, selected through the bits shown in Table 7.4.1.

Table 7.4.1: RT-1 RT Protocol Bit Definitions

Bit 4	Bit 3	Bit 2	Protocol
0	0	0	1553B
0	0	1	1553A
0	1	0	Unused
1	0	0	Unused

Note the following when selecting 1553A protocol:

- RT Status Response Time: The default status response time is set to 6.0 µs on the ABI-V5, valid for both 1553A and 1553B protocols.
- Broadcast Considerations: Broadcast commands are not supported in the 1553A specification. RT 31 is treated as a normal RT.
- Mode Codes: The only defined mode code in 1553A is mode code 00000h, designated as "dynamic bus allocation." All other mode codes are user defined. The ABI-V5 responds to all 1553A mode codes with only a status response word.
- Subaddress 31: Unlike 1553B, subaddress 31 is treated as a normal subaddress and standard message transfer types may designate subaddress 31 in the subaddress field of the command word.

- ***Bit 5: Dynamic Bus Control.*** The state of this bit determines how a simulating RT responds to a Dynamic Bus Control Mode Code. If this bit is a "1", the RT responds to a Dynamic Bus Control Mode Code with the Dynamic Bus Control Acceptance bit set in the status response word.
- ***Bits 6–13: Reserved.*** Bits 6–13 are reserved for future use.
- ***Bit 14: Bus B Operation.*** If this bit is set to "0", the simulated RT responds to messages received on Bus B. If this bit is set to "1", the RT does not respond to messages received on Bus B.
- ***Bit 15: Bus A Operation.*** If this bit is set to "0", the simulated RT responds to messages received on Bus A. If this bit is set to "1", the RT does not respond to messages received on Bus A.

(2) RT Status Transmit Control

Status Response Gap and Error Injection Type. The first ten bits of this half-word contain the status response gap time for the RT. This value specifies how long the ABI-V5 channel waits before transmitting the status response word for a simulating RT. The firmware sets a default value for the status gap of 6 µs, calculated using the formula:

```
Status Gap = ((1553B gap – 2) * 4) – 3.
```

The result of this formula yields the dead bus time for the RT Status Response gap in 250-ns tics. This default value is valid for both 1553A and 1553B protocols. It is the user's responsibility to program the status response gap time for each simulated RT. (Default status words and status response gaps are programmed by the firmware upon startup.) The next six bits specify what type of error to inject on the status response word that will be transmitted on the 1553 bus.

(3) RT Status Response Word

This half word contains the status response value that will be transmitted on the 1553 bus in response to a command word received for the corresponding RT.



Note: The following bits deal with mode commands which apply only to 1553B protocol. 1553A protocol does not support mode commands.

- ***Bit 0: Terminal Flag Bit.*** The host software sets this bit to "1" to indicate a "terminal-failed" condition. The ABI-V5 firmware also controls this bit when responding to Inhibit Terminal Flag Bit and Override Inhibit Terminal Flag Bit mode commands. The firmware responds to an Inhibit Terminal Flag Bit mode command by sending a status word containing a terminal flag bit set to "0", regardless of the actual status of this bit. The firmware responds to a subsequent Override Inhibit Terminal Flag Bit by sending a status word containing a terminal flag bit which is set to the value contained in the RT Status Response Table.
- ***Bit 1: Dynamic Bus Control Acceptance Bit.*** This bit is set by the ABI-V5 firmware in response to a Dynamic Bus Control mode command. (The state of the bit is determined by the state of bit 5 in the RT Global Control half-word for the particular RT. If the bit is set to "1" in the RT Global Control half-word, the dynamic bus control acceptance bit will be set to "1" in the RT Status Response half-word.) The host software must also respond to this mode command by enabling an interrupt so that it may initiate a bus controller program. Both of these actions must take place in order for Dynamic Bus Control to be accepted.
- ***Bit 2: Subsystem Flag Bit.*** This bit is set to "1" by the host software to indicate a potentially failed subsystem. It is never set by the ABI-V5 firmware.
- ***Bit 3: Busy.*** This bit is set to "1" by the host software to indicate a "terminal-busy" condition. It is never set by the ABI-V5 firmware. When this bit is set to "1", the RT will not transmit any data words, only status words.

  If the busy bit is set in the status word of a receiving RT involved in a BC-RT or RT-RT message, data will be stored in the RT data buffer. Upon successful completion of the BC-RT or RT-RT message, the RT will transmit a status word with the busy bit set.

  When the busy bit is set in the status word of a transmitting RT, the resulting action depends on the message type. For RT-BC and RT-RT messages, the RT will transmit its status response with the busy bit set and will suppress the transmission of data words. For RTs involved in mode codes, the transmission of an associated data word will be suppressed.
- ***Bit 4: Broadcast Command Received Bit.*** On receipt of a broadcast command, the ABI-V5 firmware sets this bit to "1" for all active RT simulations. In response to a Transmit Status Word and/or Transmit Last Command mode code immediately following a broadcast command, the firmware sets this bit to "1" in the RT's status response word. Otherwise, the firmware resets this bit to "0".

- ***Bit 8: Service Request Bit.*** This bit is set to "1" by the host software; thereafter, it appears in each status response word until it is cleared by the host software. This bit is typically used to signal the BC to issue a Transmit Vector Word mode command to the RT. After the RT has issued a status response with the Service Request Bit set, the host should clear this bit via an Update RT Status Response host command.
- ***Bit 9: Instrumentation Bit.*** This bit is used by the host software for specialized applications. It is never set by the ABI-V5 firmware.
- ***Bit 10: Message Error Bit.*** This bit indicates that the previous message received by the RT did not pass all validity checks. It remains set to "1" until the next message is received, at which time it is updated to reflect the validity of that message. It is set only by the ABI-V5 firmware.

7.5 RT Definition Table

The RT Definition Table contains control and pointer words for managing RT transmit data buffers, real-time mapped monitor buffers, mode code information, command counters, and overall RT operations. The overall structure of an RT definition is illustrated in Figure 7.5.1.

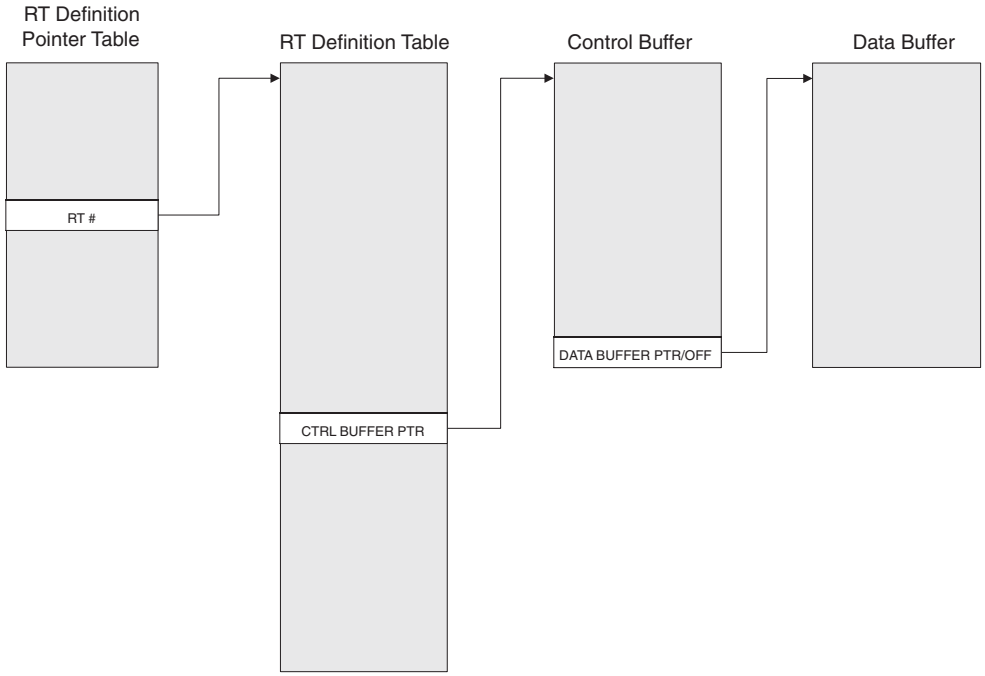


Figure 7.5.1: RT Definition Structure

The definition of an RT begins with the RT definition pointer table, located at 32-bit offset 0240h (byte offset 0900h) from the control base address of a channel. The RT definition pointer table is 32 words long and is arranged by RT number. Each location in the table may contain a pointer to an RT Definition. RT Definitions are allocated from free memory space and are 128 words long. Each definition in the table contains control words and control buffer pointers for the corresponding RT. Figure 7.5.2 details the RT Definition Table.

If an RT is ON in the RT Status Response Table an RT definition table must be allocated. If an RT is ON and an RT definition table is not allocated (i.e., the corresponding RT pointer is NULL), error code 104h is updated with the number of the RT in error.

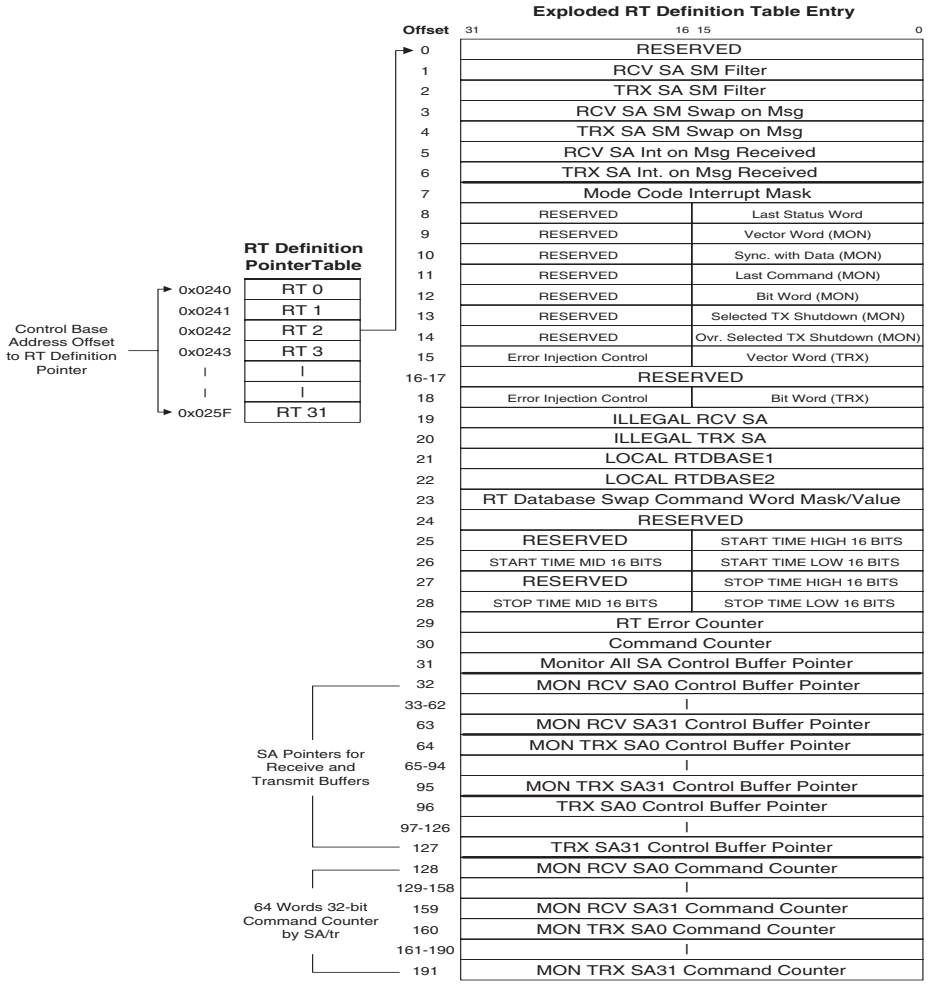


Figure 7.5.2: RT Definition Table Entry

7.5.1 Words in an RT Definition Table Entry

The following paragraphs detail each of the words in an RT definition table entry.

Reserved Offset 0. This word is reserved for future expansion.

RCV SA SM Filter, TRX SA SM Filter Offsets 1, 2. The Receive (RCV) and Transmit (TRX) Subaddress (SA) Sequential Monitor (SM) Filter words determine which subaddresses of the RT are to be sequentially monitored. Each word consists of a bit field organized by subaddress (bit 0 = SA 0, bit 1 = SA 1, etc.). The first word is for receiving subaddresses and the second for transmit. If a bit is “1”, then the corresponding subaddress will be sequentially monitored. When a bit is “1”, the ABI-V5 will place messages to or from the in the sequential monitor if the message is received without error.



Cross Reference: Refer to *Chapter 8: 1553 Bus Monitoring* for details on these messages.

RCV SA SM Swap on Msg, TRX SA SM Swap on Msg

Offsets 3, 4. The Receive (RCV) and Transmit (TRX) Subaddress (SA) Sequential Monitor (SM) Swap on Message words are used to force a sequential monitor buffer swap when the ABI-V5 receives an error-free message for the. Each word consists of a bit field organized by subaddress (bit 0 = SA 0, bit 1 = SA 1, etc.). The first word is for receiving subaddresses and the second for transmit. If a bit is "1", messages to that combination force the sequential monitor buffers to swap. If the combination is being sequential monitored, the firmware places the message in the current sequential monitor buffer and then swaps the buffers (the message is *always* the last message in a sequential monitor buffer).



Note: The following actions are dependent on whether the RT is on or off and whether the message received contains an error. If the RT is turned off in the RT Status Response Table or a received message contains an error, none of the following actions can occur, regardless of how the host has programmed the RT. In all cases, the firmware determines whether the RT is on or off. If the RT is on and the message is error free, processing will proceed as programmed by the host. If the RT is off, the firmware simply skips any of the following processing.

RCV SA Int on Msg Received, TRX SA Int on Msg Received

Offsets 5, 6. The Receive (RCV) and Transmit (TRX) Subaddress (SA) Interrupt on Message Received bit fields cause the ABI-V5 to generate an interrupt for messages received by the RT/SA combination. Each word consists of a bit field organized by subaddress (bit 0 = SA 0, bit 1 = SA 1, etc.). The first word is for receiving subaddresses and the second for transmit. If a bit is "1", messages to that combination cause the ABI-V5 to generate an interrupt. The interrupt is generated after the entire message is processed.

Mode Code Interrupt Mask

Offset 7. The Mode Codes Interrupt Mask word provides the means of generating an interrupt code 00000002 upon detection of a programmed mode command. The word consists of a bit field organized by mode codes (bit 0 = Mode Code 0, bit 1 = Mode Code 1, etc.). If a bit is "1", the ABI-V5 generates interrupt for command words detected containing that mode code. The interrupt is generated after the entire message is processed.

Last Status Word

Offset 8, Updated by Firmware. Bits 0–15 of this word contain the Last Status Word transmitted by the RT. The RT Status Response table controls error injection of status responses to both normal messages and Transmit Last Status mode commands to the RT. Bits 16–31 are reserved.

Vector Word (MON) Offset 9, Updated by Firmware. Bits 0–15 of this word contain the data word from a Transmit Vector Word mode code received by the RT. Bits 16–31 are reserved.

Synchronize with Data (MON) Offset 10, Updated by Firmware. Bits 0–15 of this word contain the data word of a Synchronize with Data Word mode code received by the RT. Bits 16–31 are reserved.

Last Command (MON) Offset 11, Updated by Firmware. Bits 0–15 of this word contain the last command word received by the RT. Bits 16–31 are reserved.

Bit Word (MON) Offset 12, Updated by Firmware. Bits 0–15 of this word contain the data word from a Transmit Bit Word mode code received by the RT. Bits 16–31 are reserved.

Selected TX Shutdown (MON) Offset 13, Updated by Firmware. Bits 0–15 of this word contain the data word from a Selected Transmitter Shutdown mode code received by the RT. Bits 16–31 are reserved.

Ovr. Selected TX Shutdown (MON) Offset 14, Updated by Firmware. Bits 0–15 of this word contain the data word from an Override Selected Transmitter Shutdown mode code received by the RT. Bits 16–31 are reserved.

Vector Word (TRX) Offset 15. Bits 0–15 of this word contain the data word the ABI-V5 transmits in response to a Transmit Vector Word mode code. The data word itself occupies bits 0–15 of the word. Bits 16–31 contain the Error Injection Control information for the vector word. Refer to Figure 7.5.2, to Figure 7.5.3, and to the accompanying discussion for bit definitions.

Reserved Offsets 16, 17. These words are reserved for future use.

Bit Word (TRX) Offset 18. Bits 0–15 of this word contain the data word the ABI-V5 transmits in response to a “Transmit Bit Word” mode code. The data word itself occupies bits 0–15 of the word. Bits 16–31 contain the Error Injection Control information for the bit word. Refer to Figure 7.5.2, to Figure 7.5.3, and to the accompanying discussion for bit definitions. Figure 7.5.3 shows the layout of the Error Injection Control word.

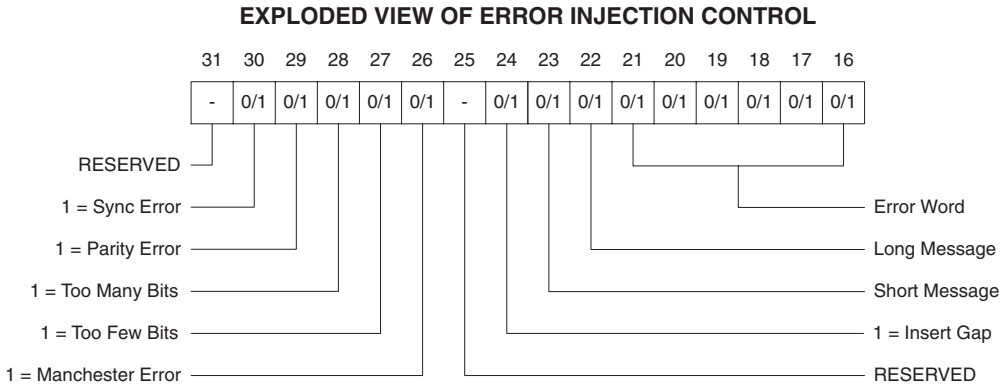


Figure 7.5.3: Error Injection Control

Error Injection Control provides control information for injecting an error in the data word of a mode code with Associated Data Word. The Error Word field indicates whether to inject an error at the word level. A value of "0" in this field indicates no word error. A value of "1" indicates the first data word transmitted. Because there is only one data word in a transmit mode code with Associated Data Word, a value of "1' is the only valid value. Values of "2" or more have no effect for Transmit Mode Codes with an associated data word. It is the user's responsibility to program the correct value in this field.

Long Message and Short Message inject an error at a message level rather than a word level. These bits cause the ABI-V5 to transmit one too many or one too few data words, respectively. If the Long Message bit is set, the ABI-V5 transmits the corresponding mode code data word twice. If the Short Message bit is set, the ABI-V5 suppresses transmission of the corresponding mode code data word. The Insert Gap bit inserts a short gap in the data stream. This gap is inserted prior to the word specified in the Error Word field. The remaining bits indicate what type of error to inject on the word specified in the Error Word field. For RT Simulation, these bits apply only to RT-BC or Transmit Mode Codes with Associated Data Words. These bits, including the Insert Gap bit, have no meaning if the error word field is zero.



Note: Errors cannot be inserted on the data word of Transmit Last Command.

Reserved Offsets 19–29. These bits are reserved for future use.

Command Counter Offset 30, Updated by Firmware. This word contains a count of the number of command words an RT receives. It is updated upon the completion of each BC message from the 1553 data bus and before generation of any interrupts. This word is *not* incremented if an error occurs in the message.

7.5.2 RT Simulate/RT Monitor Summary

An RT subaddress may be simulated only, monitored only, or both simulated and monitored. The following paragraphs discuss how these functions are achieved using the ABI-V5.

Simulate Only

A simulated subaddress responds to bus commands; i.e., in response to a command, a receive subaddress sends a status response word and a transmit subaddress sends status and data, if required. To simulate a subaddress, set bit 1 in the RT Global Control half-word to "1". For a transmit subaddress, define a buffer in the appropriate TRX SA Pointer/Offset (RT Definition Table offsets 96–127). (If the simulated transmit subaddress is processing mode codes, it does not require a buffer.) Since it will only respond with a status word, a simulated receive subaddress does not require the definition of a control buffer pointer/offset.

Monitor Only

A subaddress may be monitored regardless of the state of bit 1 in the RT Global Control half-word. To monitor a subaddress, define a monitor buffer and place its value in the appropriate control buffer pointer (or define offset 31 to monitor all subaddresses). Receive subaddresses are monitored using RT Definition Table offsets 32–63. Transmit subaddresses are monitored using RT Definition Table offsets 64–95. If offset 31 is used to monitor all subaddresses, it supersedes offsets 32–95.

Simulate and Monitor

The ABI-V5 has the capability both to simulate and to monitor data for the same subaddress. This is possible because the RT Definition Table contains separate control buffer pointer/offsets for monitoring and transmitting. To simulate and monitor a subaddress, set bit 1 in the RT Global Control half-word to "1" and then define a monitor buffer, placing its value in the appropriate control buffer pointer (or define offset 31 to monitor all subaddresses). In addition, for a transmit subaddress, define a value in the appropriate TRX SA Pointer (RT Definition Table offsets 96–127).

7.5.3 Illegal Subaddress

Offsets 19 and 20 in an RT Definition Table entry are used to specify the Illegal Receive and Illegal Transmit Subaddresses, respectively. If the firmware running on the V5 does not contain the Illegal Command option, these words remain reserved. Figure 7.5.4 illustrates these offsets.

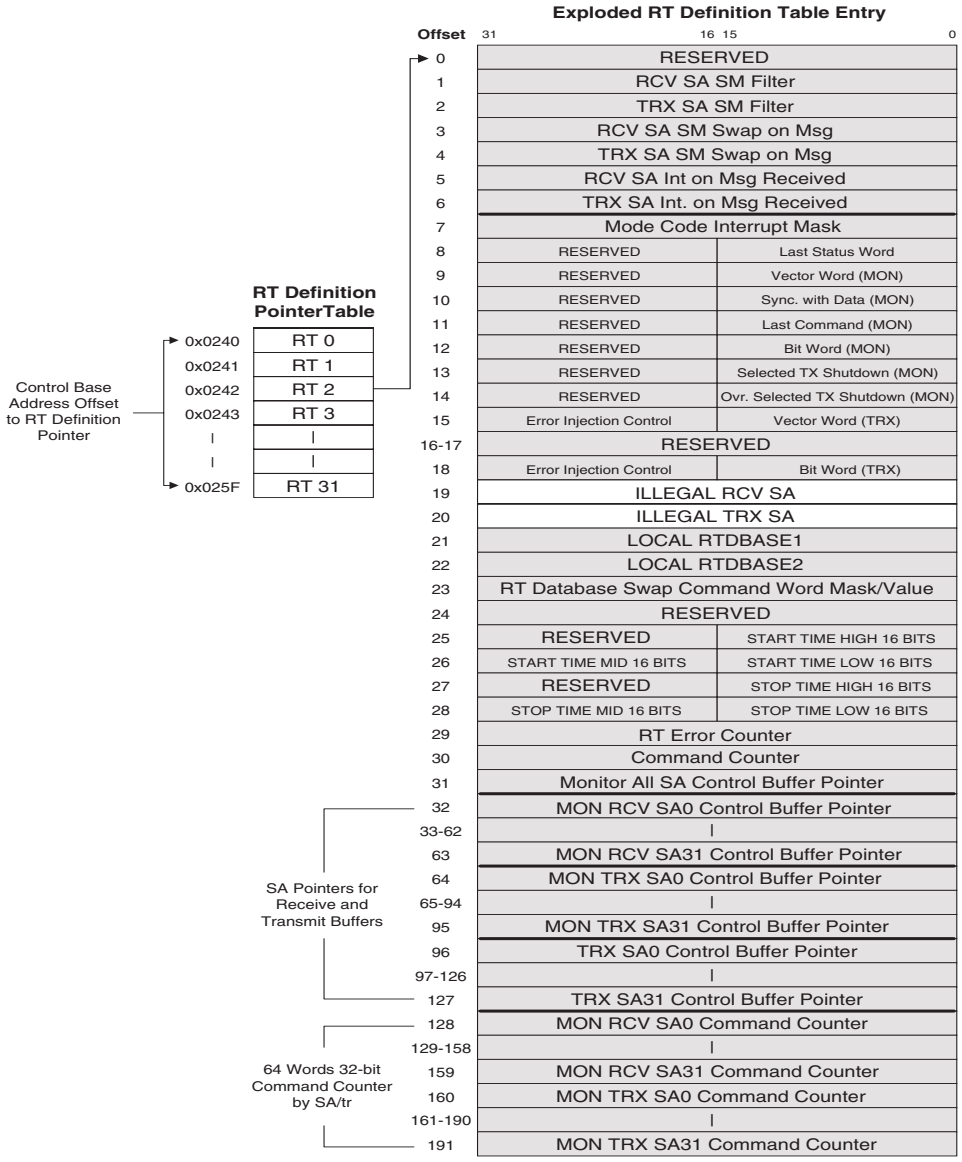


Figure 7.5.4: RT Definition Table Entry for Illegal Command Option

Each illegal subaddress word consists of a bit field organized by subaddress (bit 0 = SA0, bit 1 = SA1, etc.) If a bit is “1”, the corresponding subaddress is illegal. When a subaddress is illegal, the RT responds by setting the Message Error (ME) bit to “1” in the status response word. If the command received is a transmit command, the RT suppresses transmission of data.

Update RT Illegal Subaddress Host Command

HOSTCMD1 = 0000000Dh. This host command is available only with ABI-V5 firmware that includes the Illegal Command option. This host command must be issued in order for any illegal subaddresses of an RT to take effect.

1. Make any change necessary to a single RT's Illegal Subaddress mask entries (offsets 19 and 20 in the RT Definition Table entry).

2. Upon completion of the changes, write the RT number to HOSTCMD2, followed by command word 0000000Dh to HOSTCMD1.

   The specified RT's illegal subaddress masks will then become active.

7.6 Control Buffer Pointers/Offsets

Control buffer pointers occupy offsets 31–127 of an RT Definition Table entry. These pointers fall into three categories:

- Monitor all subaddresses
- Monitor subaddress
- Transmit subaddress

These pointer locations contain the starting address or current address in a chain of user-allocated control buffers, each of which contains a pointer to a data buffer. (Control buffers are detailed later in this chapter on 7-19.) During processing, the firmware uses the address for a particular RT/SA, as found in the RT definition table, to look up a control buffer and data buffer. If the firmware encounters a NULL pointer value in the RT definition table, the simulation/monitoring for that RT will cease.

Monitor All BPT/ Subaddresses Pointer/Offset

Offset 31. This location contains the offset of a user-defined linked list of RT control buffers. A nonzero value in offset 31 supersedes pointers in the Monitor RCV SA and Monitor TRX SA pointer locations. This provides the user the capability of monitoring messages at an RT level rather than an RT/SA level. If the content of offset 31 is zero, monitoring of messages proceeds according to the Monitor RCV SA and Monitor TRX SA pointer locations.

Upon receipt of an error-free "Complete 1553 Message" packet, message placement in the buffers of the RT proceeds as follows: The ABI-V5 firmware examines the contents of offset 31. If the contents are nonzero, the firmware uses the value as a pointer to a control buffer. If bit 1 of the DBCTRL control register is set to "1", the firmware adds the contents of the DATABASE control register to the contents of offset 31 to obtain the actual control buffer address. (DBCTRL is located at offset 1Fh from the control base address of a channel.) The firmware uses the "Monitor Data Buffer Pointer/Offset" from the control buffer as an address or offset from the DATABASE value and copies the current error-free "Complete 1553 Message" packet to that buffer. If "Monitor Data Buffer Pointer/Offset" contains a null value, error table offset 105h is updated with the command word, and bit 16 is set. In addition, the control buffer pointers are not swapped, and DMA transfers do not occur. Otherwise, the firmware copies the NEXT pointer of the control buffer to offset 31. The firmware generates any interrupts after completely copying the message to the buffer.

Monitor RCV SA Pointers/ Offsets, Monitor TRX SA Pointers/ Offsets

Offsets 32–63, Offsets 64–95. Each of these locations contains a pointer to a user-defined linked list of RT control buffers. Offsets 32–63 are used for storage of messages sent to both simulated and monitored receive subaddresses. Offsets 64–95 are used for storage of transmit subaddress messages which are being monitored. (Offsets 96–127 are used for transmit buffers for simulated transmit subaddresses.)

Upon receipt of an error-free "Complete 1553 Message" packet, message placement in the buffers of the RT proceeds as follows: The ABI-V5 firmware examines the contents of offset 31. If the contents are zero, the firmware searches for the appropriate control buffer pointer according to the SA of the command word. The firmware uses the "Monitor Data Buffer Pointer/Offset" from the control buffer as an address or offset from the DATABASE value and copies the current error-free "Complete 1553 Message" packet to that buffer. If "Monitor Data Buffer Pointer/Offset" contains a null value, error table offset 105h is updated with the command word, and either bit 17 (RCV) or bit 18 (TRX) is set. In addition, the control buffer pointers are not swapped, and DMA transfers do not occur. The firmware copies the NEXT pointer of the control buffer to the RT definition table at the location from which it was originally read. The firmware generates any interrupts after completely copying the message to the buffer.

TRX SA Pointers/ Offsets

Offsets 96–127. Each of these locations contains a pointer to a user-defined linked list of RT control buffers. Upon receipt of a transmit command, the ABI-V5 firmware determines if the RT of the command word is on or off. If the RT is on and the command is not a mode code, the firmware searches for the subaddress control buffer pointer for the transmit command. A nonzero value in the SA pointer location indicates that a buffer is available for transmission. The firmware uses the WC field of the command word to determine how many words to transmit from the buffer pointed to by the "Transmit Data Buffer Pointer/Offset" in the control buffer. If "Transmit Data Buffer Pointer/Offset" contains a null value, error table offset 105h is updated with the command word, and bit 19 is set. The firmware copies the NEXT pointer of the buffer to the RT definition table at the location from which it was originally read.



Note: When an RT is running under 1553B protocol, the transmit buffer pointers for SA0 and SA31 are never accessed by the firmware. SA0 and SA31 are reserved for mode codes, whose transmit data is retrieved either from the transmit mode code data locations in an RT definition or from memory locations maintained on the daughter board. When the RT is running under 1553A protocol, transmit SA31 becomes a valid SA from which data may be transmitted. SA0's transmit buffer pointer still remains unused.

7.7 Control Buffers

RT data buffer allocation consists of a linked list of control buffers, each of which contains a pointer to a data buffer. Each RT Control Buffer is 11 words long. The host must allocate these buffers from the free memory region beginning at 1800h.

NEXT Pointer, PREV Pointer

Offsets 0, 1. As shown in Figure 7.7.1, each control buffer, whether receive (monitor) or transmit, contains a next (NEXT) pointer and an optional previous (PREV) pointer. These locations contain the offset of the next and previous control buffers in the chain.

RT Monitor Control Buffer Format

Offset	31 — 16	15 — 0
0	NEXT Pointer	
1	PREV Pointer	
2	Buffer Control	
3	Reserved	DMA CNTL
4	ABI ADDRESS/OFFSET	
5	VME ADDRESS	
6	Reserved / DMA Vector	DMA Word Count
7	Reserved	
8	Monitor Data Buffer Offset	
9	Reserved	Reserved
10	Monitor Data Buffer Pointer/Offset	

RT Transmit Control Buffer Format (Standard Clock)

Offset	31 — 16	15 — 0
0	NEXT Pointer	
1	PREV Pointer	
2	Buffer Control	
3	Reserved	DMA CNTL
4	ABI ADDRESS/OFFSET	
5	VME ADDRESS	
6	Reserved / DMA Vector	DMA Word Count
7	Unused	TimeH
8	TimeM	TimeL
9	Reserved	Error Injection Control
10	Transmit Data Buffer Pointer/Offset	

RT Transmit Control Buffer Format (IRIG Time)

Offset	31 — 16	15 — 0
0	NEXT Pointer	
1	PREV Pointer	
2	Buffer Control	
3	Reserved	DMA CNTL
4	ABI ADDRESS/OFFSET	
5	VME ADDRESS	
6	Reserved / DMA Vector	DMA Word Count
7	free run bit/BCD Day	BCD Hour / BCD Min
8	BCD Sec / Rsrv'd	Binary µsec (bits 0-19)
9	Status Word	Error Injection Control
10	Transmit Data Buffer Pointer/Offset	

Figure 7.7.1: RT Data Control Buffer Formats

The programmer may use the PREV pointer to construct a doubly-linked list of control buffers. Subaddress control buffer pointers *always* point to the next control buffer the firmware will use. When the ABI-V5 receives or transmits data,

the firmware performs a search of the corresponding SA control buffer pointer in the RT's Definition Table. If the pointer location is nonzero, the ABI-V5 channel either uses the "Monitor Data Buffer" or "Transmit Data Buffer" pointer/offset from the control buffer for placement or retrieval of data. The firmware then copies the address contained in the NEXT Pointer of the current control buffer to the RT Definition Table. Therefore, the control buffer pointers always point to the next buffer available for data monitoring or transmission. If used, the PREV Pointer points to the previous control buffer in the chain, providing access to the last receive or transmit data buffer used by an RT/SA. The firmware never accesses the PREV Pointer. It is provided solely for the programmer's use to facilitate quick access to the last data buffer used by an RT/SA.



Note: It is the programmer's responsibility to ensure that the NEXT Pointer register contains a valid pointer. If the NEXT Pointer of a control buffer in use contains a null value, the firmware copies that value to the RT definition table. This may cause an RT/SA to stop transmitting data or monitoring to cease for a given RT/SA combination.

Buffer Control

Offset 2. Each control buffer contains the Buffer Control word at offset 2. This word allows the ABI-V5 to interrupt the host after completely processing a specific buffer in a chain of RT/SA buffers.

- ***Bit 0: Interrupt on Buffer Processed.*** If bit 0 of this word is set to "1", an RT Buffer Processed interrupt (interrupt 00000010h) will be generated when the message to the RT/SA buffer is complete. For monitor buffers, the interrupt will be generated after the error-free message is written to the buffer. For transmit buffers, the interrupt will be generated after the buffer has been time stamped.

DMA CNTL, ABI ADDR, VME ADDR, WORD CNT/ VECTOR

Offsets 3, 4, 5, 6. These four words control DMA operations for writing/reading ABI-V5 RT data buffers to/from VMEbus memory.



Cross Reference: These words are defined in *Chapter 5: Host Commands/Control Registers*.

Standard V5 TimeH, TimeM, Time L

Offsets 7, 8 (Transmit Control Buffers Only). For the Standard V5, these three half-words are the 48-bit time stamp of the message and indicate the ABI-V5 time at which the data is retrieved for transmission from the buffer.

V5 IRIG TimeH Time L

Transmit Control Buffers having the IRIG time format are structured as shown in Figure 7.7.1.

Offsets 7, 8 (RT Transmit Control Buffers). For the V5 IRIG, RT Transmit control buffer offsets 7 and 8 contain the time stamp of the message and indicate the

IRIG time at which the data is retrieved for transmission from the buffer. The format for these words is identical to that of BC Control Block offsets 10 and 11 (see Table 6.3.1).

Figure 7.7.2: Transmit Data Control Buffer with Standard IRIG Time Stamp

Error Injection Control

Offset 9. The Error Injection Control half-word provides control information used to inject an error in the RT transmit data, excluding the vector, bit, and status words (vector, bit, and status words have their own Error Injection Control words). The Error Word field (bits 0–5) specifies the word on which to inject the error. A value of "0" in this field indicates no word error. A value of "1" indicates the first data word transmitted, "2" the second, and so on. It is the user's responsibility to program the correct value in this field. Long Message (bit 6) and Short Message (bit 7) inject an error at a message level rather than a word level. These bits cause the ABI-V5 to transmit one too many or one too few data words, respectively, relative to the word count of the Transmit Command. Insert Gap (bit 8) inserts a short gap in the data stream. This gap is inserted prior to the word specified in the Error Word field. The remaining bits indicate what type of error to inject on the word specified in the Error Word field. These bits, including the Insert Gap bit, have no meaning if the error word field is zero.

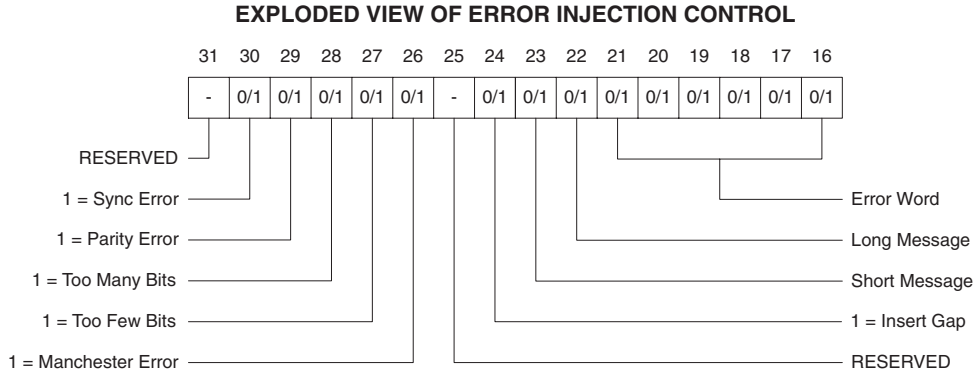


Figure 7.7.3: Error Injection Control

Data Buffer Pointer

Offset 10. The last location (offset 10) of an RT control buffer contains the data buffer pointer. For monitor buffers, this location holds the pointer to a buffer where the firmware writes a complete 1553 message packet. For transmit buffers, this location holds the pointer to a buffer where the firmware retrieves data for transmission by the RT.



Cross Reference: For monitor buffers, refer to the subsection *Receive Data Buffers* on 7-24 for details. For transmit buffers, refer to the subsection *Transmit Data Buffers* on 7-29 for details.

The firmware adds the contents of the DATABASE control register to the contents of Data Buffer Pointer/Offset to obtain the actual data buffer address. Therefore, the interpretation of this pointer/offset is dependent on the contents of DATABASE. If DATABASE contains a zero, this location provides an actual physical memory pointer to the beginning of the data buffer. If DATABASE contains a nonzero value, this location serves as an offset from DATABASE to the beginning of the data buffer.

If the firmware encounters a null value for the data buffer pointer location, error code 105h is generated. The Error Table entry will contain the command word and an ID bit identifying which buffer type is in error.

7.7.1 Data Buffer Options

Figure 7.7.1 outlines three allocation schemes for the data buffers. Option #1 illustrates a linked list consisting of only one buffer. Option #2 illustrates a linked list of multiple buffers. Both of these options permit the firmware to monitor or transmit data continuously. In order for the host to identify the last buffer used with option 2, the application software must traverse the list of buffers searching for the buffer whose next pointer contains the same value as that in the RT Definition Table pointer entry. Option #3 consists of a circular doubly-linked list which provides the most complete linking mechanism for data buffers. To find the previous buffer used, and thus the buffer with the newest data, the host software can simply read the PREV Pointer of the current buffer. This option also provides for continuous monitoring and/or transmission of data.



Note: The PREV Pointer is provided solely for the use of the programmer; the firmware does not use this register.

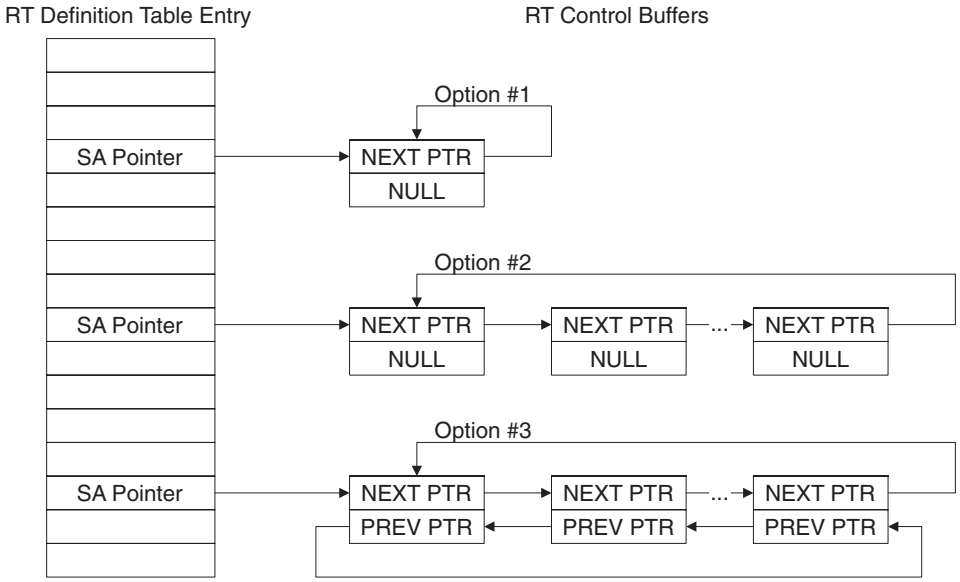


Figure 7.7.4: RT Control Buffer Allocation Options

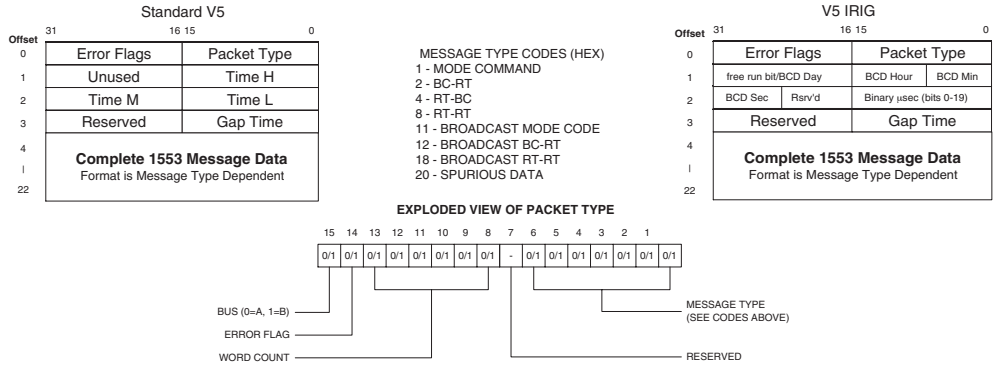
7.8 Data Buffers

MONCTRL Pointer Table Offset 12h (Associated Control Register). This control register acts as a global monitor control register for both the sequential monitor and RT monitor buffers. The bits for this register are defined below.

- ***Bit 0: Disable Capture of Spurious/Error Data.*** Set this bit to "1" to disable the sequential monitor from capturing spurious data and the sequential monitor from storing messages which contain an error. Spurious data is defined to be any data on the bus which cannot or is no longer associated with a command.
- ***Bit 1: Sequential Monitor All.*** Set this bit to "1" to result in the sequential monitor capturing bus traffic for all RTs and subaddresses. This gives the same result as setting all the bits of each RT subaddress filter word in the RT definition table. When set, this bit supersedes the sequential monitor filter word bits in the RT definition table. If this bit is "0", sequential monitoring proceeds according to the state of the RT subaddress filter words.
- ***Bit 2: Interrupt on Error.*** Set this bit to "1" to generate a MON Interrupt: Error Received, if an error is detected in a message. This interrupt applies to both RT monitoring and sequential monitoring.
- ***Bits 3–31: Reserved.*** Bits 3–31 are reserved and should be set to "0" by the host.

7.8.1 Receive Data Buffers

As shown in Figure 7.7.1 (7-19), offset 10, the RT Monitor Data Control Buffer contains a pointer to the receive (monitor) data buffer. Receive data buffers are allocated by the host beginning at 1800h in ABI-V5 memory. Each receive buffer is 23 words long and may contain a complete 1553 message. The firmware places a 1553 message in a user-allocated monitor data buffer after the message has completed. Complete 1553 messages, shown in Figure 7.8.1, include type information, a time stamp, and command, data, and status words as they occurred on the 1553 data bus.



Complete 1553 Message Data Formats (Data 2 - Big Endian)

Data 2- Big Endian

BC-RT

31 16	15 0
Reserved	Command 1
Reserved	Reserved
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32
Rcv Sts Gap	Rcv Sts Word

RT-BC

31 16	15 0
Reserved	Command 1
Trx Sts Gap	Trx Sts Word
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32

RT-RT

31 16	15 0
Command 2	Command 1
Trx Sts Gap	Trx Sts Word
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32
Rcv Sts Gap	Rcv Sts Word

Mode Code w/o ADW

31 16	15 0
Reserved	Command 1
Trx Sts Gap	Trx Sts Word

Receive Mode Code w/ADW

31 16	15 0
Reserved	Command 1
Reserved	Reserved
Data Word	Rcv Sts Gap
Rcv Sts Word	Unused

Transmit Mode Code w/ADW

31 16	15 0
Reserved	Command 1
Trx Sts Gap	Trx Sts Word
Data Word	Unused

Broadcast BC-RT

31 16	15 0
Reserved	Command 1
Reserved	Reserved
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32

Broadcast RT-RT

31 16	15 0
Command 2	Command 1
Trx Sts Gap	Trx Sts Word
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32

Spurious Data

31 16	15 0
Reserved	Reserved
Reserved	Reserved
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32

Broadcast Mode Code w/o ADW

31 16	15 0
Reserved	Command 1

Broadcast Mode Code w/ADW

31 16	15 0
Reserved	Command 1
Reserved	Reserved
Data Word	Unused

ADW = Associated Data Word

NOTE: The data formats shown above are for Data 2 "Big Endian" format.
For Data1 "Little Endian" format, the 16-bit values for data words, and if applicable, the transmit/receive status word(s) and gap(s) are swapped within each 32-bit word.

Data 2- Big Endian

Example

31 16	15 0
Trx Sts Gap	Trx Sts Word
Data Word 1	Data Word 2
|	|
Data Word 31	Rcv Sts Gap
Rcv Sts Word	Unused

Data 1- Little Endian

Example

31 16	15 0
Trx Sts Word	Trx Sts Gap
Data Word 2	Data Word 1
|	|
Rcv Sts Gap	Data Word 31
Unused	Rcv Sts Word

Figure 7.8.1: Complete Monitor Buffer Message Format

Packet Type Word

The Packet Type word identifies the message type and the number of words in the message packet (including packet type, command(s), data, status, and gaps). It also indicates whether an error occurred in the message. The word count field in the packet type word indicates the total number of 16-bit words in the message including command words, status response gaps, and data words. The word count field also includes words which are reserved due to the transfer type. The bus identification bit (bit 15) indicates the bus on which the message was received. If this bit is equal to 0, the message was received on the primary bus (bus A); if equal to 1, the message was received on the secondary bus (bus B). This word is always the last word written to the buffer by the firmware. The host may poll this location to determine when the firmware has completed writing data to the buffer. If this location is polled by the host software, it should be cleared after the buffer is processed so that the host software can determine when the next message is received.

Last Error Message Buffer

Messages containing errors do not appear in RT data buffers but are copied to the Last Error Message buffer (see the explanation of this buffer on the following pages). The last word in the buffer corresponds to the word containing the error or the position in the message where the error occurred. The Error Flags word indicates what type of error occurred and, therefore, will always be zero for messages stored in an RT data buffer. Spurious data does not appear in either the RT data buffers or the Last Error Message buffer. However, both spurious data and messages containing errors may be stored in the Sequential Monitor buffer.

TimeH, TimeM, and TimeL

For Standard V5, TimeH, TimeM, and TimeL are three 16-bit components of the 48-bit time stamp of the message and indicate the ABI-V5 time at which the first command word of the packet was received. Upon receipt of the first command word on the bus, the ABI-V5 firmware latches the current ABI-V5 time. The firmware then inserts this time in locations TimeH, TimeM, and TimeL before copying the complete 1553 message to an RT buffer.

Monitor Buffer Offsets 1, 2

For the V5 IRIG option, monitor buffer offsets 1 and 2 contain the time stamp of the message and indicate the IRIG time at which the first command word of the packet was received. The format for these words is identical to that of BC Control Block offsets 10 and 11 (see Table 6.3.1).

Gap Time

Gap Time is a 16-bit location that specifies the dead bus time between the last word of the previous message to the beginning of the first word in this message (generally the first command word). The gap value is in 250-ns tics and uses only bits 0–9 for a maximum of 256 µs.

Complete 1553 Message Data

Complete 1553 Message Data, as shown in Figure 7.8.1, contains the message as it is received from the 1553 data bus. The first command received always appears in Command Word 1. If the transfer type is RT-RT, Command Word 2 will contain the second command word. The two 16-bit locations which follow store

the status response gap and status response word from a transmit command. If the command is a receive command, these two locations are reserved. The 16-bit word locations which follow contain the data from the message. Data always starts in this location and consumes 16-bit quantities equal to the number of words in the word count of the command. Immediately following the last data word is the status response gap and status word of the receive command. If the transfer type is a transmit command, these locations have no meaning. The status gap always immediately follows the last data word of a receive command.



Note: If the word count is odd, the 16 bits following the status word are unused.



Note: Status response gap times give the dead bus time of a status gap. The gap value is in 250-ns tics and uses only bits 0–9 for a maximum of 250 µs.

7.9 Last Error Message Buffer

Messages which are received with an error are not placed in the RT buffers but, rather, are placed in the Last Error Message buffer. The Last Error Message buffer begins at offset 1DCh from the Control Base Address of the channel and is 36 words long (the maximum size of a packet). If an error occurs during a message (i.e., the Message Error bit in the Packet Type word is set to “1”), the firmware copies the message containing the error to this buffer rather than the RT data buffer. This buffer always contains the last message received by the channel which contained an error. This buffer is not affected by the state of either the on/off bit or Monitor/Simulating bit of an RT. This buffer never contains spurious data. Spurious data can only appear in the Sequential Monitor buffers.



Note: The Packet Type word is always the last word written to the buffer by the firmware. The host may poll this location to determine when the firmware has completed writing data to the buffer. If this location is polled by the host software, it should be cleared after the buffer is processed so that the host software can determine when the next message is received.

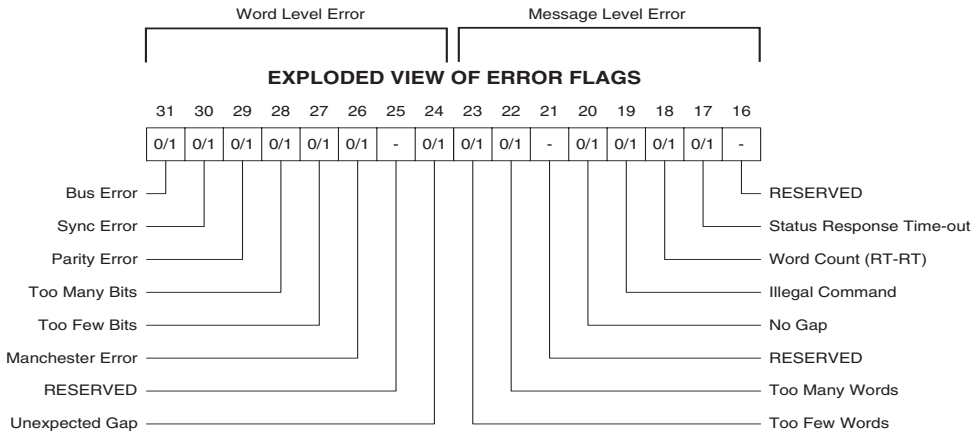


Figure 7.9.1: Error Flags Word

When an error occurs, the Error Flags word becomes nonzero, indicating the type of error that occurred in the message. The Error Flags word, shown in Figure 7.9.1, contains information concerning the packet and the type of error. Each bit in the error flags word identifies a particular type of error which occurred. Errors can be broken into two categories:

- Those which occur at a message level
- Those which occur at a word level

Bits 16–23 pertain to errors which occur at a message level and bits 24–31 pertain to errors which occur at a word level. When an error occurs at a word level, the word in which the error occurred is stored as the last word in the packet. This word is located by the host based on the message type and word count in the packet type word.

- ***Bit 16: Reserved.*** This bit is reserved for future use.
- ***Bit 17: Status Response Time-out.*** This bit indicates that a status response was not received before the time programmed in RSPGPA expired.
- ***Bit 18: Word Count (RT-RT).*** This bit indicates that the word count fields of the two command words in an RT-RT transfer did not match.
- ***Bit 19: Illegal Command.*** This bit indicates that a command word was received which appeared to be a mode code but contained an illegal value in the word count field.
- ***Bit 20: No Gap.*** This bit indicates that a gap was expected on the bus but a contiguous word was received.
- ***Bit 21: Reserved.*** This bit is reserved for future use.

- ***Bit 22: Too Many Words.*** This bit indicates that more data words were received than indicated in the associated command word.
- ***Bit 23: Too Few Words.*** This bit indicates that fewer data words were received than indicated in the associated command word.
- ***Bit 24: Unexpected Gap.*** This bit indicates that a gap was detected on the bus that was not expected according to the current command and transfer type.
- ***Bit 25: Reserved.*** This bit is reserved for future use.
- ***Bit 26: Manchester Error.*** This bit indicates that a Manchester error was detected on a word.
- ***Bit 27: Too Few Bits.*** This bit indicates that less than the expected number of bits were detected on a word before the next valid sync was detected.
- ***Bit 28: Too Many Bits.*** This bit indicates that more than the expected number of bits were detected on a word before the next valid sync was detected.
- ***Bit 29:Parity Error.*** This bit indicates that a parity error was detected on a word.
- ***Bit 30: Sync Error.*** This bit indicates that a sync error was detected on a word. This error can occur when a command sync is expected but a data sync is received, or when a data sync is expected but a command sync is received.
- ***Bit 31: Bus Error.*** This bit indicates that a word was received on the opposite bus from which the command word or words were received.

7.9.1 Transmit Data Buffers

Transmit data buffers contain data to be transmitted on the 1553 bus in response to a Transmit Command (other than a mode command). The data buffer header contains a time stamp which is explained in the description following Figure 7.7.1 on page 19.

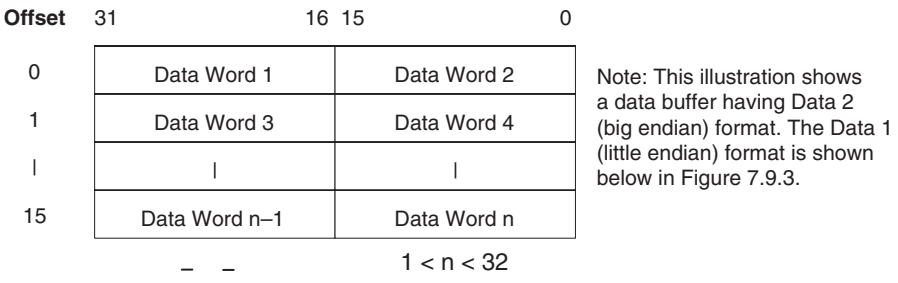


Figure 7.9.2: RT Transmit Data Buffer Format (Big Endian)

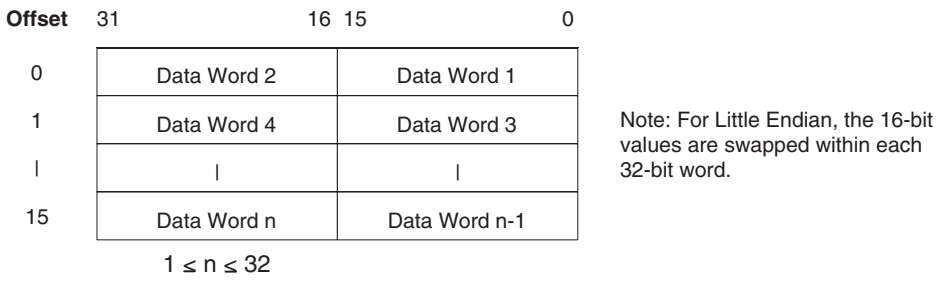


Figure 7.9.3: RT Transmit Data Buffer Format (Big Endian)

Data words 1 through *n*

Data words 1 through *n* contain the 16-bit data words for transmission. The number of 32-bit data words allocated by the host in ABI-V5 memory may range from 1 to 16. The host must pack the data words as 32-bit quantities.

8: 1553 Bus Monitoring

The ABI-V5 can perform two independent modes of bus monitoring (Sequential Monitoring and Mapped Monitoring) while performing simulation of a BC and 31 RTs. This chapter of the manual details these bus monitoring modes.



Note: The terms offset, pointer, and address are used interchangeably when referring to SBS device memory. The device memory is accessed by an offset from the device's base memory address.

Sequential Monitoring

Sequential Monitoring stores raw 1553 traffic in a linked list of buffers for real-time data logging, recording, and analysis applications. The ABI-V5 is capable of storing every 1553 message to these buffers and notifying the host system of buffer swaps through interrupt and polling methods. The messages are stored with 48-bit, 1-µs time stamps. This monitoring method is useful when the host system has tight processing constraints (relative to 1553 traffic) but must still receive every 1553 message word. The key data structures for Sequential Monitoring are the following:

- Sequential Monitor Buffers
- RT Definition Table Subaddress Filter Words
- Interrupt Queue

Mapped Monitoring

Mapped Monitoring provides user-defined linked-list data buffers for capturing RT commands and data words. This monitoring method allows the host system to set up fixed buffer locations for 1553 messages and then poll these buffers as host processing time permits. These data buffers may be chained together indefinitely, limited only by the available ABI-V5 memory. Two methods are provided for mapped monitoring RT commands (either through selective subaddress capture or capture of any message to a given RT). Mapped Monitoring is often used in conjunction with RT simulation applications where real-time polling of RT data buffers is preferred. The key data structure for Mapped Monitoring is the RT Definition Table.



Cross Reference: The details of this data structure may be found in the section *RT Definition Table* on 7-9.

Both of the these monitoring methods can use the VMEbus master capabilities of the ABI-V5. The method for performing DMA transfers of RT data buffers to VMEbus memory (usually VMEbus/host processor) is identical for mapped monitor buffers and simulated receive RT data buffers.



Cross Reference: Please see *Chapter 7: RT Simulation/RT Monitoring* for details on setting up Mapped Monitor buffers for DMA transfers.

Sequential monitor buffers may also be automatically transferred to VMEbus memory, and the control registers for each sequential monitor buffer are detailed in the following paragraphs. The use of automatic transfer of monitored 1553 data to VMEbus memory can greatly improve the system performance.



Note: Addresses listed in this chapter are 32-bit longwords.

8.1 Sequential Monitoring

Sequential Monitoring provides the host system a method of receiving every 1553 message *in the same data format as is presented on the bus*. The host system defines at least one buffer with a minimum length of 36 words. The maximum size of a sequential monitor buffer is limited by the number of buffers defined and the amount of ABI-V5 memory available. The host system also specifies which RT and subaddress (SA) combinations will have their messages stored in these buffers. Messages fill one buffer at a time. Switching to the next buffer automatically occurs only when the current buffer is filled. Each buffer in the list is filled in succession. The last buffer may be linked (pointed) to the first buffer to facilitate the continuous capture of bus traffic. Alternatively, the last buffer (which may be the only buffer defined) may contain a NULL pointer, thus limiting each buffer to be filled only once. This method provides a "snapshot" of bus traffic. The ABI-V5 firmware switches buffers when there are less than 23 words remaining in the current buffer (the size of the largest 1553 RT-RT message plus overhead words). Sequential Monitoring of data continues indefinitely until a NULL link pointer is encountered or until the host system disables this function.

Two structures must be defined before Sequential Monitoring can take place:

- The Sequential Monitor Buffer(s)
- The Interrupt Queue Buffers

The starting addresses for all these data structures must be programmed in the Pointer Table. Additionally, the subaddress filter words must be set in the RT

definition table for each combination for which data is to be captured. If interrupt on sequential monitor buffer swap or RT sequential monitor interrupt on message received are being used, allocate the interrupt queues.



Cross Reference: The following paragraphs as well as *Chapter 7: RT Simulation/RT Monitoring* provide programming details for each of the above data structures.

8.1.1 Operational Steps for Sequential Monitoring

The following steps give a summary of the procedures required to perform Sequential Monitoring functions on the ABI-V5:

1. Load the Instruction Code per the *Chapter 4: Module Startup/Test*.
2. Allocate the sequential monitor buffer(s) and place the pointer to the first of these buffers in SMFPTR.
3. Program MONCTRL as desired.
4. If the "Sequential Monitor All" option (bit 1) was not selected in MONCTRL, program the desired sequential monitor filter and swap bits in the RT Definition Table.
5. If the ABI-V5 is also performing BC or RT simulation, define these data structures prior to issuing Host Command 1.
6. Issue the "Initialize Channel" command (Host Command 1).

   The buffer length of the buffer pointed to by SMFPTR is validated; if it is valid, the pointer value is copied to SMCPTR. (If it is not valid, an entry is added to the error table.)
7. Issue the "Enable Channel" command (Host Command 2).
8. To restart the sequential monitor during 1553 processing, simply issue a "Sequential Monitor Buffers" command (Host Command 8). To initialize sequential monitoring during 1553 processing, follow steps 2, 3, and 4 and then issue a "Sequential Monitor Buffers" command.

8.1.2 Setting Sequential Monitor Filter and Swap Bits

RCV SA SM Filter, TRX SA SM Filter

Offsets 1, 2. The Receive (RCV) and Transmit (TRX) Subaddress (SA) Sequential Monitor (SM) Filter words determine which subaddresses of the RT are to be sequentially monitored. Each word consists of a bit field organized by subaddress (bit 0 = SA 0, bit 1 = SA 1, etc.). The first filter word (word offset 1 of an RT definition) sets filtering for receive subaddresses, and the second filter word (word offset 2 of an RT definition) sets filtering for transmit subaddresses. Filtering for each of the possible 64 SAs (32 receive and 32 transmit) is controlled by setting the corresponding bit of a filter word. The LSB of a filter word corresponds to subaddress 0, and the MSB corresponds to subaddress 31. *The state of a bit in the filter word indicates whether the messages for the respective RT/SA are to be placed in a Sequential Monitor buffer.* If a bit is "1", then the corresponding subaddress will be monitored, and the ABI-V5 will place messages to or from the in the sequential monitor if the message is received without error. There are no limits on the number of RT/SA messages that can be filtered or stored to buffers.

RCV SA SM Swap on Msg, TRX SA SM Swap on Msg

Offsets 3, 4. The Receive (RCV) and Transmit (TRX) Subaddress (SA) Sequential Monitor (SM) Swaps on Message words are used to force a sequential monitor buffer swap when the ABI-V5 receives an error-free message for the. Each word consists of a bit field organized by subaddress (bit 0 = SA 0, bit 1 = SA 1, etc.). The first word is for receiving subaddresses and the second for transmit. If a bit is "1", messages to that combination force the sequential monitor buffers to swap. If the combination is being sequential monitored, the firmware places the message in the current sequential monitor buffer and then swaps the buffers (the message is *always* the last message in a sequential monitor buffer).

Figure 8.1.1 shows the location of offsets 1–4 in the RT definition table.



Cross Reference: Please see the section *RT Definition Table* on 7-9 for more information.

Bit 1 of MONCTRL activates a global setting which causes all bus traffic to be captured; this supersedes the need to program the RT subaddress filter words.



Cross Reference: See the description of MONCTRL on 8-6.

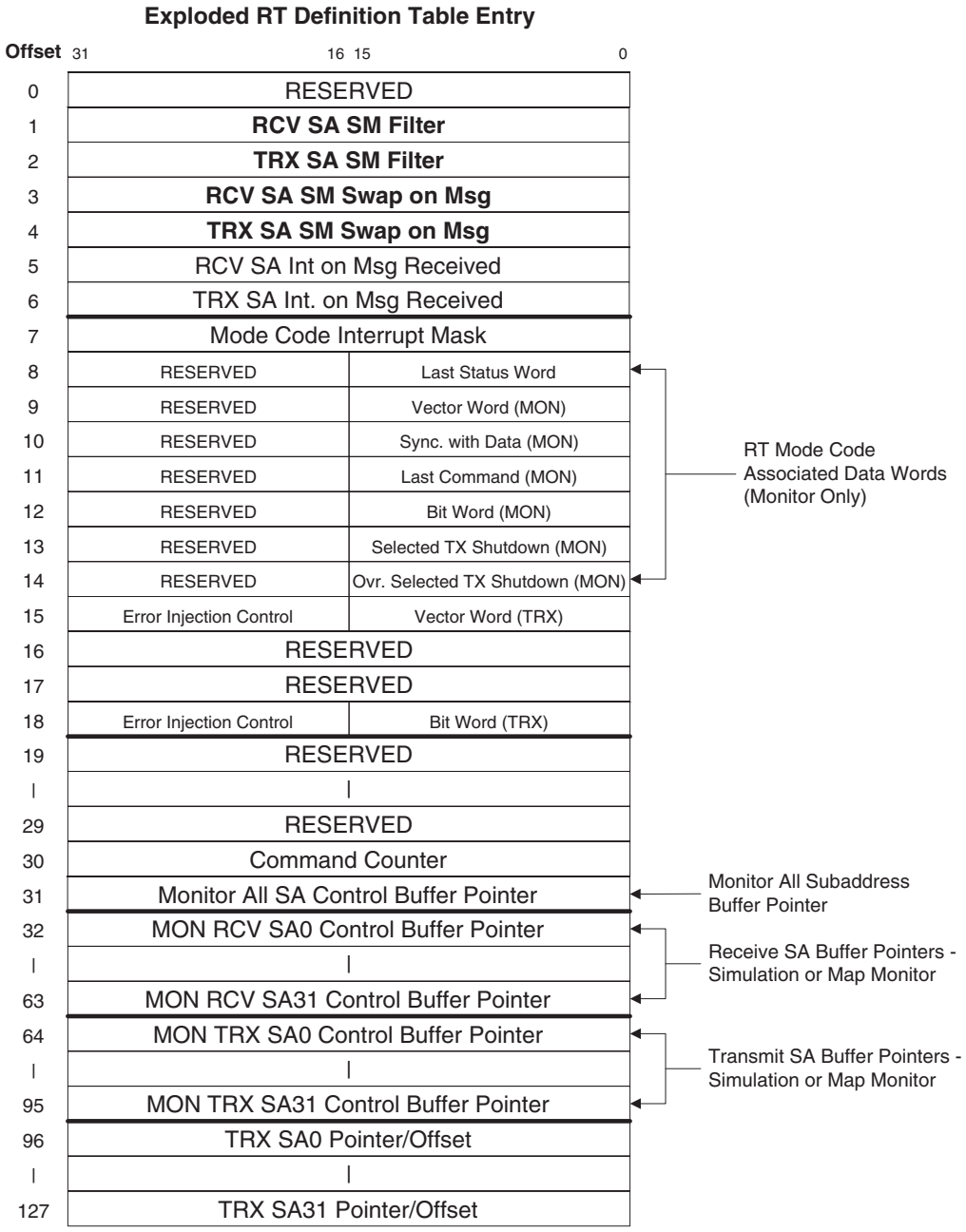


Figure 8.1.1: RT Definition Table Entry

8.1.3 Sequential Monitor Control Registers

The sequential monitor buffers provide a buffer for 1553 messages selected using the RT subaddress filter words. Five control registers control sequential monitoring operation. Four of these registers contain pointers to various sequential monitor buffers. The fifth register provides for global control of the sequential monitor feature. These five registers are defined below. These registers are offsets from the control base address of the channel.

SMFPTR Control Table Offset 0Fh. This control register contains a pointer to the first user-allocated sequential monitor buffer in a possible linked list of buffers. This register must be programmed by the user during system initialization and is not modified by the firmware during program execution.

SMCPTR Control Table Offset 10h, Updated by Firmware. This control register contains a pointer to the sequential monitor buffer currently being filled by the firmware. This register must be set to "0" prior to channel initialization. During initialization, the firmware writes the contents of SMFPTR to SMCPTR.

SMLPTR Control Table Offset 11h, Updated by Firmware. This control register contains a pointer to the last filled sequential monitor buffer. This control register must be set to "0" during system initialization.

SMBCNT Control Table Offset 13h. This control register provides an initial value which is written to the current buffer when a swap occurs. The firmware increments this number following the swap.

MONCTRL Control Table Offset 12h. This control register acts as a global monitor control register for the sequential monitor. The definition of this register is given below.

- ***Bit 0: Disable Capture of Spurious/Error Data.*** Set this bit to "1" to disable the sequential monitor from capturing spurious data and to disable both the sequential monitor and RT monitor from storing messages which contain an error. Spurious data is defined to be any data on the bus which cannot or is no longer associated with a command.
- ***Bit 1: Sequential Monitor All.*** Set this bit to "1" to result in the sequential monitor capturing bus traffic for all RTs and subaddresses. This gives the same result as setting all the bits of each RT subaddress filter word in the RT definition table. This bit supersedes the filter table sequential monitor bits when set. If this bit is "0", sequential monitoring proceeds according to the state of the RT subaddress filter words.
- ***Bit 2: Interrupt on Error.*** Set this bit to "1" to generate a MON Interrupt: Error Received, if an error is detected in a message.
- ***Bits 3–31: Reserved.*** Bits 3–31 are reserved and should be set to "0" by the host.

8.1.4 Sequential Monitor Buffers

If the sequential monitor bit is set to "1" for a particular RT/SA in the RT subaddress filter word or bit 1 of MONCTRL is set, the respective 1553 messages are logged to sequential monitor buffers. When the ABI-V5 firmware detects a 1553 message for a respective RT/SA, it time stamps the message and stores it in the next available message block of the current buffer pointed to by SMCPTR. The firmware always uses the value in SMCPTR as the starting address for the current buffer. When the current buffer is filled, the firmware performs a swap by writing the value of SMCPTR to SMLPTR and writing the next pointer of SMCPTR to SMCPTR; therefore SMCPTR always points to the current buffer and SMLPTR always points to the last buffer filled. If interrupt on buffer swap is enabled for the current buffer, an interrupt is generated when the swap is complete. This swapping process continues until a NULL link is encountered.

Figure 8.1.2 provides an illustration of the control and data word sections of the Sequential Monitor Buffers.

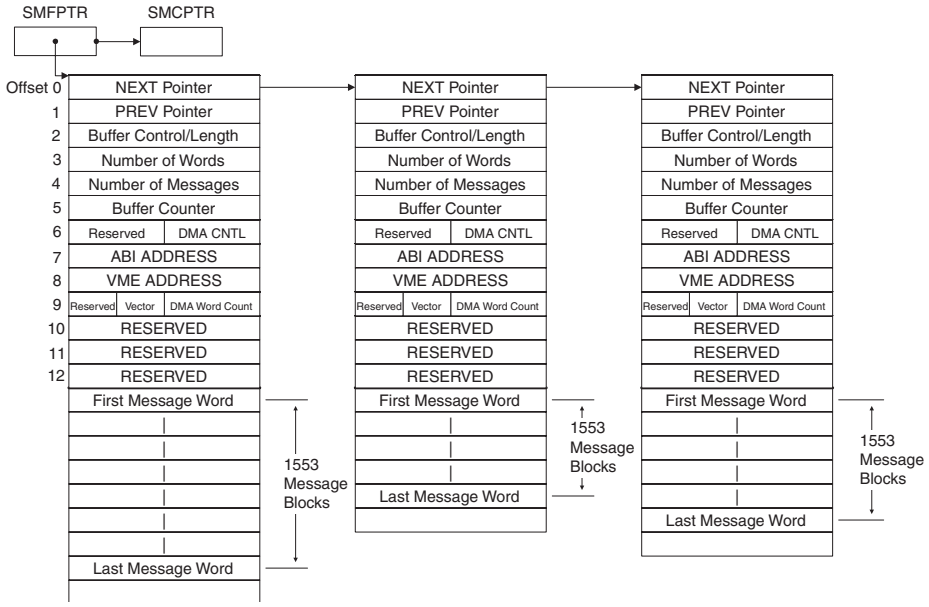


Figure 8.1.2: Sequential Monitor Buffer Data Structures

NEXT Pointer, PREV Pointer

Offset 0, Offset 1. The NEXT pointer is used to chain multiple sequential monitor buffers together. When a sequential monitor buffer swap is necessary, the firmware looks at the contents of the next pointer to determine the address of the next buffer to be used. If the value of a NEXT pointer is NULL, sequential monitoring ceases.

Two standard configurations, shown in Figure 8.1.3, are possible when linking buffers together. The PREV pointer is provided for the user to create a doubly-linked list of buffers. The use of the PREV pointer is optional and is provided for host processing.

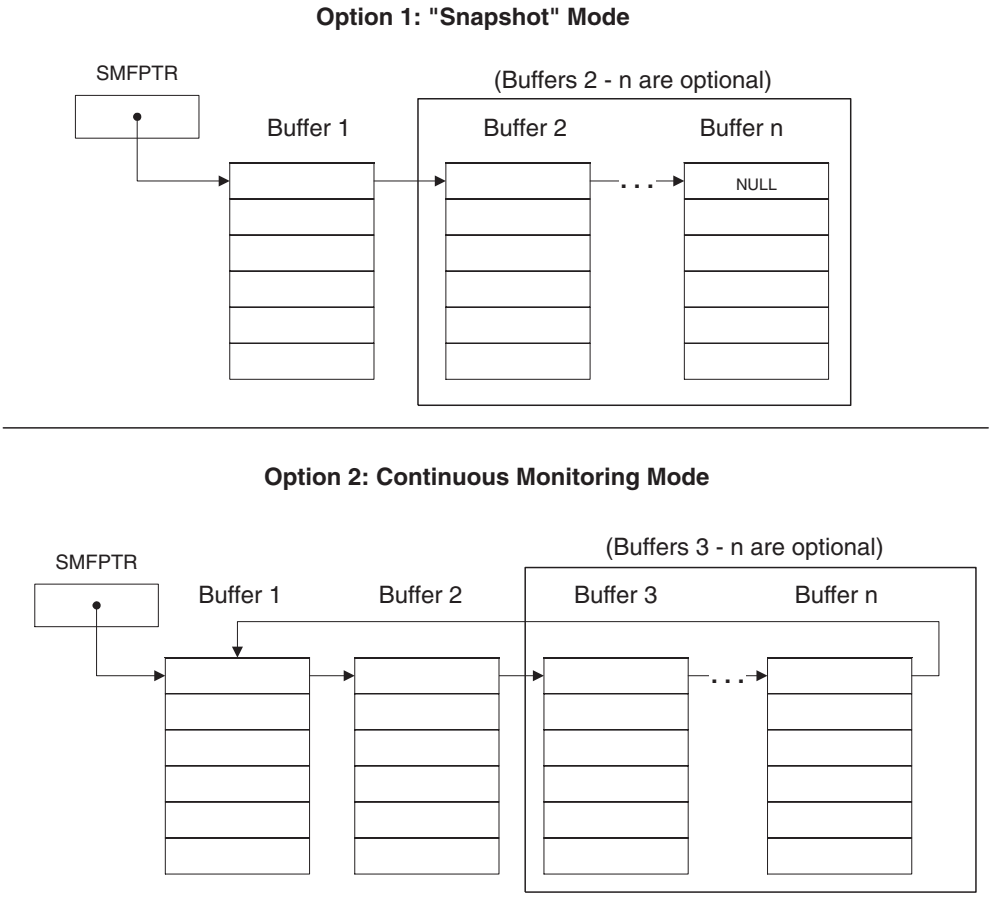


Figure 8.1.3: Buffer Linking Options

Option 1 provides a means of capturing a “snapshot” of the 1553 traffic. When this configuration is used, sequential monitoring ceases when the last buffer is filled. As shown in Figure 8.1.3, any number of buffers may be chained together. If desired, only one buffer may be used.

Option 2 is considered the normal sequential monitor configuration. When using this configuration, it is recommended that no less than two buffers be allocated. This permits the firmware to fill one buffer while the host is processing the last filled buffer (pointed at by SMLPTR). When sequential monitoring is performed with this configuration, 1553 traffic is monitored continuously.

For both options 1 and 2, when a buffer is filled, a swap occurs. If the Interrupt on Buffer Swap bit in the Buffer Control/Length word is set to “1”, interrupt 4 is generated.

Buffer Control/ Length

Offset 2. The Buffer Control/Length word defines what the length of the sequential monitor buffer is and whether an interrupt is generated when the buffer is swapped.

- ***Bits 0–20: Buffer Length.*** Use these bits to set the length of the corresponding sequential monitor buffer. Bit 0 is the LSB, and bit 20 is the MSB. Determine the size of sequential buffers by considering the host system's processing capabilities and the amount of 1553 bus loading. For example, at the maximum transfer rate of the 1553 bus (about 49K words/second), two 12.5K longword or two 25K 16-bit buffers are required to produce two buffer swaps per second (and probably two interrupts to the host system per second). Most 1553 buses have much lower transfer rates, and most host systems can process much more than two interrupts per second; therefore, the actual buffer size required for most applications is 4K–16K words.
- ***Bits 21–30: Reserved.*** Bits 21–30 are reserved and should be set to "0" by the host.
- ***Bit 31: Interrupt on Buffer Swap.*** Set this bit to "1" to cause an interrupt to be generated when a sequential monitor buffer swap has been completed. If an "Initialize Channel" host command has been issued and the host application then sets this bit to "1" in the buffer pointed to by SMCPTR, an interrupt will *not* be generated for that buffer until the next time that buffer becomes active. This is because the flag is only checked when the buffer becomes active, not when the swap occurs. In this case, issue a "Reset Sequential Monitoring" host command to activate this bit in the first buffer. It is recommended that the initial state of this bit be determined for all sequential monitor buffers prior to issuing an "Initialize Channel" host command. The state of this bit may be changed "on the fly" in any of the buffers other than the currently active buffer (i.e., the buffer pointed to by SMCPTR).

Number of Words

Offset 3, Updated by Firmware. This word contains the actual number of words in the sequential monitor buffer. When a buffer has been filled with selected 1553 messages, the firmware updates this word with the total buffer word count prior to performing a buffer swap. This word count includes the buffer control words, DMA control words, and all 1553 message words.

Number of Messages

Offset 4, Updated by Firmware. This word contains the 1553 message count of the sequential monitor buffer. Each 1553 message contains a type word, time stamp, and data words as captured from the bus. Details of the 1553 message format are given later in this chapter.

Buffer Counter Offset 5, Updated by Firmware. The buffer counter contains a number assigned to the buffer by the firmware when a buffer swap occurs. It is initialized to zero during startup and is incremented once for each monitor buffer swap and written to this location by the firmware at the time of a buffer swap.



Note: Write a nonzero value to SMBCNT to preload this register with a value other than zero.

It can be used by the host system to track message buffers and to ensure that no buffer losses occurred. The value of any buffer counter is always equal to that of the previous buffer counter plus one.

DMA CNTL, ABI ADDR, VME ADDR, WORD CNT/VECTOR Offsets 6, 7, 8, 9. These four words control DMA operations for writing sequential monitor buffers to VMEbus memory.



Cross Reference: See the subsection *Host DMA Registers* on 5-6 for the definitions of these words.

8.1.5 Message Block Format (Updated by Firmware)

The remainder of the buffer contains message blocks which consist of one type word and three time stamp words followed by the complete 1553 message as it appeared on the bus. The message block structures are illustrated in Figure 8.1.4. The type word of the message block always begins on a 32-bit boundary.

Type Word The first word of a message block consists of the packet type. The packet type word indicates the type of 1553 message that has been stored, the bus on which the message was detected, the word count of the message block (including overhead), and whether or not an error occurred. The host software can use this information to read the command word and determine the configuration of the status and data words.

- ***Bits 0–6: Message Type.*** These bits define the type of 1553 transfer for the message.
- ***Bit 7: Reserved.*** This bit is reserved for future use.
- ***Bits 8–13: Word Count.*** These bits indicate the total word count of the message, including the Type, Error Flags, Time Stamp, Command, Data, Status, and any reserved words as defined by the message type shown in Figure 8.1.4. This word count is the total number of 16-bit words in the message. It also includes any reserved 16-bit quantities.
- ***Bit 14: Error Flag.*** If this bit is "1", the message contained an error; if it is "0", no error occurred.

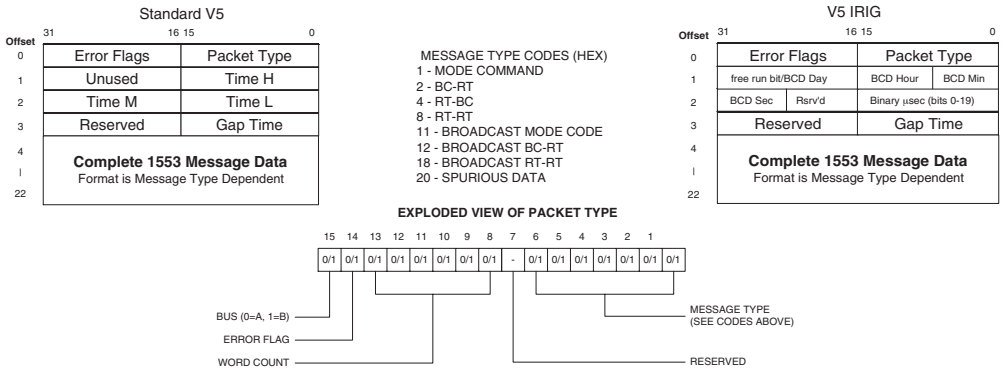
➢ ***Bit 15: Bus.*** This bit indicates the bus on which the message was detected. If this bit is "0", the message was detected on bus A; if this bit is "1", the message was detected on bus B.

TimeH, TimeM, TimeL

For the Standard V5, each BC Block contains a 48-bit time stamp, consisting of three 16-bit words, with a resolution of one microsecond. The time value is taken from the ABI-V5 hardware clock. Upon receipt of the first command word on the bus, the ABI-V5 firmware latches the current ABI-V5 time. The firmware then copies this time to locations TimeH, TimeM, and TimeL prior to copying the message to the Sequential Monitor buffer. TimeH consists of the 16 most significant bits of the time stamp, followed by TimeM and TimeL. The ABI-V5 clock can be driven from an internal or external source and may be preset by the host.

Message Block Offsets 1, 2

For the V5 IRIG time option, monitor block offsets 1 and 2 contain the time stamp of the message and indicate the IRIG time at which the first command word of the packet was received. The format for these words is identical to that of BC Control Block offsets 10 and 11 for the IRIG time option (see Table 6.3.1 on page 13).



Complete 1553 Message Data Formats (Data 2 - Big Endian)

Data 2- Big Endian

BC-RT

31 16	15 0
Reserved	Command 1
Reserved	Reserved
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32
Rcv Sts Gap	Rcv Sts Word

RT-BC

31 16	15 0
Reserved	Command 1
Trx Sts Gap	Trx Sts Word
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32

RT-RT

31 16	15 0
Command 2	Command 1
Trx Sts Gap	Trx Sts Word
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32
Rcv Sts Gap	Rcv Sts Word

Mode Code w/o ADW

31 16	15 0
Reserved	Command 1
Trx Sts Gap	Trx Sts Word

Receive Mode Code w/ADW

31 16	15 0
Reserved	Command 1
Reserved	Reserved
Data Word	Rcv Sts Gap
Rcv Sts Word	Unused

Transmit Mode Code w/ADW

31 16	15 0
Reserved	Command 1
Trx Sts Gap	Trx Sts Word
Data Word	Unused

Broadcast BC-RT

31 16	15 0
Reserved	Command 1
Reserved	Reserved
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32

Broadcast RT-RT

31 16	15 0
Command 2	Command 1
Trx Sts Gap	Trx Sts Word
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32

Spurious Data

31 16	15 0
Reserved	Reserved
Reserved	Reserved
Data Word 1	Data Word 2
|	|
Data Word 31	Data Word 32

Broadcast Mode Code w/o ADW

31 16	15 0
Reserved	Command 1

Broadcast Mode Code w/ADW

31 16	15 0
Reserved	Command 1
Reserved	Reserved
Data Word	Unused

ADW = Associated Data Word

NOTE: The data formats shown above are for Data 2 "Big Endian" format.
For Data1 "Little Endian" format, the 16-bit values for data words, and if applicable, the transmit/receive status word(s) and gap(s) are swapped within each 32-bit word.

Data 2- Big Endian

Example

31 16	15 0
Trx Sts Gap	Trx Sts Word
Data Word 1	Data Word 2
|	|
Data Word 31	Rcv Sts Gap
Rcv Sts Word	Unused

Data 1- Little Endian

Example

31 16	15 0
Trx Sts Word	Trx Sts Gap
Data Word 2	Data Word 1
|	|
Rcv Sts Gap	Data Word 31
Unused	Rcv Sts Word

Figure 8.1.4: Complete 1553 Message Formats

Gap Time

The Gap Time is a 16-bit location that specifies the dead bus time between the last word on the bus to the beginning of the first word in this message (generally the first command word). The gap value is in 250-ns tics and uses bits 0–9 only for a maximum of 256 µs. In the event that a gap occurs in the data, the Gap Time contains the value of that gap, and the data words will follow in the packet.

Data Words

Following the TIME words and gap word, the 1553 data words are stored as they are received from the 1553 data bus. The first command to be received always appears in Command Word 1. If the transfer type is RT-RT, Command Word 2 contains the second command word. The two 16-bit locations which follow are used to store the status response gap and status response word from a transmit command. If the command is a receive command, these two locations are reserved. The 16-bit word locations following the status response word contain the data from the message. Data always starts in this location and consumes 16-bit quantities equal to the number of words in the word count of the command. Immediately following the last data word is the status response gap and status word from the receive command. If the transfer type is a transmit command, these locations have no meaning. The status gap immediately follows the last data word of a receive command. Thus if the word count is odd, the 16 bits following the status word are unused.



Note: Status response gap times give the dead bus time of a status gap. The gap value is in 250-ns tics and uses only bits 0–9 for a maximum of 250 µs.

If the ABI-V5 receives a word with error, the error flags indicate what type of error occurred. The word that the error applies to or the position in the message where the error occurred corresponds to the last word in the buffer. The following section details the Error Flags word of a Sequential Monitor Entry.



Note: Spurious data never appears in an RT data buffer.

Messages Received with Errors

If an error occurs during a message, the Message Error bit in the Packet Type word is set to “1”. The Error Flags word, shown in Figure 8.1.5, becomes non-zero, indicating the type of error that occurred in the message. Each bit in the error flags word identifies a particular type of error.

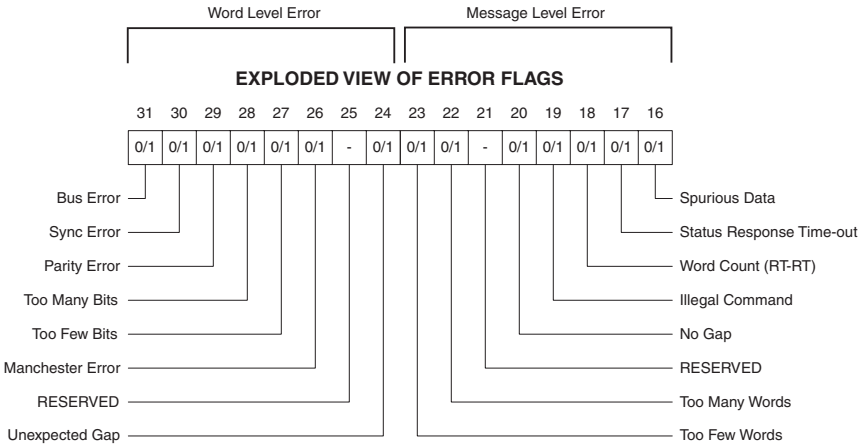


Figure 8.1.5: Error Flags Word

The error flags can be broken into two categories:

- Errors that occur at a message level
- Errors that occur at a word level

Bits 16–23 pertain to errors that occur at a message level. Bits 24–31 pertain to errors that occur at a word level. When an error occurs in a message, the associated word to which the error applies is the last word in the packet. This word is located by the host based on the message type and word count in the packet type word.

- ***Bit 16: Spurious Data.*** This bit indicates that data which can not be associated with a command was received on the bus.
- ***Bit 17: Status Response Timeout.*** This bit indicates that a status response was not received before the time programmed in RSPGPA had expired.
- ***Bit 18: Word Count (RT-RT).*** This bit indicates that the word count fields of the two command words for a RT-RT transfer did not match.
- ***Bit 19: Illegal Command.*** This bit indicates that a command word was received that appeared to be a mode code but contained an illegal value in the word count field.
- ***Bit 20: No Gap.*** This bit indicates that a gap was expected on the bus, but a contiguous word was received.
- ***Bit 21: Reserved.*** This bit is reserved for future use.
- ***Bit 22: Too Many Words.*** This bit indicates that more data words were received than indicated in the associated command word.
- ***Bit 23: Too Few Words.*** This bit indicates that fewer data words were received than indicated in the associated command word.

- ***Bit 24: Unexpected Gap.*** This bit indicates that a gap was detected on the bus that was not expected according to the current transfer type and command.
- ***Bit 25: Reserved.*** This bit is reserved for future use.
- ***Bit 26: Manchester Error.*** This bit indicates that a Manchester error was detected on a word.
- ***Bit 27: Too Few Bits.*** This bit indicates that less than the expected number of bits were detected on a word before the next valid sync was detected.
- ***Bit 28: Too Many Bits.*** This bit indicates that more than the expected number of bits were detected on a word before the next valid sync was detected.
- ***Bit 29: Parity Error.*** This bit indicates that a parity error was detected on a word.
- ***Bit 30: Sync Error.*** This bit indicates that a sync error was detected on a word. This error can occur when a command sync is expected but a data sync is received or when a data sync is expected but a command sync is received.
- ***Bit 31: Bus Error.*** This bit indicates that a word was received on the opposite bus from which the command word or words were received.

By default, the ABI-V5 firmware places errors or spurious data into the sequential monitor buffer regardless of filter word bit settings in the RT definition tables. (Spurious data refers to unexpected data which appears on the bus.) This can cause unexpected buffer swaps if 1553 words appear that are of poor transmission quality. This condition can also be caused by error conditions such as RT "no response." Disable spurious/error data monitoring by setting bit 0 of MONCTRL, as detailed on 8-6.

8.1.6 Minimum Programming Requirements for Sequential Monitor Buffers

Sequential monitor buffers are an optional data structure. If sequential monitoring is being used, SMFPTR must point to a valid sequential monitor buffer. Sequential monitor buffers have a minimum length of thirty-six 32-bit words. This includes a 13 word header and space for a minimum of one message packet of a maximum length of 23 words. Chaining of multiple buffers via the link pointer of a sequential monitor buffer is optional.

8.1.7 Buffer Swap Detection

The user can detect when a buffer swap has occurred through hardware interrupts or by monitoring SMLPTR for address changes. The ABI-V5 forces a hardware interrupt to the host system when a buffer swap occurs only if interrupts are enabled in the CSR control register (offset 0000h) and "Interrupt on Buffer Swap" has been enabled in the monitor buffer. Interrupt queues must also be defined if interrupts are being used.



Cross Reference: Please see the *Chapter 5: Host Commands/ Control Registers* and *Chapter 9: Interrupts* for more details.

The host system may detect a buffer swap by polling SMLPTR for a change of address. Upon ABI-V5 startup, SMFPTR contains the address to the first sequential monitor buffer while SMCPTR and SMLPTR both contain "0". While a sequential monitor buffer is being filled, the address of that buffer is stored in SMCPTR. When a buffer swap occurs, the address contained in SMCPTR is copied to SMLPTR, and the address contained in the link pointer of the full buffer is copied to SMCPTR. Thus, SMCPTR always contains the address of the current buffer being filled, and SMLPTR always contains the address of the last buffer filled. When a swap occurs, the firmware checks the DMA control word of the newly-filled buffer (pointed to by SMLPTR) to see if a DMA transfer has been enabled; if so, the transfer will be initiated.

8.1.8 Forcing Buffer Swaps

The host software can force the sequential monitor buffers to swap by issuing a Sequential Monitor host command using a parameter of 0.



Cross Reference: Refer to 5-29 for details on forcing a buffer swap and resetting the sequential monitoring function.

8.1.9 Resetting Sequential Monitoring

Sequential monitoring will proceed normally unless an invalid buffer length field or a NULL “next” pointer is encountered. In either case, sequential monitoring will cease regardless of bus activity or programming of RT filter words. Sequential monitoring can be restarted from the beginning of the chain of sequential monitor buffers by issuing a sequential monitor buffer host command (host command 8) with a parameter of 1.



Cross Reference: Refer to 5-29 for details on forcing a buffer swap and resetting the sequential monitoring function.

8.2 Mapped Monitoring

Mapped monitoring is typically used by data analysis systems or screen update applications. In these situations, the host computer must poll fixed locations in ABI-V5 memory for 1553 data at specific time intervals or on event interrupts. Special linked data buffers are set up by the host system to be used for this function. Data buffers for any nonsimulated RT can be indefinitely linked together, limited only by the available ABI-V5 memory. This allows the host system to configure the length of buffer lists as required by system resources. For example, if the host can read data buffers only 10 times a second, and the particular RT subaddress is updated 20 times a second, the linked-list should contain two or three buffers.

Some ABI-V5 applications combine mapped monitoring with message interrupts. The host system first programs the RT Definition Table for each desired RT to produce a hardware interrupt when a critical message has occurred (INTERRUPT ON MESSAGE RECEIVED). The host then reads the Mapped Monitor data word buffer(s) for the respective RT during the interrupt service routine.



Cross Reference: A review of the RT definition table is included in the subsection *Mapped Monitoring Data Structures* on 8-20 with Figure 8.2.1 provided for reference. Detailed discussions of RT data buffer allocation are provided in *Chapter 7: RT Simulation/RT Monitoring*.

Only messages which complete error free will be placed in an RT monitor buffer. Messages which contain an error will not appear in a RT data buffer but will be copied to the Last Error Message buffer. The Last Error Message buffer is located at word offset 01DCh from the Control Base Address of the channel.

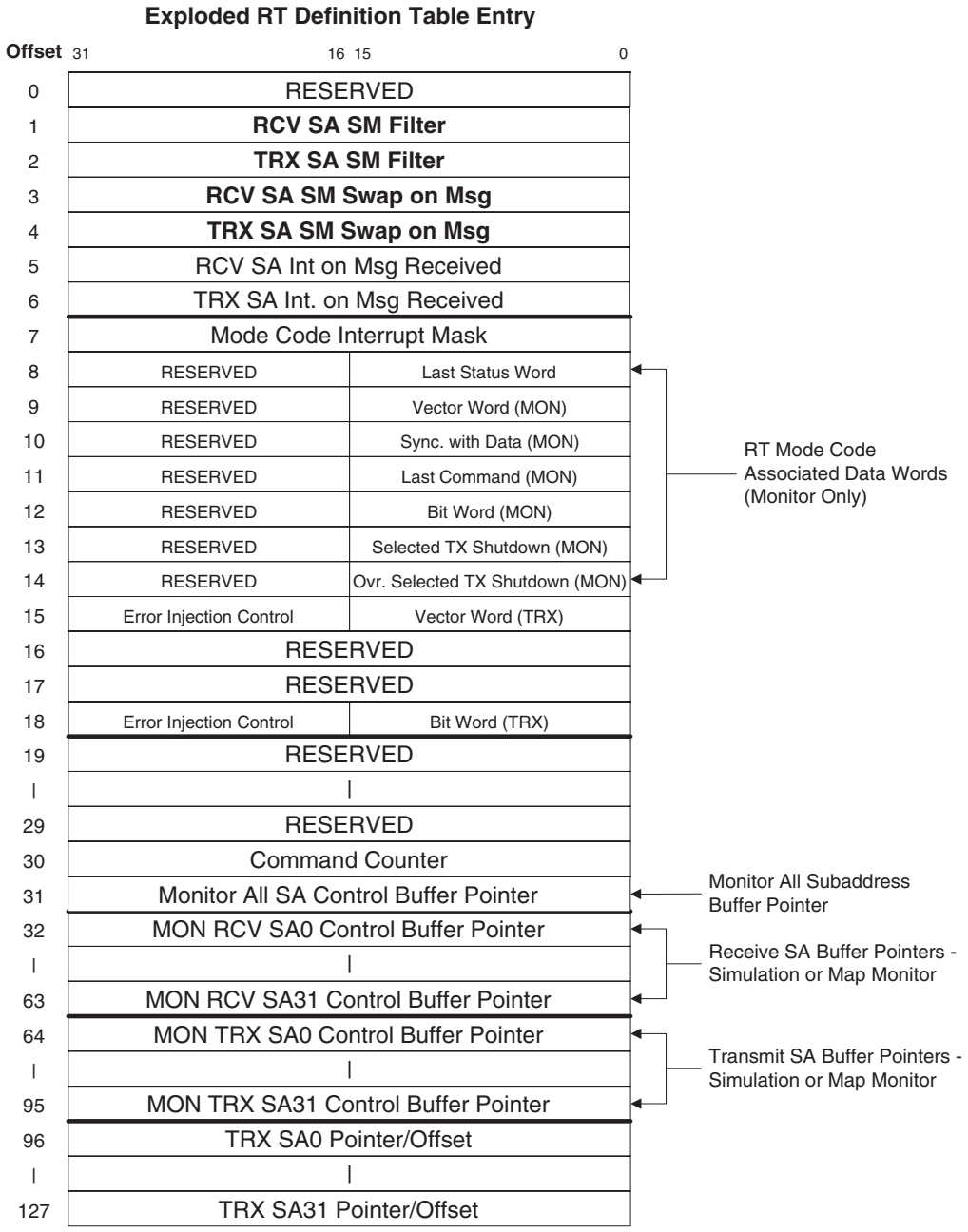


Figure 8.2.1: RT Definition Table Entry

8.2.1 Mapped Monitoring Data Structures

The key data structure for Mapped Monitoring is the RT definition table. The RT definition table is the same structure used for either RT Simulation or Mapped Monitoring. Mapped monitoring of RT data may be controlled either at the RT level of the RT/SA level.

Monitoring on an RT level is accomplished by placing a pointer to a linked list of RT data buffers in the "Monitor All SA Control Buffer Pointer" of an RT definition table entry. When this pointer is nonzero, all messages to the RT will be placed in the current buffer. As each buffer is filled, the offset of that buffer is written to "Monitor All SA Control Buffer Pointer." This provides a pointer to the last message received by the RT. Monitoring continues in this fashion until a NULL link is found in the chain of data buffers. This mode of operation overrides the monitor receive and transmit subaddress buffer pointers.

Monitoring at the RT/SA level provides the user with the means of monitoring data sent to a specific subaddress of a given RT. This mode is enabled by writing a zero to the "Monitor All SA Control Buffer Pointer" location. For each transmit or receive subaddress to be monitored, a pointer to a linked list of RT data buffers is placed in the corresponding Monitor SA pointer location of an RT Definition Table entry. Data received is placed in the buffer pointed to by the link pointer of the current Monitor SA pointer. When the complete message has been received, the Monitor SA pointer is updated with the current buffer pointer. Thus, the Monitor SA pointer always points to the last message received.

In both modes of operation, an "Interrupt on Message Received" can be generated by setting the bit for the SA in the appropriate "Interrupt on Message Received" control word. If an RT is both monitoring and simulating a transmit subaddress, only one "Interrupt on Message Received" interrupt will be generated per message.

Receive Broadcast Commands may be mapped monitored in the same manner as any other message. The RT definition table must be set up for RT address 31 with the appropriate subaddress data buffers defined.

9: Interrupts

The ABI-V5 offers extensive interrupt capabilities for notifying the host processor of BC, RT, Monitoring, and system events that have occurred on the 1553 bus. The ABI-V5 utilizes a doubled buffered queue structure to store detected or generated events for the host system interrupt service routines. The queue addresses and event counters for this data structure are managed by control registers. The host system reads these control registers during an interrupt service procedure to determine which interrupt queue is currently active and how many interrupt events are stored in the active queue.



Cross Reference: See the section *Software Control Registers (1553 Processing Control)* on 5-16 for more information on the control registers.

The host system may handle interrupt events in one of two ways:

- By polling the interrupt control registers with host software with the hardware interrupts disabled.
- By using an interrupt service routine (ISR) written for the specific device with the hardware interrupts enabled.

The ABI-V5 fills the interrupt queues and updates interrupt control registers regardless of which method is used.

The interrupt queue data structures and control registers allow the host system to easily determine what type of BC, RT, Monitor, or system event occurred. This chapter details the types of ABI-V5 interrupts and the data structures and control registers used by interrupt service procedures to manage interrupts.



Note: Addresses listed in this chapter are 32-bit longwords.

9.1 Interrupt Queue Data Structures and Control Registers

The ABI-V5 manages interrupts using double buffered queues and counter control registers. When an interrupt event occurs, the firmware stores ten words in the current interrupt queue.

The format of an interrupt queue entry is illustrated in Figure 9.1.1.

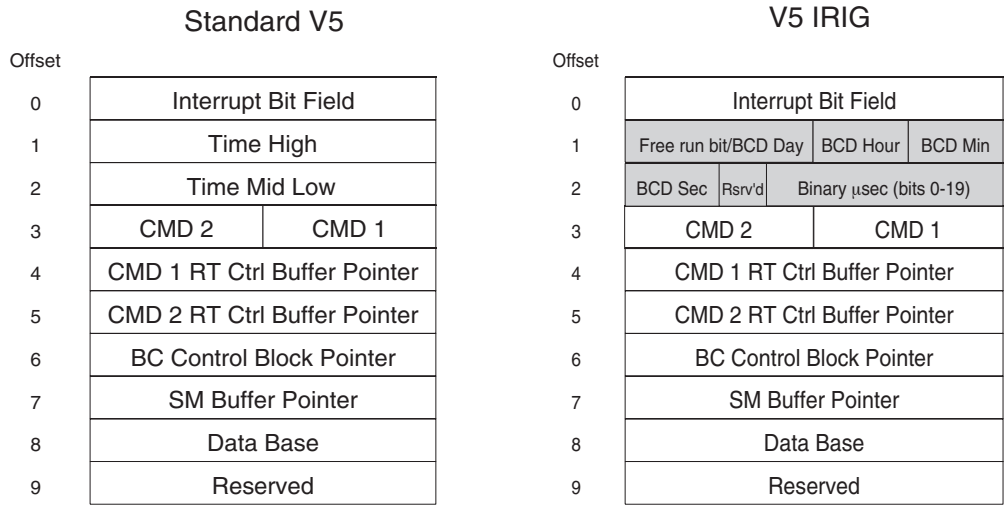


Figure 9.1.1: ABI-V5 Interrupt Queue Entry

Interrupt Bit Field Offset 0. This word indicates (by the bits which are set) the conditions which caused the interrupt to occur. The bit definitions are shown in Table 9.1.2.

Table 9.1.1: BC Control Block Offsets 10 and 11

Bit	Description
Offset 10	
0–3	1s position of BCD Minutes
4–7	10s position of BCD Minutes
8–11	1s position of BCD Hours
12–15	10s position of BCD Hours
16–19	1s position of BCD Days
20–23	10s position of BCD Days
24–27	100s position of BCD Days
28–30	Reserved
31	0 = board is in sync with an external IRIG source 1 = board is in “free run”
Offset 11	
0–19	Binary microseconds
20–23	Reserved
24–27	1s position of BCD Seconds
28–31	10s position of BCD Seconds

Table 9.1.2: Interrupt Bit Field Definitions

Bit	Definition
0	RT Interrupt: Message Received
1	RT Interrupt: Mode Code Mask
2	SM Interrupt: Buffer Swap
3	MON Interrupt: Error Received
4	RT Buffer Processed
5–7	Reserved
8	BC Interrupt: Error Received
9	BC Interrupt: Block Complete
10	BC Interrupt: BC Halt
11	BC Interrupt: Retries Attempted
12	BC Buffer Processed
13	BC Interrupt: Masked Status
14–30	Reserved
31	Interrupt Queue Overflow

If bit 31 is set, indicating an Interrupt Queue Overflow, the data from the previous interrupt entry has been overwritten.

Time High Offset 1. Bits 0–15 of this word contain the upper 16 bits of the 48-bit time stamp.

Time Low Offset 2. This word contains the lower 32 bits of the 48-bit time stamp (both middle and lower 16 bits).

IRIG Time Interrupt queue offsets 1 and 2 contain the time stamp. The format for these words is identical to that of BC Control Block offsets 10 and 11 (see Table 9.1.1 on page 9-3).

Interrupt Queue Offsets 1, 2 Interrupt queue offsets 1 and 2 contain the time stamp. The format for these words is identical to that of BC Control Block offsets 10 and 11 (see Table 6.3.1 on page 6-13).

CMD2, CMD1 Offset 3. These half-words contain the command words for the message which generated the interrupt.

CMD1 RT Control Buffer Pointer Offset 4. This word contains the pointer to the RT mapped monitor buffer (for CMD1) containing the message which generated the interrupt.

CMD2 RT Control Buffer Pointer Offset 5. This word contains the pointer to the RT mapped monitor buffer (for CMD2) containing the message which generated the interrupt.

BC Control Block Pointer Offset 6. This word contains the pointer to the BC Control Block which generated the interrupt. This word is only valid if a BC interrupt bit is set (bits 8–13).

SM Buffer Pointer Offset 7. This word contains the offset of the Sequential Monitor Buffer which generated the interrupt.

Database Offset 8. This word contains the Database value which was current when the event(s) that caused the interrupt occurred. If the value of Database is nonzero and the database control word DBCTRL bit 1 is set to "1" (word offset 1Fh in the control table), it must be added to CMD 1 RT Control Buffer Pointer, CMD 2 RT Control Buffer Pointer, and the BC Control Block Pointer to obtain the actual address of a buffer or BC control block.

9.2 Interrupt Types

There are interrupts associated with each major ABI-V5 function (BC and RT simulation and Monitoring) and interrupts for general system functions. Table 9.2.1 details the various ABI-V5 interrupt conditions. The Interrupt ID Bit column lists the bit which will be set if this interrupt condition occurs. The Cross

Reference Details column provides a cross reference to other chapters of this manual.

Table 9.2.1: ABI-V5 Interrupt Types

Type	Interrupt ID Bit	Description	Cross Reference Details
BC Simulation			
RT Response Error	8	This interrupt is generated when an error is detected in an associated RT response by the BC. The most common error is RT "no response".	6-16 BC Control Block offset 14, bit 8
BC Block Complete	9	This interrupt is generated when BC command block processing is complete.	6-16 BC Control Block offset 14, bit 9
BC Halt	10	This interrupt is generated whenever BC execution is halted. This most commonly occurs due to a zero link in the last control block of a BC chain but an RT response error or an interrupt on masked status may also cause BC execution to halt.	6-16 BC Control Block offset 14, bit 10
BC Retries Attempted	11	This interrupt is generated if retries are enabled and attempted for a BC control block.	6-16 BC Control Block offset 14, bit 11
BC Buffer Processed	12	This interrupt is generated when this data buffer has been processed and the message has completed error free.	6-20 BC Data Buffer offset 0, bit 28
Masked Status	13	This interrupt is generated when bit 12 of a BC Control Word is 1, and a nonzero value results from logically ANDing the RT Status Response word(s) in BC Control Block offset 8 with the contents of control register BCSMSK (offset 1Dh).	—
RT Simulation			
Message Received	0	This interrupt is generated if bit 1 of the Filter Table for a specific RT address and subaddress is set to 1, and the command word for the RT is received. This function is extremely useful for RT simulation and monitoring in determining when important command words have been received on the 1553 bus. The interrupt occurs after the completion of the entire BC message.	7-11 RT Definition Table offsets 5 & 6
Mode Code Command Word Detected	1	This interrupt is generated when a Mode Code command word is detected on the 1553 bus and the respective bit has been set to 1 in the Mode Command Interrupt Mask of an RT Definition.	7-11 RT Definition Table offset 7

Type	Interrupt ID Bit	Description	Cross Reference Details
RT Buffer Processed	4	This interrupt is generated when this RT/SA buffer has been processed. For monitor buffers, the interrupt is generated after the error free message is written to the buffer. For transmit buffers, the interrupt is generated after the buffer has been time stamped.	7-20 RT Control Buffer offset 2, bit 0
Monitoring			
Sequential Monitor Buffer Swap	2	This interrupt is generated when a monitor buffer swap occurs and the "interrupt on buffer swap" bit is set in the buffer control/length word. A buffer swap may occur either because the current buffer is full or because the "sequential monitor swap on message" bit is set in the RT Definition.	8-9 SM Buffer offset 2, bit 31
Error Received	3	This interrupt is generated when the "interrupt on error" bit is set in the Monitor Control word and a message containing an error is received.	5-18, 8-6 MONCTRL, bit 2
General System			
Interrupt Overflow	31	This interrupt indicates that the interrupt queue has overflowed. The host system neglected to service the first interrupt of the queue before the entire queue filled with data.	

9.3 Minimum Programming Requirements for Interrupt Queues

To use interrupts, the host software must program two interrupt queues for each channel. The queues allocated on a channel must be identical in length and must hold a minimum of four interrupt entries each. A pointer (IQPTR1 and IQPTR2) must be programmed for each queue to indicate the starting address. Each queue must also have a counter (IQCNT1 and IQCNT2) to count interrupt entries as they occur. Each entry will contain ten words.

The host programs the starting address of the first queue into IQPTR1 (offset 0004h) and the address of the second queue into IQPTR2 (offset 0005h). The host programs the length of the queues (these lengths must be identical) into control register IQNUM (offset 0008h). The value in IQNUM determines the number of entries in each queue; each entry contains ten words. For example, enter a value of "10" (000Ah) into IQNUM for a 100-word queue. Ten interrupt entries may be stored in a queue this size.

The control registers IQCNT1 and IQCNT2 (offset 0006h and 0007h) provide an event count for each buffer (there are ten words per event).

Issue Host Command 9 to initiate host processing of the interrupt queue. To begin host processing, write a FFFFFFFFh to HOSTCMD2 followed by command 00000009h to HOSTCMD1. This command causes the ABI-V5 firmware to complete any interrupt queue entries in progress to IQPTR1. The firmware then acknowledges to the host by writing a "1" to HCRESP. The firmware clears the count contained in IQCNT2 and writes any subsequent interrupts to the queue pointed at by IQPTR2. After reading a "1" in HCRESP, process the interrupts contained in IQPTR1. When the queue is completely processed, write a 00000001h to HOSTCMD2 followed by the command 00000009h to HOSTCMD1. The firmware then performs an interrupt queue buffer swap, copying the pointer in IQPTR1 to IQPTR2, and the pointer in IQPTR2 to IQPTR1. Upon completion, the firmware again acknowledges by writing a "1" to HCRESP.



Cross Reference: See section Software Polling Interrupt Service Procedure on 9-9 for detailed procedures on software polling of interrupts.

When the host services an interrupt, it reads the Interrupt Bit Field word from the interrupt entry to determine the interrupt types.

9.4 Hardware Interrupt Service Procedure

The procedure that follows is recommended for servicing ABI-V5 hardware interrupts.

1. Host receives ABI-V5 hardware interrupt and starts ISR.
2. Host ISR writes a FFFFFFFFh to HOSTCMD2 followed by command 00000009h to HOSTCMD1.
3. Firmware acknowledges by writing a "1" to HCRESP.
4. Firmware clears IQCNT2 and writes subsequent interrupts to the queue pointed at by IQPTR2.
5. After reading a "1" in HCRESP, Host ISR reads IQCNT1 (offset 6) to determine the number of interrupt events to process.
6. Host ISR reads IQPTR1 to get the address of the interrupt queue to process.
7. The host reads each interrupt event, beginning at the address indicated by IQPTR1. The host reads each interrupt event in the queue, parsing the interrupt bit field word to determine if data words are present. The host ISR may either process the interrupt data at this point or after execution has returned to the host software.
8. After completely processing the queue, Host writes 00000001h to HOSTCMD2 followed by command 00000009h to HOSTCMD1.
9. Firmware swaps interrupt queue buffers, copying the pointer in IQPTR1 to IQPTR2 and vice versa.
10. Clear the pending interrupt bit. This step is performed by the hardware for VME systems.
11. Firmware writes 0001h to HCRESP.

9.5 Software Polling Interrupt Service Procedure

The procedure that follows is recommended for software polling of ABI-V5 interrupt events.

1. Host polls (reads) bit 7 for channel 1 or bit 9 for channel 2 of the CSR (offset 0000h - D16 access only) until it detects a value of 1. A "1" indicates an interrupt event has occurred on the ABI-V5.
2. Host ISR writes a FFFFFFFFh to HOSTCMD2 followed by command 00000009h to HOSTCMD1.
3. Firmware acknowledges by writing a "1" to HCRESP.
4. Firmware clears IQCNT2 and writes subsequent interrupts to the queue pointed at by IQPTR2.
5. After reading a "1" in HCRESP, Host ISR reads IQCNT1 (offset 6) to determine the number of interrupt events to process.
6. Host ISR reads IQPTR1 to get the address of the interrupt queue buffer to process.
7. The host reads each interrupt event, beginning at the address indicated by IQPTR1. The host reads each interrupt event in the queue, parsing the interrupt bit field word to determine if data words are present. The host ISR may either process the interrupt data at this point or after execution has returned to the host software.
8. Host ISR writes a "1" to bit 7 for Channel 1 or bit 9 for Channel 2 of the CSR control register to clear the interrupt request.
9. After completely processing the queue, Host writes 00000001h to HOSTCMD2 followed by command 00000009h to HOSTCMD1.
10. Firmware swaps interrupt queue buffers, copying the pointer in IQPTR1 to IQPTR2 and vice versa.
11. Firmware writes 0001h to HCRESP.

MIL-STD-1553 INTERFACE

LIBRARIES

FOR THE ABI-V5

Reference Manual



MIL-STD-1553 Interface Libraries

Version V5-2.5 C Language
Version V5-1.1 Ada Language
Document Number V5-LIB-96302

Reference Manual

SBS Technologies, Inc.
2400 Louisiana Boulevard, NE, Building 5, Suite 600
Albuquerque, NM 87110
1-800-SBS-1553 or (505)875-0600
FAX: (505)875-0400
e-mail: sbshelp@sbs1553.com

SBS Technologies, Inc. does not warrant the operation of the program nor documentation to be error free. The contents of this manual are subject to change without notice.

Copyright (C) 1996 by SBS Technologies, Inc.
All rights reserved.

MIL-STD-1553 Library for ABI-V5 Reference Manual

Table of Contents

Introduction

Software Distribution 1-2
Ada Library 1-2
C Library 1-3
Customer Support Services 1-5

Installation

Copy the Software 2-1
Distribution Disks 2-1
Interface Libraries Disk 2-1
Driver Disk 2-2
Firmware Disk 2-3
FTP Site 2-3
Optional Tape Formats 2-3
Run the Unit Test 2-4
Creating and Installing System Software 2-4
Low Level Tuning 2-5
Tunable Parameters 2-6
Bit Representation 2-6
Initialize The Device Configuration 2-7
Device Configuration Header 2-7
Device Configuration File 2-7
Format 2-7
Exceptions 2-8
Build the Software 2-9
Building the Ada 1553 Library 2-9
Building the C 1553 Library 2-9
Define Directives for the C 1553 Library 2-9
Build the C 1553 Library 2-11

Ada Programming Notes

Library Package Descriptions 3-1
1553 Core Packages 3-1
BC_Management Package 3-2
BIT_Management Package 3-2
BM_Management Package 3-2
Interrupt_Management Package 3-2
RT_Management Package 3-2
General Programming Notes 3-3
Portability versus Efficiency 3-3
Bus Controller Notes 3-3
Ada Example Programs 3-3
ABI Ada Examples 3-4
Initializing the 1553 Device 3-4
Defining Remote Terminals and Subaddresses 3-5
Bus Controller Programming 3-6
Interrupt Processing 3-8
Bus Sequential Monitoring Enabling 3-11

C Programming Notes

General Programming Notes 4-1
Portability versus Efficiency 4-1
Bus Controller Notes 4-1
ABI Examples 4-2
Initializing the 1553 Device 4-2
Defining Remote Terminals and Subaddresses 4-3
Bus Controller Programming 4-4
Interrupt Processing 4-5
Bus Sequential Monitoring Enabling 4-8

Ada Library Reference

ABI-V5 Table of Contents 5-i
ABI-V5 Ada Reference Pages ..... 5-1 to 5-112

C Library Reference

ABI-V5 Table of Contents 6-i
ABI-V5 C Reference Pages ..... 6-1 to 6-101

Appendix A - System Software Support

Device Drivers A-2
UNIX Device Driver Interrupt Routines A-2
Encore Computer A-3
Introduction A-3
Contents of this Document A-3
Hardware/Software Requirements A-3
Installation Procedures A-4
Media Contents A-6
Distribution Tape Format A-7
HP-RT Device Driver A-8
HP-RT Device Driver Files A-8
Building The Device Drivers A-8
Kernel Memory And Device Driver Declarations A-8
LynxOS A-10
Distribution Files A-10
Driver Installation Procedures A-10
Interrupt Operation A-12
C Directives A-12
Radisys A-14
C Directives A-14
Silicon Graphics Computer System A-15
Distribution Files A-15
IRIX 5.x Installation A-15
C Directives A-16
UNIX System V A-17
C Directives A-17
VxWorks A-18
Distribution Files A-18
Driver Routines A-18
Library Routines A-19

Porting Notes A-19
Installation A-19
Execution A-20

Appendix B - Unit Test Program

Using the ABI Unit Test Application B-2
Device Management B-2
Initializing the Device B-4
Remote Terminal Management B-5
Setting Up Remote Terminals B-7
Bus Controller Management B-8
Setting Up a Bus Controller B-9
Bus Sequential Monitor Management B-10
Setting Up the Sequential Monitor B-11
Starting the Bus B-11

Table of Figures and Tables

Table 1-1.	Ada Language 1553 Library Files	1-2
Table 1-2.	Ada Language Unit Test Files	1-3
Table 1-3.	C Language 1553 Library Files	1-3
Table 1-4.	C Language Unit Test Files	1-4
Table 2-1.	C 1553 Library Directives	2-
Table B-1.	C Unit Test Files	B-1
Table B-2.	Ada Unit Test Files	B-1
Figure B-1.	Unit Test Main Menu	B-2
Figure B-2.	ABI Device Management Menu	B-3
Figure B-3.	Read/Write RAM Page	B-5
Figure B-4.	ABI Remote Terminal Management Menu	B-6
Figure B-5.	ABI Bus Controller Management Menu	B-8
Figure B-6.	ABI Bus Sequentail Monitor Menu	B-11

INTRODUCTION

SBS Technologies software is supplied at no cost with all its MIL-STD-1553 products. It is intended to give the user simple MIL-STD-1553 functionality and a head start on application development. SBS Technologies software is written in C and Ada languages and packaged into two components: a MIL-STD-1553 Library and an Unit Test application. The MIL-STD-1553 Library routines, which are called by the application software, interface with the MIL-STD-1553 board to initialize, setup, and start the 1553 bus functions. The Unit Test application tests the MIL-STD-1553 Library and provides immediate 1553 bus operations without requiring significant development effort on behalf of the user. Since our software is used with several of our 1553 products, the term *1553 device* will be used throughout this document to refer to any one of our MIL-STD-1553 products.

The purpose of this reference manual is to provide information on the use of the MIL-STD-1553 Libraries and Unit Test application software supplied with the SBS Technologies ABI-V5 MIL-STD-1553 product. This manual is divided into eight sections:

Introduction describes the software distributed with your 1553 device.

Installation describes the procedures necessary to configure the C or Ada 1553 library source code and compile it for your system.

Ada Programming Notes illustrates Ada programming examples to initialize the 1553 device and set up remote terminal, bus controller, monitor, and interrupt data structures.

C Programming Notes illustrates C programming examples to initialize the 1553 device and set up remote terminal, bus controller, monitor, and interrupt data structures.

Ada Library Reference contains 1553 library programming commands for the ABI-V5 product.

C Library Reference contains 1553 library programming commands for the ABI-V5 product.

Appendix A - System Software Support describes support software and drivers for a variety of computer systems.

Appendix B - Unit Test Program describes the use of the Unit Test application program.

Software Distribution

SBS Technologies software is normally distributed on a DC600 cartridge tape written at low density using UNIX *tar* format or on 5.25" or 3.5" diskettes using MS-DOS format.

Ada Library

The Ada 1553 Library source code distributed with the 1553 device is composed of the Ada 1553 Library and Unit Test application. The names of the Ada 1553 Library files are listed in their compilation order in Table 1-1. The names of the Unit Test application and example files are listed in their compilation order in Table 1-2

File Name	Package Name	Description
por_typ_ada	portable_types	Specification containing system types and type conversion subprograms.
por_typ.ada	portable_types	Body.
1553_typ.ada	1553_types	Specification of MIL-STD-1553 types and constants used by the ABI-V5 library.
abi_type.ada	abi_types	Specification of ABI types and constants used by the ABI-V5 library.
low_lvl_.ada	low_level	Specification containing operating system dependent subprograms.
crt_io.ada	crt_io	Specification and body for screen I/O subprograms.
low_lvl.ada	low_level	Body.
ll_parsr.ada	low_level.parse_file	Separate procedure that reads *1553dev.cfg* in order to initialize the 1553 device.
int_pkg_.ada	interrupt_management	Specification for 1553 device interrupt subprograms.
int_pkg.ada	interrupt_management	Body.
bm_pkg_.ada	bm_management	Specification for bus monitor subprograms.
bm_pkg.ada	bm_management	Body.
dev_pkg_.ada	device_management	Specification for general 1553 device management subprograms. The ABI-V5 must be properly initialized using these routines before any remote terminal, bus controller, or monitor functions can be initiated.
dev_pkg.ada	device_management	Body.
bit_pkg_.ada	bit_management	Specificaton containing Built-in-Test procedures.

File Name	Package Name	Description
bit_pkg.ada	bit_management	Body.
bc_pkg_.ada	bc_management	Specification for bus controller subprograms.
bc_pkg.ada	bc_management	Body.
rt_pkg_.ada	rt_mangement	Specification for remote terminal subprograms.
rt_pkg.ada	rt_management	Body.
1553dev.cfg		Text file containing 1553 device initialization information. This file is not compiled.
firmware.cfg		Text file containing the names of the instruction RAM and data RAM files. The instruction RAM file name is on the first line of firmware.cfg and the data RAM file name is on the second line.

Table 1-1. Ada Language 1553 Library Files

The Unit Test application source code, included for testing the library routines, may be linked with the Ada 1553 Library. The names of these files are listed below in their compilation order.

File Name	Description
unit_tes.ada	Main procedure of the Unit Test application.
bc_test.ada	Separate procedure that tests the library subprograms in bc_pkg.ada.
bm_test.ada	Separate procedure that tests the library subprograms in bm_pkg.ada.
dev_test.ada	Separate procedure that tests the library subprograms in dev_pkg.ada.
rt_test.ada	Separate procedure that tests the library subprograms in rt_pkg.ada.

Table 1-2. Ada Language Unit Test Files

C Library

The C 1553 Library source code distributed with the 1553 device is composed of the C 1553 Library and Unit Test application. The names of the 1553 Library files are listed in Table 1-3. The names of the Unit Test application and example files are listed in Table 1-4.

File Name	Description
abi_type.h	Specification of types and constants used by the ABI library.
ansi_crt.h	Screen I/O macros.
dev_conf.h	Physical address information to be used in lieu of *1553dev.sbs*.

File Name	Description
prototyp.h	Prototype file specifying all library routine calls.
sbs_incl.h	Include file containing all necessary library header files.
system.h	Directive specifications. May be deleted if directives are specified in the makefile.
crt_io.c	Screen I/O subprograms.
ll_parsr.c	Parser program that reads *1553dev.cfg* in order to initialize the device.
low_lvl.c	Operating system dependent subprograms.
bit_mgmt.c	Built in Test procedures. Not included with the ASF library.
dev_mgmt.c	General 1553 device management subprograms.
int_mgmt.c	1553 device interrupt subprograms.
rt_mgmt.c	Remote terminal subprograms.
bc_mgmt.c	Bus controller subprograms.
bm_mgmt.c	Bus sequential monitor subprograms.
1553dev.cfg	Text file containing 1553 device initialization information. This file is not compiled.

Table 1-3. C Language 1553 Library Files

The Unit Test application source code, included for testing the library routines, may be linked with the 1553 Interface Library. The names of these files are listed below.

File Name	Description
unit_tes.c	Main procedure of the Unit Test application.
dev_test.c	Procedure that tests the library subprograms in dev_mgmt.c.
rt_test.c	Procedure that tests the library subprograms in rt_mgmt.c
bc_test.c	Procedure that tests the library subprograms in bc_mgmt.c.
bm_test.c	Procedure that tests the library subprograms in bm_mgmt.c.

Table 1-4. C Language Unit Test Files

Customer Support Services

SBS Technologies offers technical support for all of its 1553 interface products and their respective library packages. We also support the installation and integration of our product and software into the user's host system. You can contact SBS Technologies via phone, fax, regular mail, or email. For questions or problems regarding our product, request technical or customer assistance at the following location:

SBS Technologies, Inc.
2400 Louisiana Boulevard, NE, Building 5, Suite 600
Albuquerque, NM 87110
Phone: 1-800-SBS-1553 or (505) 875-0600
FAX: (505) 875-0400
e-mail: sbshelp@sbs1553.com

INSTALLATION

SBS Technologies software is structured so that it can be easily ported to a variety of systems. Our software has been installed onto several types of operating systems including UNIX based systems; real-time systems such as VxWorks, pSOS, and LynxOS; and MS-DOS and VMS. These systems use a variety of bus architectures, e.g., VMEbus, Qbus, Multibus I and II, and PC/AT-ISA. The operating system routines to allocate shared memory, open the 1553 device, close the 1553 device, and handle interrupts for the systems mentioned above are included with our distribution software. However, it is the responsibility of the user to correctly configure and install these routines for a particular system. In some cases, additional operating system calls will have to be inserted in order to gain access to our board. Also, some operating systems permit the allocation of shared memory only under the strict guidelines of a device driver. The following steps are necessary for the installation of the 1553 library software.

1. Copy the Software onto your Host System
2. Run the Unit Test Executable
3. Create and Install System Dependent Software
4. Tune the Low Level Routines and Parameters
5. Select the Proper Bit Fields Representation
6. Initialize the Device Configuration
7. Build the Software

Copy The Software

The support software for SBS MIL-STD-1553 products is currently available in three formats: (1) 3.5" MS-DOS diskettes, (2) File Transfer Protocol (FTP) site files, and (3) ¼" cartridge or 4mm DAT tape. The software is included with product shipments on 3.5" disks. The remaining two formats are available to those users who cannot use this media. Each of these formats is discussed in the following paragraphs.

Distribution Disks
Support software for SBS MIL-STD-1553 products is provided on three 3.5" MS-DOS diskettes: (1) Interface Libraries Disk, (2) Driver Disk, and (3) Firmware Disk.

Interface Libraries Disk. This disk contains the source files for the Interface Libraries. All products include "C" libraries. Some products also include Ada libraries. The disk is

product-specific (ABI/ASF-PC3, ABI-V5, etc.) and contains the files in a compressed format. Run the *install.bat* file to decompress the files and copy them to your computer.

To install the software, insert the disk into the floppy drive and change directories to that drive (*a:* or *b:*). Type *install*. The installation program will attempt to install the software at *c:\sbs*. To install the software in another directory, type *install <path>*, where *<path>* is the location of your working directory (for example, *install d:\1553*). The software will be installed in the following directory structure:

(product)		-Name of the SBS product
	c_lib	-Subdirectory for the "C" libraries
	-examples	-Example "C" programs
	-library	-Source files for the "C" libraries
	ada_lib	-Subdirectory for the Ada libraries
	-examples	-Example Ada programs
	-library	-Source files for the Ada libraries
	dll	-We provide DLLs for some PC products. Currently, this directory is only included on the ABI/ASF-PC3 distribution disk.
	unittest	-For PC products only
	-abi	-Executable unit test for ABI products
	-asf	-Executable unit test for ASF products

Driver Disk. The driver disk provides the files necessary to install the interface libraries under various operating systems. See the "System Software Support" of the Interface Libraries manual for details.

The disk contains the files in a compressed format. To install the software, insert the disk into the floppy drive and change directories to that drive (*a:* or *b:*). Type *install*. The installation program will attempt to install the software at *c:\sbs*. To install the software in another directory, type *install <path>*, where *<path>* is the location of your working directory (for example, *install d:\1553*). The software will be installed in the following directory structure:

drivers
- **vme**
  - **axpvme**
  - **concrrnt**
  - **encore**
  - **harris**
  - **hp-rt**
  - **hp700**

irix
lynx
os9
sun4_1
unix5_0
vms
vxworks

pc

lynx
msdos
os2
sco
win_nt

Firmware Disk. This disk is not required for older products (ABI-V, ABI-V2, ABI-V3, ABI-V4, ASF-V, ABI-PC, ABI-PC2, ASF-PC). It is currently distributed with the ABI-V5, ABI/ASF-V6, ABI/ASF-PC3, ABI/ASF-PC104, and ABI/ASF-PCMCIA. The disk contains the firmware files which must be downloaded to the board on initialization.

FTP Site
SBS has an FTP site from which files may be retrieved. The site is

ftp.ni.net

and the SBS home directory is

/cust/sbstech

The login name for this site is **anonymous** and the password is your internet name, i.e., **user@somewhere.com**. This is the only combination of user login name and password that will work on this site.

From the home directory there are directories for each major device in the SBS family of products, such as ABIV5, ABIV6, ABIPC3, etc. Under each of these directories, you will find subdirectories for C and Ada source code, drivers, and other useful files.

Optional Tape Formats
On request, SBS will provide the interface libraries, drivers, and firmware in a TAR format on a quarter-inch cartridge tape or a 4mm DAT tape. In this case, the TAR file will contain the following directory structure:

SBS2

-abiv3

-c_lib

-examples
-library
-ada_lib
-examples
-library
-abiv4
-c_lib
-examples
-library
-abiv5
-c_lib
-examples
-library
-ada_lib
-examples
-library
-firmware
-abiv6
-c_lib
-examples
-library
-ada_lib
-examples
-library
-firmware
-drivers
(same as above for VME products)

The TAR tape will include libraries, drivers, and firmware for the ABI-V3, ABI-V4, ABI-V5, and ABI/ASF-V6. To view the contents of the tape, enter the command *tar -tvf tape_device* where *tape_device* is the input tape device. The contents of the tape should appear on the console. To read the tape into your current directory, enter *tar -xvf tape_device*.

Run The Unit Test

The Unit Test Application provided on the Interface Libraries diskette uses the 1553 library to provide basic 1553 bus operations and test library subprograms, thereby testing the 1553 device. See *Appendix B - Unit Test Application* of this manual for instructions on running the Unit Test executable.

Creating And Installing System Software

System software is additional software necessary to support the interface between the 1553 libraries and a computer system. This software may include device drivers, interrupt handlers, makefiles, etc. SBS Technologies has written and installed device drivers for our products on a

number of systems. If you need to install additional system software, please read *Appendix A - System Software Support* of this manual for a general discussion on the systems supported by SBS Technologies. If you have a system that is not supported in Appendix A, in many but not all cases, SBS Technologies will assist in writing a device driver and/or interrupt handler software designed specifically for your operating system or application.

Low Level Tuning

Operating system and hardware dependent source code has been bundled and placed in one low level file. Three routines found in the file *low_lvl.ada* in the Ada 1553 library or *low_lvl.c* in the C 1553 library may require tuning. The following paragraphs describe the operations these three low level routine must perform.

close_device()

This routine closes an opened device. In a UNIX environment, it will call a device driver close routine.

initialize_shared_memory()

This routine provides shared memory access. The actual physical address of the 1553 board is mapped to a virtual address for those operating systems that require it.

Most UNIX platforms provide a system call to allocate shared memory. The returned value of this call is a virtual address in the user's process space pointing to the first byte of mapped memory. Access to the board's memory is through this pointer. System V UNIX does not allow for a user process to map directly to physical memory; however, many of the System V UNIX distributors have added a separate system call that will allow a process access to physical memory. This call may vary depending upon the distributor.

If you are porting to the VMS operating system, VMS provides a call similar to UNIX that also initializes a shared memory segment. With VMS, the shared memory segment must be created and placed into the systems table before any application attempts to attach to it. This is usually done with a batch file that is run during the boot process. This will guarantee that the segment will be there when the user process makes the call.

open_device()

This routine opens the 1553 device for read and write access. It will call a device driver open routine in an UNIX environment. The returned file descriptor is saved in a device structure and is used for device reference by other library calls such as *close_device()*.

The software composing these low level routines is operating system dependent. These dependencies have been included in *low_lvl.ada* and *low_lvl.c*. If you are using the Ada 1553 library, edit *low_lvl.ada* and uncomment the source code needed for your system. Comment the source code that is not needed for your system. If you are using the C 1553 library, system dependent source code is automatically compiled if the proper directives have been declared. Directives are discussed below in paragraph *Define Directives for the C 1553 Library*. If you have an operating system that is not defined within these files, some research may be necessary in order to find out how to modify these routines to function with the system.

Tunable Parameters. There are several constants that should be checked before compiling the software. These constants are declared at the top of the files *abi_type.ada* and *abi_type.c*. They are described below.

byte_access	Declared only for the Ada 1553 library. This value specifies the number of bytes of data to access for each read or write. This constant is used regularly by all memory access routines. The value should be four for ABI-V5 VMEbus boards.
max_dev	The number of 1553 devices per system.
dev_config_file	Specifies the name of the device configuration file. It is usually initialized to *1553dev.cfg*.
max_chain_number	The maximum number of bus controller chains that can be used by the 1553 library.
max_rt_buffers	The maximum number of remote terminal subaddress data buffers that can be initialized for a subaddress. For example, the value would be two for double buffering.

In addition, verify that the correct instruction RAM file name is on the first line of *firmware.cfg* and that the correct data RAM file name is on the second line of *firmware.cfg*.

Bit Representation

Some compilers differ on their numbering of bits within a word, e.g., bit 0 of a word may be the least significant bit or the most significant bit. Some of the data structures in the 1553 Interface Libraries represent 1553 data words and are defined by representation clauses in Ada or bit fields in C. The ordering of the structure elements in memory is compiler dependent.

If you are using the C library, structures composed of bit fields are found in the file named *[1553_device]_type.h* where *[1553_device]* represents either *abi, asf, ame,* or *ami*. Each of these structures is declared twice and encapsulated within the directive **#ifdef BITFIELD_1**. This directive is used to select the proper structure for your system. Most VMEbus based machines

will need to have **BITFIELD_1** declared in the file *system.h* or in your makefile. For PCs and some other machines, do not declare this directive.

If you are using the Ada library, representation clauses are found in the files named *por_typ_.ada* and *abi_type.ada*. Records with representation clauses have been duplicated to accommodate two types of bit representation. Check these files and un-comment the record structure that is compatible with your compiler. These records can be easily found by searching for comments containing **$BITFIELD_1**. The *set_bit()* and *clear_bit()* functions also have been duplicated to accommodate two types of bit representation. Check these functions in por_type.ada and un-comment the function that is compatible with your compiler.

The *get_ram32()* and *put_ram32()* subprograms include a call to *low_level.swap_words()*. This call is necessary in the SBS development environment, which uses the Alsys Activada compiler on a PC with a BIT3 VME interface. This call may not be required for other development environments, so the call to *swap_words()* should be commented out. The firmware files will not be loaded correctly if the word ordering for *get_ram32()* and *put_ram32()* is not correct.

Initialize The Device Configuration

The configuration for each 1553 device in the system must be passed to the 1553 library software. Configuration parameters may include physical base address, IO address for PC boards, interrupt vector, the size of RAM, device number, device driver name, and product name. There are two ways to pass this information: (1) hard coding it in a C header file, or (2) reading a device configuration file at run time.

Device Configuration Header. A header file, named *dev_conf.h*, accompanies the C library and is primarily for embedded systems that cannot access disk files. To use *dev_conf.h* you must declare the NO_PARSE directive so that the parser will know not to read the device configuration file. Edit *dev_conf.h* and modify the **NUM_BOARDS** constant to reflect the number of boards you have in the system. Initialize the **dev_conf** array with the configuration parameters for each board. See *dev_conf.h* for examples.

Device Configuration File. The device configuration file, *1553dev.cfg*, is an ASCII text file that contains information to initialize one or more 1553 devices. (The name of this file may be changed by adjusting the tunable parameter named *dev_config_file*.) Information in this file includes the physical base address of the 1553 device, RAM size, device number, product name, an optional filename used with device drivers, and the IO base address used with PC base boards. As part of the 1553 device's initialization procedure, a library parser program is called to read the information in *1553dev.cfg*. The parser program must be able to correctly read and verify the information for a 1553 device before further library calls can proceed.

Format. The *1553dev.cfg* file is initialized by the user in a format readable by the library parser. The file is composed of keywords and values. Each keyword is associated with a value. The format for keywords and values are as follows: *<keyword>= <value>* where an = separates a keyword from a value. Below is a description of each keyword. The symbol *num* represents a

numeral value and *name* represents a character string. Keywords are in **bold** type. Four keywords are mandatory for proper initialization and two others are optional which means their use depends upon the type of 1553 device and host system.

Required Keywords	Description
device=num	A unique number identifying the 1553 product.
base_address=num	The location of the 1553 product in physical address space. The value *num* specifies the address in bytes.
ram_size=num	The amount of memory (in words) onboard the 1553 product.
product=name	The type of 1553 product. A valid value for *name* is ABI.

Optional Keywords	Description
io_base=num	Used only for PC based 1553 products and PC based host adapters. It contains the location of the 1553 device's IO register.
io_value=num	Used only for PC based 1553 products and PC based host adaptors. It contains the value to write to the 1553 device's IO register.
int_vector=num	The interrupt vector that will be used by the 1553 device.
filename=name	A text string containing the name of the device driver associated with the 1553 product.

Below is an example of *1553dev.cfg* that would set up one ABI for two channels. The device is an ABI with 64K words of memory addressed at 48000000 hexadecimal. The device specifies an IO address of 390 hexadecimal. The IO addresses are valid if the host computer is PC based; otherwise, the values are ignored.

```
# This is an example file that sets up one device for two channels.
device=1 base_address=48000000h io_base=390h ram_size=10000h io_value=3
int_vector=10 product=ABI filename=/dev/abi
device=2 base_address=48000000h io_base=390h ram_size=10000h io_value=3
int_vector=10 product=ABI filename=/dev/abi
```

Numeral values are decimal by default unless the value is followed by an *h* to indicate hexaecimal. Keywords are not case sensitive. Comments begin with # and may appear anywhere in a line. Spaces and/or tabs separate the keyword and value sets. Information per device may appear on one or more lines. There must be one valid ***base_address=***, ***ram_size=***, and ***product=*** keyword for each ***device=*** keyword. The ***device=*** keyword must precede all required and optional keywords pertaining to a particular 1553 device.

Exceptions. Exceptions will be raised by the parser for the violation of the file format. Four types of exceptions may be raised:

format_error	The *device=* not found or it does not precede its dependent keywords. It may be raised if a required keyword is not found.

invalid_keyword	A keyword was read and determined to be not valid.
invalid_value	A value was read and determined to be not valid.
constraint_error	A value is out of range. This exception is raised exclusively by the Ada 1553 library.

Each exception raised will be displayed by the parser on the screen with the following information: (1) line and column number where the exception was raised, and (2) an exception name. The parser will continue reading a *1553dev.cfg* after an exception is raised in an effort to detect all exceptions. The 1553 device will not be initialized whenever an exception is raised. Below is an example of the output displayed by the parser when an exception is raised:

```
Parser: Line  2:  Column 12:  INVALID_KEYWORD Exception
Parser: Line  3:  Column 30:  INVALID_VALUE Exception
```

Build The Software

We recommend that you begin by building an executable from the 1553 library and Unit Test application. Even though you will probably be writing your own application program, the Unit Test application is useful for quickly checking the proper installation of the 1553 board and for enabling remote terminals, the bus controller, and monitor for simple bus operations.

Building the Ada 1553 Library. It seems that each Ada compiler vendor has its own mechanisms for building software. These mechanisms may include special library installation procedures and commands, makefiles, etc. Thus, we do not provide any makefiles or special files for building the Ada 1553 library. It is the responsibility of the user to determine what procedures are necessary to compile and link the Ada 1553 library using their compiler. Please refer to the *Introduction* section of the manual for a list of the Ada library files and their compilation order. Be sure you have tuned the file *low_lvl.ada* to accommodate your system and checked for proper bit representation before compiling the code.

Building the C 1553 Library. Two steps are necessary for building the C 1553 library and Unit Test application. First, a set of directives need to be declared so that the proper system dependent source code within the 1553 library will be compiled. Second, build the 1553 library and Unit Test application using a makefile included with the libraries or one created by yourself.

Define Directives for the C 1553 Library. Directives specifying the operating system type, unique machine types, and any other specialized system definitions needed for a particular system must be defined. There are two places to define these directives: 1) in a header file named *system.h* or 2) in a makefile. To specify the directives in *system.h*, edit this file and define the appropriate directives to reflect your environment. The product type (ABI) must be declared. For example, an ABI-V board install in a VxWorks platform with a MC68020 single board computer would have the following directives:

```
#define VXWORKS                 /* VxWorks environment */
#define CPU        MC68020      /* System dependent CPU type  */
#define NORMAL                  /* Normal screen I/O */
#define BITFIELD_1       /* Declares proper field structures */
#define ABI                     /* Product type must be declared */
```

If you chose to place these directives in your makefile, the file *system.h* and its references in *sbs_incl.h* and the makefile must be deleted. The directives for VxWorks would be specified in a makefile in the following manner:

```
CCDEFINES = -DVXWORKS -DCPU=MC68020 -DNORMAL -DBITFIELD_1 -DABI
```

The following table describes the directives that may be used with the C 1553 library.

Directive	Description
ABI	Product type
BCOS2	Borland C++ compiler for OS2
BITFIELD_1	Reverses elements within bit field structures
BORLAND	Borland compiler
CPU	Processor type necessary for VxWorks
DOSREAL	MS-DOS real mode segments
EISA	EISA for the Silicon Graphics workstations version 5.x
ENCORE_UMAX	Encore computer system
HARRISUNIX	Harris Nighthawk computer system
HP700	Hewlett Packard HP-UX with I/O mapping
HP700VME	Hewlett Packard HP-VX with VME2 support
HP-RT	Hewlett Packard Real-Time operating system
IRIX	UNIX for the Silicon Graphics workstations version 4.x
IRIX5_0	UNIX for the Silicon Graphics workstations version 5.x
LYNX	LynxOS system for 386/486 or MVME167 platforms
NO_MACRO	Choose subroutines for *put_ram()* and *get_ram()* macros
NO_PARSE	Embedded systems without a disk, i.e., VxWorks
NORMAL	Normal Screen I/O
PROTOTYPES	Use function prototyping
RADISYS	Radisys system (MS-DOS to VMEbus with EPC1 32K access)
UNIXSYSVR3	UNIX System V, Release 3 system
VXWORKS	VxWorks system
WYSE50	Screen I/O for a Wyse50

Table 2-1. C 1553 Library Directives

Below are examples of directives that would be declared in *system.h* for various systems:

```
/*  HP-UX Hewlett Packard  */
#define  HP700
#define  NORMAL
#define  NO_MACRO
#define  BITFIELD_1

/*  IRIX for Silicon Graphics  */
#define  IRIX
#define  NORMAL
#define  BITFIELD_1

/*  LYNXOS for 386/486 or MVME167 Platforms  */
#define  LYNX
#define  NORMAL
#define  BITFIELD_1

/*  MS-DOS w/Borland C++ Compiler  */
#define  BORLAND
#define  PROTOTYPES

/*  RADISYS VMEbus board and Borland C Compiler  */
#define  RADISYS
#define  BORLAND
#define  DOSREAL

/*  UNIX System V system  */
#define  UNIXSYSVR3
#define  NORMAL
#define  BITFIELD_1

/*  VXWORKS environment running on a MVME147 CPU board  */
#define  VXWORKS
#define  CPU    MC68020
#define  NORMAL
#define  BITFIELD_1

/* The product type (ABI) must be declared.  */
#define ABI
```

Build the C 1553 Library. The C 1553 library can be built using one of the several makefiles and archive files that are included with the software distribution. A makefile for the ABI, named *makeabi*, is found in the subdirectories under *SBS/drivers*. You will not find makefiles with the C 1553 library source code in directory *SBS/abi/c_src/abiv5*. The makefile is structured to build the C 1553 library and Unit Test application. To build the software, follow these steps:

1. Copy the makefile for your system into the directory where the source is located.

2. Type *make -f makeabi* or the appropriate build command. The name of the executable file will be *unit_test*. See *Appendix B - Unit Test Program* for details regarding the use of this program.

ADA PROGRAMMING NOTES

This section provides general notes and examples to assist the user in the use of the 1553 library written in Ada language.

Library Package Descriptions

1553 Core Packages
The core packages of the ABI library are:

1553_types
abi_types
crt_io
device_management
low_level
portable_types

These packages, or equivalent functionality, are required for the ABI-V5 library operation.

1553_Types	The 1553 package is contained in the file 1553_TYP.ADA. This package contains common MIL-STD-1553 types and constants used by the ABI-V5 libraries.
ABI_Types	The ABI_Types package is contained in the file ABI_TYPE.ADA. This package contains ABI types and constants used by the library.
CRT_IO	The CRT_IO package is contained in the file CRT_IO.ADA. This package contains general screen input and output functions, and hexadecimal to ASCII conversions.
Device_Management	The Device_Management package is contained in the files DEV_PKG_.ADA and DEV_PKG.ADA. This package contains a set of functions and procedures that provide an interface to the 1553 device. The 1553 device must be properly initialized using these routines before any remote terminal, bus control, or monitor functions can be initiated.
Low_Level	The Low_Level package is contained in the files LOW_LVL_.ADA and LOW_LVL.ADA. This package provides a low level interface between the

	1553 Interface Library routines and the operating system. These routines open the 1553 device, initialize the shared memory segment for a 1553 device, and allocate 1553 device memory. This package contains a separate procedure, parse_file, that is contained in the file LL_PARSR.ADA. The parse_file procedure parses and reads the text file named 1553_DEV.CFG. The information in the file is used to set up and initialize the 1553 device.
Portable_Types	The Portable_Types package is contained in the files POR_TYP_.ADA and POR_TYP.ADA. This package declares types and contains type conversion subprograms used by all 1553 interface libraries.

BC_Management Package
The BC_Management package is required for ABI-V5 library Bus Controller functionality. The package is contained in the files BC_PKG_.ADA and BC_PKG.ADA. This package provides bus controller management functions for the 1553 device. These routines allow the user to build a chain program, read and write the data buffers of the chain links, and start and stop the execution of a chain.

BIT_Management Package
The BIT_Management package is required for ABI-V5 library Built-In-Test capability. The package is contained in the files BIT_PKG_.ADA and BIT_PKG.ADA This package provides the bit test routine for the V5.

BM_Management Package
The BM_Management package is required for ABI-V5 library Bus Monitor functionality. The package is contained in the files BM_PKG_.ADA and BM_PKG.ADA. This package provides bus sequential monitor management routines that allow the user to create and read the monitor buffer and force a monitor buffer swap.

Interrupt_Management Package
The Interrupt_Management package is required for ABI-V5 Interrupt_Management functionality. The package is contained in the files INT_PKG_.ADA and INT_PKG.ADA. This package provides a set of procedures to handle specific interrupts generated by the 1553 device.

RT_Management Package
The RT_Managment package is required for ABI-V5 Remote Terminal functionality. The package is contained in the files RT_PKG_.ADA and RT_PKG.ADA. This package provides remote terminal management subprograms for the 1553 device. These subprograms allow the user to define subaddresses, set values in the filter and protocol tables, read and write data buffers, etc.

General Programming Notes

The ABI libraries support device driver development (often required in a UNIX system) by offering routines to open and close a device and attach a shared memory segment to that device. A device driver ensures exclusive access to the board and provides interrupt handling capabilities. Multiple 1553 devices may be accessed via the 1553 Interface Libraries. The routine *open_1553()* must be called as part of the 1553 device initialization procedure. (In these libraries, *open_1553()* is called from *init_1553_device()*.) This function calls the operating system's routines that open a device and initialize a shared memory segment. It will return a device handle which must be used by other calls to the device. If *open_1553()* fails, all other calls to the device will fail as well. A failure may result from improperly initializing a shared memory segment or from the physical address known to the library not matching the physical address jumpered on the board. For single user operating systems such as MS-DOS, a device driver is not needed. In this case, the routines *open_1553()* and *close_1553()* do very little.

The return value from *open_1553()* is an access type referring to a record of type *device_record* which is defined in the package *abi_types*. The values of the record's components are set by the library parser when it reads the *1553dev.cfg* file and by *open_1553()*. It is important for this record to be properly initialized because it is used by all other library subprograms.

Portability versus Efficiency. Since this library must be ported to a variety of systems and Ada compilers, portability is an important factor in its design. This priority may have jeopardized the execution efficiency of the code. The use of exception handling and subroutine calls to read and write data to and from the 1553 device, i.e., *get_ram()* and *put_ram()*, can be extremely costly on certain systems. This may not be acceptable for a demanding application. Overhead can be reduced by taking advantage of vendor supplied subprogram calls where applicable, compiler optimization, and reducing the number of subroutine calls. In one case, these three techniques combined to reduce a VMEbus data word access in a user's application from 13 to 1.5 microseconds.

Bus Controller Notes. Bus controller chains are built using the procedure *add_link_to_chain()*. This procedure adds one link to a BC chain and creates a new chain if the link is the first to be added. Information regarding the block such as the contents of the command word, controls, etc., are set by the user in a structure of type *chain_link_info_type* and passed to this call.

The number of chains and the number links that may be added to any given chain depends upon the application and the amount of memory. The number of chains possible for a given application is set in the constant named *max_chain_number* found in the package *abi_types*. To start the bus controller, first load the chain using the call *load_chain()* and then call *start_io()*.

Ada Example Programs

The following two example programs are included with the distribution:

testlib.ada
schedtst.ada

The *testlib* program implements a bus controller with transmit and receive RT/SAs on each channel. The program checks for sequential monitor buffer swaps and updates all transmit buffers on command.

The *schedtst* program sets up minor frame scheduling in a bus controller block.

The programs are designed to run under the ActivAda compiler by Thomson Software Products. Complete the following steps to use these programs:

1. Enter Alsys Ada World. (If the path values are set correctly, this is done by typing **ada** from a DOS prompt.)
2. Type **inv test_lib.inv**. This will compile and link the library packages and the *test_lib* program.
3. You can then execute the *test_lib* program.
4. To compile and link the *schedtst* program, type **inv schedtst.inv**.

ABI Ada Examples

The following examples use the Ada library to illustrate how an application should 1) initialize the 1553 device, 2) define remote terminals and their subaddresses, 3) program the bus controller, 4) process interrupts, and 5) set up bus sequential monitoring functions. Please refer to the *Ada Library Reference* of this manual for a detailed explanation of the Ada 1553 library routines and their parameters. These examples do not contain exception handling which is recommended during actual use.

Initializing the 1553 Device. General initialization programs are found in the package *device_management.* The ABI must be initialized before access is permitted to any other interface library package. Two methods are available for initialization. In the first method, the user must call the following initialization routines: 1) *open_1553()*, 2) *load_firmware*, 3) *start_firmware()*, 4) *init_interrupt_queue()*, and 5) *create_bsm__ buffer()*. This method is usually used for debugging purposes. In the second method, one routine named *init_1553_device()* calls in order the routines mentioned above. In the following example, the 1553 device is initialized using method two, one routine call.

```
dev : device_handle;

BEGIN
  --
  --  Initialize the 1553 device with one call. Use 4 interrupt queue     --
 entries and an 1K sequential monitor buffer.  The return value,     --
"dev", is used in subsequent calls.
  --
  dev := device_management.init_1553_device( device_index => 1,
                                             queue_length => 4,
                                             bsm_length   => 1000 );
```

The second way to initialize the 1553 device using multiple calls.

```
dev : device_handle;

BEGIN
  --  Initialize the 1553 device with multiple calls.  Use 4 interrupt   --
 queue entries and an 1K sequential monitor buffer.  The return
  --  value, "dev", is used in subsequent calls.
  --
  --  First, open the device.
  --
  dev := device_management.open_1553 ( device_index => 1 );

  --
  -- Load the firmware and switch to application mode
  --
  device_management.load_firmware ( dev );
  device management.start_firmware ( dev );

  --
  -- Initialize the length of the interrupt queue
  --
  interrupt_management.init_interrupt_queue( device => dev,
                                             queue_length => 4 );

  --
  -- Create the sequential monitor buffer
  --
  bm_managment.create_bsm_buffer( device => dev, buffer_size => 1024 );
```

Defining Remote Terminals and Subaddresses. Subprograms for the initialization and management of remote terminals, subaddress, and their data buffers are found in the package *rt_managment*. The set up of remote terminals and subaddresses for real-time monitor mode or emulation mode cannot occur until after the ABI has been properly initialized as illustrated above. This example defines two remote terminals and enables one receive and one transmit subaddress. Each will be monitored by the sequential monitor. The subaddress data buffers will be single buffered.

```
--
-- Define RT 1 SA 2 as a receive subaddress
--
rt1sa2  : sa_info_type := ( rt           => 1,
                            sa           => 2,
                            t_r          => receive,
                            filter_bit   => true,
                              interrupt_bit=> true,
                              swap_bit     => true,
                            num_buffers  => 1,
                            buff_id      => (OTHERS => 0) );
--
-- Define RT 5 SA 10 as a transmit subaddress
--
rt5sa10 : sa_info_type := ( rt           => 5,
                            sa           => 10,
                            t_r          => transmit,
                            filter_bit   => false,
```

```
                                interrupt_bit=> false,
                                swap_bit     => true,
                               num_buffers  => 1,
                               buffer_id    => (OTHERS => 0) );

-- Initialize a transmit buffer.
sabuf : data_buffer := (1, 2, 3, 4, 5, 6, OTHERS => 0 );
wflag : uword;

BEGIN

  --
  -- Define Remote Terminals
  --
  rt_management.define_rt_sa( device  => dev,
                              sa_info => rt1sa2 );

  rt_management.define_rt_sa( device  => dev,
                              sa_info => rt5sa10 );

  --
  -- Initialize the transmit subaddress data buffer.
  --
  rt_management.write_sa_buffer( device     => dev,
                                 word_count => 6,
                                 buffer_id  => rt5sa10.buffer_id(1),
                                 out_buffer => sabuf,
                                 status     => wflag );

  --
  -- Start the 1553 Bus
  --
  device_mangement.start_io ( device => dev );
  ...
```

Bus Controller Programming. Subprograms that perform general control of the bus controller are packaged in *bc_management*. These subprograms initialize BC command chains, read and write from and to individual command blocks, and set interrupt and control flags. Set up the bus controller mode after the device has been properly initialized. In this example, a two link chain program is built. One link transmits data and the other receives data.

```
--
-- BC list block initialization.
--
bc_link1  : link_handle;
bc_link2  : link_handle;

--
-- Create and initialize a test buffer.
--
bcbuf     : data_buffer := (OTHERS => 16#C#);
bcflag    : bc_control_word;
bcerror   : error_injection_type;

BEGIN

  --
  -- Initialize link structures for a chain.
  --
  -- Build the first link of the chain program.
  --
  bc_link1 := NEW chain_link_info_type'
```

```
                                ( bc_command       => bcrt,
                                  chain_id         => 1,
                                rt1          => 1,
                                  sa1              => 2,
                                word_count   => 12,
                                  control          => bcflag,
                                link_id      => 0,
                                  loop_flag        => FALSE,
                                  error_injection  => bcerror,
                                  buff_id          => 0,
                                  num_buffers      => 2,
                                  time_delay => 0 );

   --
   -- Add link 1 to a chain.
   --
   bc_management.add_link_to_chain( device => dev,
                                    link   => bc_link1 );

   --
   -- Initialize the transmit link data buffer.
   --
   bc_management.write_link_data( device     => dev,
                                  link_id    => bc_link1.link_id,
                                  out_buffer => bcbuf );

   --
   -- Build the second link of chain program.
   --
   bc_link2 := NEW chain_link_info_type'
                                ( bc_command       => rtbc,
                                  chain_id         => 1,
                                rt1          => 5,
                                  sa1              => 10,
                                word_count   => 6,
                                  control          => bcflag,
                                link_id      => 0,
                                 loop_flag   => TRUE,
                                  error_injection  => bcerror,
                                  buff_id          => 0,
                                  num_buffers      => 1,
                                  time_delay => 0 );

   --
   -- Add link 2 to a chain.
   --
   bc_management.add_link_to_chain( device => dev,
                                    link   => bc_link2 );

   --
   -- Load the chain for execution
   --
   bc_management.load_chain( device => dev, chain_id => 1 );

   --
   -- Start the 1553 device.
   --
   device_management.start_io( device => dev );

   -- The chain program should run continuously.
...
```

Three high-priority interrupts, when enabled and occur, will automatically halt the execution of a chain. These are the BC Halt, Masked Status, and Message Error interrupts.

Interrupt Processing. There are four interrupt subprograms found in the package *interrupt_management*. The procedure *init_interrupt_queue()* is required as part of the ABI initialization. It is called from *init_1553_device()*, or if this initialization routine is not used, *init_interrupt_queue()* must be called separately. Interrupts generated to the host may be enabled or disabled using the procedures *disable_1553_interrupts()* or *enable_1553_interrupts()*.

A generic interrupt handling procedure named *handle_1553_interrupt()* is provided to manage the interrupt queue aboard the ABI, to transfer data to the host, and store the information in an interrupt log. The interrupt log is used only for general testing and would probably be unused by a genuine application program. Since interrupt handling is application specific, the handler should branch to a user provided subprogram that manages the interrupt of interest. Also, this routine is callable from an Interrupt Service Request (ISR) that is installed on the host computer by the user. Some ISRs are provided by SBS Technologies for certain computer systems. The user may use *handle_1553_interrupt()* in a polling fashion if so desired. Below is the code for *handle_1553_interrupt()*. As you can see, the procedure will determine the type of interrupt and branch to a user routine that will manage it.

```
PROCEDURE handle_1553_interrupt (device : IN device_handle) IS
    int_num, int_que1_ptr : long := 0;
    int_word              : long := 0;

    int_type : int_num_type;
    temp     : long;

FUNCTION to_int_num IS NEW unchecked_conversion (long, int_num_type);

BEGIN

  -- Make sure that a host command is not processing
  WHILE (get_ram32 (device, device.control_base_addr + hcresp) = 2) LOOP
       temp := get_ram32 (device, device.control_base_addr + hcresp);
  END LOOP;

  BEGIN
      -- Begin processing of interrupt queue
      device_management.issue_host_cmd ( device, process_int_queue,
    begin_process);

  EXCEPTION
      WHEN host_cmd_error =>
         RAISE host_cmd_error;
      WHEN host_cmd_timeout =>
         RAISE host_cmd_timeout;
  END;


  -- read the number of interrupts in the queue
  int_num := get_ram32 (device, device.control_base_addr + iqcnt1);

  -- Read the interrupt pointer
  int_que1_ptr := get_ram32 (device, device.control_base_addr + iqptr1);

  IF int_num /= 0 THEN
```

```
FOR i IN 0 .. (int_num  1) LOOP
    -- Log interrupt entry.
    int_word := get_ram32 (device, int_que1_ptr + i * 10);


    interrupt_log (device.device_number, 1, interrupt_index
(device.device_number)) := int_word;
    interrupt_log (device.device_number, 2, interrupt_index
(device.device_number)) :=
   get_ram32 (device, 1 + int_que1_ptr + i * 10);
    interrupt_log (device.device_number, 3, interrupt_index
(device.device_number)) :=
         get_ram32 (device, 2 + int_que1_ptr + i * 10);
    interrupt_log (device.device_number, 4, interrupt_index
(device.device_number)) :=
         get_ram32 (device, 3 + int_que1_ptr + i * 10);
    interrupt_log (device.device_number, 5, interrupt_index
(device.device_number)) :=
         get_ram32 (device, 4 + int_que1_ptr + i * 10);
    interrupt_log (device.device_number, 6, interrupt_index
(device.device_number)) :=
         get_ram32 (device, 5 + int_que1_ptr + i * 10);
    interrupt_log (device.device_number, 7, interrupt_index
(device.device_number)) :=
         get_ram32 (device, 6 + int_que1_ptr + i * 10);
    interrupt_log (device.device_number, 8, interrupt_index
(device.device_number)) :=
         get_ram32 (device, 7 + int_que1_ptr + i * 10);
    interrupt_log (device.device_number, 9, interrupt_index
(device.device_number)) :=
         get_ram32 (device, 8 + int_que1_ptr + i * 10);
    interrupt_log (device.device_number, 10, interrupt_index
(device.device_number)) :=
          get_ram32 (device, 9 + int_que1_ptr + i * 10);

    IF interrupt_index (device.device_number) = log_length  1 THEN
      interrupt_index (device.device_number) := 0;
    ELSE
interrupt_index (device.device_number) := interrupt_index
  (device.device_number) + 1;
    END IF;

    int_type := to_int_num (int_word);

    IF int_type.rt_message_received = true THEN
      NULL;
      -- Branch to RT Message Received Handler

      -- Increment the interrupt counter.
      interrupt_cnt := interrupt_cnt + 1;
    END IF;
    IF int_type.rt_mode_code = true THEN
      NULL;
      -- Branch to RT Mode Code Mask Handler

      -- Increment the interrupt counter.
      interrupt_cnt := interrupt_cnt + 1;
    END IF;
    IF int_type.sm_buffer_swap = true THEN
      NULL;
      -- Branch to SM Buffer Swap Handler

      -- Increment the interrupt counter.
      interrupt_cnt := interrupt_cnt + 1;
    END IF;
```

```
            IF int_type.mon_error_received = true THEN
               NULL;
               -- Branch to MON Error Received

               -- Increment the interrupt counter.
               interrupt_cnt := interrupt_cnt + 1;
            END IF;
            IF int_type.rt_buffer_processed = true THEN
               NULL;
               -- Branch to RT Buffer Processed

               -- Increment the interrupt counter.
               interrupt_cnt := interrupt_cnt + 1;
            END IF;
            IF int_type.bc_error_message = true THEN
               NULL;
               -- Branch to BC Error Received Handler

               -- Increment the interrupt counter.
               interrupt_cnt := interrupt_cnt + 1;
            END IF;

            IF int_type.bc_block_complete = true THEN
               NULL;
               -- Branch to BC Block Complete Handler

               -- Increment the interrupt counter.
               interrupt_cnt := interrupt_cnt + 1;
            END IF;
            IF int_type.bc_halt = true THEN
               NULL;
               -- Branch to BC Halt Handler

               -- Increment the interrupt counter.
               interrupt_cnt := interrupt_cnt + 1;
            END IF;
            IF int_type.bc_retries_attempted = true THEN
               NULL;
               -- Branch to BC Retries Handler

               -- Increment the interrupt counter.
               interrupt_cnt := interrupt_cnt + 1;
            END IF;
            IF int_type.bc_buffer_processed = true THEN
               NULL;
               -- Branch to BC Buffer Processed Handler

               -- Increment the interrupt counter.
               interrupt_cnt := interrupt_cnt + 1;
            END IF;
            IF int_type.int_queue_overflow = true THEN
               NULL;

               -- Branch to Interrupt Queue Overflow Handler

               -- Increment the interrupt counter.
               interrupt_cnt := interrupt_cnt + 1;
            END IF;

         END LOOP;
      END IF;

      -- Clear interrupt request
      put_ram (device, csr, get_ram (device, csr));
```

```
      -- Clear HCRESP
      put_ram32 (device, device.control_base_addr + hcresp, 0);


      BEGIN
         -- Complete the queue processing
         device_management.issue_host_cmd (device, process_int_queue,
          end_process);

      EXCEPTION
         WHEN host_cmd_error =>
            RAISE host_cmd_error;
         WHEN host_cmd_timeout =>
            RAISE host_cmd_timeout;
      END;

   END handle_1553_interrupt;
```

Bus Sequential Monitoring Enabling. Sequential monitoring is enabled for both subaddresses that were defined for the remote terminals above. The sequential monitoring may be enabled by calling *seq_mon_all_rt_sa()*, *seq_mon_all_sa*, or *seq_mon_one_sa*.

```
   BEGIN

     --
     -- Enable Sequential Bus Monitoring for two subaddresses.
     --
     seq_mon_one_sa(   device => dev,
                       sa     => rtsa2.sa );
     seq_mon_one_sa(   device => dev,
                       sa     => rtsa10.sa );
     --
     -- Start the 1553 device.
     --
     device_management.start_io(device => dev);

     -- The monitor will begin storing data in its queue.
     ...
```

C PROGRAMMING NOTES

This section provides general notes and examples to assist the user in the use of the ABI 1553 library written in C language.

General Programming Notes

The ABI library supports device driver development (often required in a UNIX system) by offering routines to open and close a device and attach a shared memory segment to that device. A device driver ensures exclusive access to the board and provides interrupt handling capabilities. Multiple 1553 devices may be accessed via the 1553 Interface Libraries. The routine *open_1553()* must be called as part of the 1553 device initialization procedure. (Note: *open_1553()* is called from *init_1553_device()*.) This function calls the operating system's routines that open a device and initialize a shared memory segment. It will return a device index which must be used by other calls to the device. If *open_1553()* fails, all other calls to the device will fail as well. A failure may result from improperly initializing a shared memory segment or from the physical address known to the library not matching the physical address jumpered on the board. For single user operating systems such as MS-DOS, a device driver is not needed. In this case, the routines *open_1553()* and *close_1553()* do very little.

The return value from *open_1553()* is a pointer to a structure of type DEVICE_RECORD which is defined in the file *abi_type.h*. The values of the record's components are set by the library parser when it reads the files *dev_conf.h* (or *1553dev.cfg*) and by *open_1553()*. It is important for this record to be properly initialized because it is used by all other library subprograms.

Portability verses Efficiency. Since this library must be ported to a variety of systems and C compilers, portability is an important factor in its design. This priority may have jeopardized the execution efficiency of the code. The use of subroutine calls to read and write data to and from the 1553 device, i.e., *get_ram(), put_ram(), get_ram32(),* and *put_ram32()* can be extremely costly on certain systems. This may not be acceptable for a demanding application. Overhead can be reduced by taking advantage of vendor supplied subprogram calls where applicable, compiler optimization, and using macros instead of subroutine calls where possible.

Bus Controller Notes. Bus controller (BC) chains are built using the procedure *add_link_to_chain()*. This procedure adds one link to a BC chain and creates a new chain if the link is the first to be added. Link information such as the contents of the command word, controls, etc., are set by the user in a structure of type *CHAIN_LINK_INFO* and passed to this call.

The number of chains and the number links that may be added to any given chain depends upon the application and the amount of memory. The number of chains possible for a given application is set in the constant named *max_chain_number* found in the file *abi_type.h*. This constant is set for an application and is used to size a table of chain pointers. The offset to the first link of each created chain is stored in the table at the offset location specified by

chain_prog_tbl_loc found in the file *abi_type.h*. This table is used by the procedure *load_chain()* to quickly look up the offset for a chain identifier and load it into an execution register. To start the bus controller, first load the chain using the call *load_chain()* and then call *start_io()*.

ABI Examples

The following examples use the C library to illustrate how an application should 1) initialize the 1553 device, 2) define remote terminals and their subaddresses, 3) program the bus controller, 4) process interrupts, and 5) set up bus sequential monitoring functions. Please refer to the *C Library Reference* section of this manual for a detailed explanation of the C 1553 library routines and their parameters. These examples do not contain exception handling which is recommended during actual use.

Initializing the 1553 Device. General initialization programs are found in the file *dev_mgmt.c*. The ABI must be initialized before access is permitted to any other interface library package. Two methods are available for initialization. In the first method, the user must call the following initialization routines: 1) *open_1553()*, 2) *load_firmware()*, 3) *start_firmware()*, and 4) *initialize channel()*. The user may optionally call 1) *init_interrupt_queue()* and 2) *create_bsm_buffer()*. This method is usually used for debugging purposes. In the second method, one routine named *init_1553_device()* calls in order the routines mentioned above. In the following example, the 1553 device is initialized using method two, one routine call.

```
#include "sbs_incl.h"       /* Include this header file */

/*  Device initialization.  */
DEVICE_HANDLE     dev;
EXCEPTION_RETURN  status;

  /*
   *   Initialize the 1553 device with one call.  Use 4 interrupt queue entries
   *   and an 1K sequential monitor buffer.  The return value, "dev", is used
   *   in subsequent calls.
   */
  dev = init_1553_device (1, 4, 1000);
```

The second way to initialize the 1553 device using multiple calls.

```
#include "sbs_incl.h"

/*  Device initialization.  */
DEVICE_HANDLE     dev;
EXCEPTION_RETURN  status;

  /*
   *  Initialize the 1553 device with multiple calls.  Use 4 interrupt queue
   *  entries and an 1K sequential monitor buffer.  The return value, "dev",
   *  is used in subsequent calls.
   *
   *  First, open the device.
   */
  dev = open_1553 ( 1 );
```

```
  /*
   * Start the microcode on the board and switch to application mode
   */
  load_firmware ( dev );
  start_firmware ( dev );

  /*
   * Initialize the length of the interrupt queue.
   */
  init_interrupt_queue( dev, 4 );

  /*
   * Create the sequential monitor buffer.
   */
  create_bsm_buffer( dev, 1024 );

  /*
   * Initialize the channel
   */
  initialize_channel(dev);
```

Defining Remote Terminals and Subaddresses. Subprograms for the initialization and management of remote terminals, subaddress, and their data buffers are found in the file *rt_mgmt.c*. The set up of remote terminals and subaddresses for real-time monitor mode or emulation mode cannot occur until after the ABI has been properly initialized as illustrated above. This example defines two remote terminals and enables one receive and one transmit subaddress. Each will be monitored by the sequential monitor. The subaddress data buffers will be single buffered.

```
/*
 * RT initialization.
 *
 * Define RT 1 SA 2 as a receive subaddress.
 */
SA_INFO_TYPE    rt1sa2    = { 1,  2, RECEIVE, 0};

/*
 * Define RT 5 SA 10 as a transmit subaddress.
 */
SA_INFO_TYPE    rt5sa10   = { 5, 10, TRANSMIT, 0};

/* Initialize a transmit buffer. */
unsigned short  sabuf[32] = { 1, 2, 3, 4, 5, 6 };

  /*
   * Define Remote Terminals.
   */
  status = define_rt_sa( dev,  &rt1sa2  );
  status = define_rt_sa( dev,  &rt5sa10 );

  /* Initialize the transmit subaddress data buffer.  */
  status = write_sa_buffer( dev, 6, rt5sa10.buff_id[0], sabuf );

  /*
   * Start the 1553 Bus.
   */
  start_io ( dev );
  ...
```

Bus Controller Programming. Subprograms that perform general control of the bus controller are found in the file *bc_mgmt.c*. These subprograms initialize BC command chains, read and write from and to individual command blocks, and set interrupt and control flags. Set up the bus controller mode after the device has been properly initialized. In this example, a two link chain program is built. One link transmits data and the other receives data.

```
/*
 * BC link variables
 *
 */
CHAIN_LINK_INFO  bc_link1;
CHAIN_LINK_INFO  bc_link2;

/* Create and initialize a data buffer */
VALID_DATA_WORD bcbuf[32] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

/*
 * BC list block initialization.
 *
 * BCRT command for RT=1, SA=2, and WC=12
 *
 */
bc_link1.command = bcrt;
bc_link1.chain_id = 1;
bc_link1.rt1 = 1;
bc_link1.sa1 = 2;
bc_link1.word_count = 12;
bc_link1.num_buffers = 1;
bc_link1.errors.errors_ =0;
bc_link1.flags.flags_ =0;
bc_link1.loop_flag = TRUE;

/*
 *
 * BCRT command for RT=5, SA=10, and WC=6
 *
 */
bc_link2.command = rtbc;
bc_link2.chain_id = 1;
bc_link2.rt1 = 5;
bc_link2.sa1 = 10;
bc_link2.word_count = 6;
bc_link2.num_buffers = 1;
bc_link2.errors.errors_ =0;
bc_link2.flags.flags_ =0;
bc_link2.loop_flag = TRUE;

/*
 *   Initialize the 1553 device with one call.  Use 4 interrupt queue entries
 *   and an 1K sequential monitor buffer.  The return value, "dev", is used
 *   in subsequent calls.
 */
dev = init_1553_device (1, 4, 1000);

/* Add the first link to the BC chain */
add_link_to_chain(dev, &bc_link1);

/* Initialize the transmit link data buffer */
write_link_data(dev, bc_link1.link_id, bc_link1.buff_id[0], bcbuf);

/* Add the second link to the BC chain */
add_link_to_chain(dev, &bc_link2);
```

```
/* Start the 1553 device */
start_io( dev );

/* Load the chain for execution */
load_chain(dev, 1);

/* The chain program should run continuously */
...
```

Interrupt Processing. There are four interrupt subprograms found in the file *int_mgmt.c*. The procedure *init_interrupt_queue()* is optionally part of the ABI initialization. It is called from *init_1553_device()*, or if this initialization routine is not used, *init_interrupt_queue()* may be called separately. Interrupts generated to the host may be enabled or disabled using the procedures *disable_1553_interrupts()* or *enable_1553_interrupts()*.

A generic interrupt handling procedure named *handle_1553_interrupt()* is provided to manage the interrupt queue aboard the ABI, to transfer data to the host, and store the information in an interrupt log. The interrupt log is used only for general testing and would probably be unused by a genuine application program. Since interrupt handling is application specific, the handler should branch to a user provided subprogram that manages the interrupt of interest. Also, this routine is callable from an Interrupt Service Request (ISR) that is installed on the host computer by the user. Some ISRs are provided by SBS Technologies for certain computer systems. The user may use *handle_1553_interrupt()* in a polling fashion if so desired. Below is the code for *handle_1553_interrupt()*. As you can see, the procedure will determine the type of interrupt and branch to a user routine that will manage it.

```
EXCEPTION_RETURN  handle_1553_interrupt (device)
DEVICE_HANDLE   device;
{
  ULONG int_num;
  int   i;
  ULONG int_que1_ptr;

  union
  {
    INTERRUPT_CODE  bits;
    ULONG           word;
  } int_bitfield;

  /* Make sure that a host command is not processing */
  while ((get_ram32(device, device->control_base_addr+HCRESP))== 2)
  {
    get_ram32(device, device->control_base_addr+HCRESP);
  }

  /* Clear HCRESP */
  put_ram32(device, device->control_base_addr+HCRESP, 0);

  /* Begin processing of interrupt queue */
  put_ram32(device, device->control_base_addr+HOSTCMD2, 0xffffffffl);
  put_ram32(device, device->control_base_addr+HOSTCMD1, 9);

  /* wait while HCRESP is zero */
  while ((get_ram32(device, device->control_base_addr+HCRESP))== 0)
  {
    get_ram32(device, device->control_base_addr+HCRESP);
```

```
}

/* wait while HCRESP is processing */
while ((get_ram32(device, device->control_base_addr+HCRESP))== 2)
{
  get_ram32(device, device->control_base_addr+HCRESP);
}

/* If no error has occurred */
if ((get_ram32(device, device->control_base_addr+HCRESP))== 1)
{
  /*  Read the number of interrupts in the queue  */
  int_num   = get_ram32(device, device->control_base_addr+IQCNT1);

  /* Read the interrupt pointer */
  int_que1_ptr = get_ram32(device, device->control_base_addr+IQPTR1);

  for (i = 0; i < int_num; i++)
  {
    /*  Log interrupt entry.  */
    int_bitfield.word = get_ram32(device, int_que1_ptr + i * 10);

    if((int_bitfield.word&0x80000000L)== 0)
      int_bitfield.bits.interrupt_overflow = 0;
    else
      int_bitfield.bits.interrupt_overflow = 1;

    interrupt_log[0][interrupt_index]=int_bitfield.word;
    interrupt_log[1][interrupt_index]=get_ram32(device,1+int_que1_ptr+i*10));
    interrupt_log[2][interrupt_index]=get_ram32(device,(2+int_que1_ptr+i*10));
    interrupt_log[3][interrupt_index]=get_ram32(device,(3+int_que1_ptr+i*10));
    interrupt_log[4][interrupt_index]=get_ram32(device,(4+int_que1_ptr+i*10));
    interrupt_log[5][interrupt_index]=get_ram32(device,(5+int_que1_ptr+i*10));
    interrupt_log[6][interrupt_index]=get_ram32(device,(6+int_que1_ptr+i*10));
    interrupt_log[7][interrupt_index]=get_ram32(device,(7+int_que1_ptr+i*10));
    interrupt_log[8][interrupt_index]=get_ram32(device,(8+int_que1_ptr+i*10));
    interrupt_log[9][interrupt_index]=get_ram32(device,(9+int_que1_ptr+i*10));

    if (interrupt_index == LOG_LENGTH)
      interrupt_index = 0;
    else
      interrupt_index++;

    /*  Branch to the User's respective handlers.  */
    if (int_bitfield.bits.rt_message_received)
    {
      /* Branch to RT Message Received Interrupt Handler */
    }

    if (int_bitfield.bits.rt_mode_code_mask)
    {
      /* Branch to RT Mode Code Mask Interrupt Handler */
    }

    if (int_bitfield.bits.monitor_buffer_swap)
    {
      /* Branch to SM Buffer Swap Interrupt Handler */
    }

    if (int_bitfield.bits.monitor_error_received)
    {
      /* Branch to MON Error Received Handler */
    }

    if (int_bitfield.bits.bc_error_received)
```

```
        {
            /* Branch to BC Interrupt Error Received Handler */
        }

        if (int_bitfield.bits.bc_block_complete)
        {
          /* Branch to BC Block Complete Handler */
        }

        if (int_bitfield.bits.bc_halt)
        {
          /* Branch to BC Halt Handler */
        }

        if (int_bitfield.bits.retries_attempted)
        {
          /* Branch to BC Retries Attempted Handler */
        }

        if (int_bitfield.bits.interrupt_overflow)
        {
          /* Branch to Interrupt Queue Overflow Handler */
        }
      }
      /* Clear interrupt request */
      put_ram(device, CSR, get_ram(device, CSR));

      /* Clear HCRESP */
      put_ram32(device, device->control_base_addr+HCRESP, 0);

      /* Complete queue processing */
      put_ram32(device, device->control_base_addr+HOSTCMD2, 1);
      put_ram32(device, device->control_base_addr+HOSTCMD1, 9);

      /* wait while HCRESP is zero */
      while ((get_ram32(device, device->control_base_addr+HCRESP))== 0)
      {
        get_ram32(device, device->control_base_addr+HCRESP);
      }

      /* wait while HCRESP is processing */
      while ((get_ram32(device, device->control_base_addr+HCRESP))== 2)
      {
        get_ram32(device, device->control_base_addr+HCRESP);
      }

      if ((get_ram32(device, device->control_base_addr+HCRESP))== 1)
      {
        return (SUCCESS);
      }
      else
      {
        return (FAILURE);
      }
    }
    else /* HCRESP is 3 */
    {
      return (FAILURE);
    }
}
```

Bus Sequential Monitoring Enabling. Sequential monitoring is enabled for both subaddresses that were defined for the remote terminals above. The sequential monitoring may be enabled by calling *seq_monitor_all_rt_sa*, *seq_mon_all_sa*, or *seq_mon_one_sa*.

```
/*
 *  Enable Sequential Bus Monitoring for two subaddresses
 */
seq_mon_one_sa( dev, rt1sa2->sa );
seq_mon_one_sa( dev, rt5sa10->sa );

/*
 * Start the 1553 device.
 */
start_io(dev);

/* The monitor will begin storing data in its queue. */
...
```

ADA LIBRARY REFERENCE

SBS Technologies' MIL-STD-1553 Ada support library routines provide a variety of operations useful to the application program. This section contains the Ada library programming commands for ABI-V5 MIL-STD-1553 interface boards.

Contents

ABI-V5 Table of Contents	5-i
ABI-V5 Ada Library Reference	5-1 through 5-112

ABI-V5 Ada Library Reference

Table of Contents

Device Management Routines

close_1553 5-1
init_1553_device 5-2
initialize_channel 5-4
issue_host_cmd 5-5
load_firmware 5-6
open_1553 5-7
read_time 5-9
reset_channel 5-10
set_host_dma 5-12
start_firmware 5-13
start_io 5-14
stop_io 5-16
stub_select 5-17

Remote Terminal Routines

define_rt_mon_sa 5-18
define_rt_sa 5-19
read_filter_word 5-21
read_mc17_data 5-22
read_rt_int_word 5-24
read_sa_buffer 5-25
read_status_word 5-27
read_swap_word 5-29
read_transmit_sa_buffer 5-31
rt_wrap 5-33
set_filter_word 5-35
set_global_control 5-37
set_mc16_data 5-38
set_mc19_data 5-39
set_mim 5-40
set_rt_dma 5-42
set_rt_int_word 5-43
set_status_word 5-45
set_swap_word 5-47
set_trx_status_control 5-49
update_rt_stat_resp_entry 5-50
write_sa_buffer 5-51

Bus Controller Routines

add_link_to_chain 5-53
delink_chain 5-57
disable_frame_timer 5-59
enable_frame_timer 5-60
link_chain 5-61
load_chain 5-63
read_bc_data_buff_control 5-65
read_link_data 5-67
read_link_status 5-69
set_bc_control 5-71
set_bc_data_buff_control 5-72
set_bc_dma 5-74
set_error_word 5-76
stop_chain 5-77
update_bc_control_info 5-78
write_link_data 5-80

Bus Monitor Routines

create_bsm_buffer 5-82
force_buffer_swap 5-84
read_bsm_buffer 5-85
seq_mon_all_clear 5-86
seq_mon_all_rt_sa 5-87
seq_mon_all_sa 5-88
seq_mon_clear_all_sa 5-89
seq_mon_clear_one_sa 5-90
seq_mon_one_sa 5-91
set_bsm_dma 5-92
set_int_on_swap 5-93

Interrupt Management Routines

disable_1553_interrupts 5-94
enable_1553_interrupts 5-95
handle_1553_interrupt 5-96
init_interrupt_queue 5-97

BIT Management Routines

perform_bit 5-99

Low Level Routines

close_1553 5-101

fetch_from_address 5-102
get_ram32 5-104
initialize_shared_memory 5-105
load_ram 5-106
malloc_1553 5-107
put_ram32 5-109
set_interrupt_level 5-110
write_to_address 5-111

This page intentionally left blank.

close_1553 Device Management Routines

Close access to a 1553 device

Syntax

device_management.close_1553 (device : IN OUT device_handle);

Description

close_1553() breaks the logical connection between the operating system and the 1553 device after it has been opened with the **open_1553()** or **init_1553_device()** call. Since only one connection to a single 1553 device can be active at a time, the 1553 device must be closed using **close_1553()** before reopening it.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

init_1553_device(), open_1553()

init_1553_device Device Management Routines

Open and initialize the 1553 device

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.init_1553_device( device_index  : IN  valid_device;
                                    queue_length  : IN  portable_types.uword;
                                    bsm_length    : IN  portable_types.uword )
                                    RETURN        device_handle;
```

Description

init_1553_device() initializes and opens one 1553 device for access by the application. This function calls a set of six library subprograms that provides the required initialization for proper firmware operation. If the board has not been previously initialized, the subprograms called are **open_1553()**, **load_firmware**, **start_firmware, init_1553_ memory()**, **init_interrupt_queue()**, and **create_bsm_buffer()**. If the other channel on the board has been initialized, the subprograms called are **open_1553()**, **initialize_channel()** and, optionally, **init_interrupt_queue** and **create_bsm_buffer**. This function or the subprograms listed above must be called for each 1553 board in the system before further initialization can proceed.

Parameters

The **device_index** parameter indicates which 1553 device to open. It is used as an index by the parser utility when the file *1553dev.cfg* is read. For systems with only one board, the value should always be equal to one.

The **queue_length** parameter contains the number of interrupt queue entries desired for each interrupt queue. Two interrupt queues, for double buffering purposes, will be initialized. Interrupt queue initialization is required for proper firmware operation even though interrupts may not be used by the application program.

The **bsm_length** parameter contains the desired length in words for each bus sequential monitor buffer. Two buffers, for double buffering purposes, will be initialized. Bus

sequential monitor initialization is required for proper firmware operation even though it may not be used by the application program.

Return Value

The **device_handle** parameter is an access type referring to a record of type **device_record**. This record contains device specific information that is used by other library calls. The components of this record are listed in the description of **open_1553()**.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
FORMAT_ERROR	A format error was detected by the parser while reading the device configuration file.
INVALID_KEYWORD	An invalid keyword was detected by the parser while reading the device configuration file.
INVALID_VALUE	An invalid value was detected by the parser while reading the device configuration file.
OPEN_FAIL	Opening the 1553 device failed. The procedure **low_level.open_device()** raises this exception.
OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.
FILE_OPEN_ERROR	The file fw_config_file could not be opened.
LOAD_FIRMWARE_FAIL	The firmware could not be loaded properly.

See Also

close_1553(), create_bsm_buffer(), init_interrupt_queue(), init_1553_memory(), open_1553(), start_firmware()

initialize_channel Device Management Routines

Initializes internal registers applicable to the channel

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.initialize_channel( device :  IN   device_handle  );
```

Description

initialize_channel() causes the ABI channel to initialize internal registers according to the current memory allocation.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
CHANNEL_INIT_ERROR	The channel initialization failed.
CHANNEL_TIMEOUT_ERROR	A timeout occurred during channel initialization.
CHANNEL_ALREADY_INITIALIZED	The channel is already initialized.

See Also

init_1553_device()

issue_host_cmd Device Management Routines

Initiates host command processing

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.issue_host_cmd( device       : IN  device_handle;
                                  host_command : IN  valid_host_cmd;
                                  data_word    : IN  valid_data_word  );
```

Description

issue_host_cmd() writes the host data word to offset 1 from the control base address (if applicable), and then writes the host command to the control base address.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **host_command** parameter specifies the number of the host command to be processed.

The **data_word** parameter contains the data word associated with the host command.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
HOST_CMD_TIMEOUT	A timeout occurred during host command processing
HOST_CMD_ERROR	A host command error occurred

load_firmware Device Management Routines

Download the V5 firmware files.

Syntax

```
WITH    abi_types,
        MILSTD1553_types,
        portable_types;
USE     abi_types,
        MILSTD1553_types,
        portable_types;

device_management.load_firmware( device   :    IN    device_handle );
```

Description

load_firmware() downloads the firmware from an instruction ram file and a data ram file. A call to this procedure is not necessary if **init_1553_device()** is used for 1553 device initialization.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
LOAD_FIRMWARE_FAIL	Unable to load the firmware.

See Also

perform_bit(), init_1553_device(), open_1553()

open_1553 Device Management Routines

Open the 1553 device and attach a shared memory segment to its local memory

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.open_1553 (  device_index  :  IN  valid_device )
                               RETURN          device_handle;
```

Description

open_1553() opens one 1553 device for access by the application. This function is required for proper firmware initialization and must be called before further 1553 device initialization can proceed. A call to this function is not necessary if **init_1553_device()** is used for 1553 device initialization. A write/read test is performed to verify that the board is present.

Parameters

The **device_index** parameter indicates which 1553 device to open. It is used as an index by the parser utility when the file *1553dev.cfg* is read. For systems with only one board, the value should always be equal to one.

Return Value

The **device_handle** parameter is an access type referring to a record of type **device_record**. This record contains device specific information that is used by other library calls. The components of this record are:

```
TYPE device_record IS
  RECORD
    is_open           : boolean := FALSE;
    device_number     : uword RANGE 1..max_dev;
    channel_number    : valid_channel_number;
    control_base_addr : long;
    abi_number        : valid_abi_number;
```

```
    product_name      : product_type;
    device_name       : string(1..80);
    extra_name        : string(1..80);
    file_descriptor   : file_descriptor_ptr;
    base_address      : system.address;
    physical_address  : system.address;
    io_base           : uword;
    io_vector         : uword;
    ram_size          : long;
    next_free_space   : long;
END RECORD;
```

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
DEVICE_PROBE_FAIL	The write/read probe test failed. The board was not accessed properly. The probable causes are: power is off or the device is addressed incorrectly.
FORMAT_ERROR	A format error was detected by the parser while reading the device configuration file.
INVALID_KEYWORD	An invalid keyword was detected by the parser while reading the device configuration file.
INVALID_VALUE	An invalid value was detected by the parser while reading the device configuration file.
OPEN_FAIL	Opening the 1553 device failed. The procedure **low_level.open_device()** raises this exception.

See Also

close_1553(), init_1553_device()

read_time — Device Management Routines

Obtain the current time from the internal timer registers

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.read_time( device : IN device_handle );
                             RETURN int_timer;
```

Description

read_time() accesses the 1553 device internal timer registers and returns the value.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Return Value

The return value is of type **int_timer**, defined as follows for the standard time format:

```
TYPE int_timer IS
RECORD
     high 16 :  uword;
     mid 16  :  uword;
     low 16  :  uword;
END RECORD;
```

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

init_1553_device(), open_1553()

reset_channel Device Management Routines

Reopen and reinitialize a channel of the 1553 device

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.reset_channel( device   : IN   OUT    device_handle   );
```

Description

reset_channel() reinitializes one channel of the 1553 bus device for operation by calling all required initialization subprograms. The subprograms are **open_1553()**, **init_interrupt_queue()**, **create_bsm_buffer()**, **start_firmware()**, **start_application()**, and **initialize_channel()**.

NOTE: **open_1553()**, **init_interrupt_queue()**, and **create_bsm_buffer()** are only called if necessary. **start_firmware()** and **start_application()** are only called if the application has not been previously started.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
DEVICE_PROBE_FAIL	The write/read probe test failed. The board was not accessed properly. Probable causes are: power is off or the device is addressed incorrectly.
MEMORY_ALLOCATION	Allocation of heap space for the device record failed.

OPEN_FAIL	Opening the 1553 device failed. The procedure **low_level.open_1553()** raises this exception.
OUT_OF_MEMORY	Allocation of memory on the 1553 device has failed.
SHARED_MEMORY_FAIL	Attachment to mapped memory failed. The function **low_level.initialize_shared_memory()** raises this exception.
LOAD_ERROR	The RAM load failed.
APP_MODE_ERROR	The switch to application mode failed.
CHANNEL_INIT_ERROR	The channel initialization failed.

See Also

open_1553(), **init_interrupt_queue()**, **create_bsm_buffer()**, **start_firmware()**, **start_application()**, **initialize_channel()**

set_host_dma Device Management Routines

Writes the host DMA control information to the DMA Software Control Registers

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.set_host_dma( device      : IN device_handle;
                                dma_control : IN dma_control_record );
```

Description

set_host_dma() writes the DMA Control, ABI Address, VME Address, and DMA word count values to the Software Control registers at offsets 15, 16, 17, and 18, respectively.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **dma_control** parameter is a record used to initialize the DMA information.

See Also

init_1553_device(), open_1553()

start_firmware Device Management Routines

Start firmware execution

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.start_firmware (  device :   IN  device_handle );
```

Description

start_firmware() halts the operations of the firmware.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
INVALID_PROD_NUM	Invalid firmware version code.
START_FIRMWARE_FAIL	Failure starting the firmware.

See Also

init_1553_device(), open_1553()

start_io Device Management Routines

Start input and output operations on the 1553 bus

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.start_io  (   device :  IN   device_handle );
```

Description

start_io() enables the 1553 device to perform bus operations by issuing Host Command 2. This procedure should be called to start 1553 bus processing, usually after the bus controller, remote terminal, and bus monitor data structures have been defined.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
CHANNEL_NOT_INITIALIZED	The channel must be initialized by calling initialize_channel before 1553 processing can be started.
ENABLE_CHANNEL_ERROR	An enable channel host command error occurred.
ENABLE_CHANNEL_TIMEOUT	A timeout occurred during host command processing.

See Also

init_1553_device(), open_1553(), stop_io()

stop_io Device Management Routines

Stop input and output operations on the 1553 bus

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

device_management.stop_io ( device : IN device_handle );
```

Description

stop_io() disables the 1553 device by issuing Host Command 3.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
DISABLE_CHANNEL_TIMEOUT	A timeout occurred during host command processing.
DISABLE_CHANNEL_ERROR	A disable channel host command error occurred.

See Also

init_1553_device(), open_1553(), start_io()

stub_select — Device Management Routines

Change the stub selection

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;
device_management.stub_select ( device     : IN  device_handle;
                                stub_value : IN  valid_stub_value );
```

Description

stub_select() provides the capability to change the stub length between short stub (stub_value = nonzero) and long stub (stub_value = 0). Long stub is the default.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **stub_value** parameter selects long stub (0) or short stub (1).

define_rt_mon_sa Remote Terminal Routines

Allocate a transmit subaddress for a remote terminal mapped monitor

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.define_rt_mon_sa(  device  :  IN  OUT  device_handle;
                                 sa_info :  IN  OUT  sa_info_type  );
```

Description

define_rt_mon_sa() allocates a subaddress for a remote terminal mapped monitor. It defines the appropriate data structures and sets the filter table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info** parameter is a record containing information relative to the initialization of the subaddress. The components of this record are listed in the description of **define_rt_sa()**.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553()

define_rt_sa — Remote Terminal Routines

Define a remote terminal subaddress

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       rt_management;
USE    MILSTD1553_types;

rt_management.define_rt_sa ( device   : IN OUT  device_handle;
                             sa_info  : IN      abi_types.sa_info_type );
```

Description

define_rt_sa() defines and sets up a remote terminal subaddress. This procedure allocates and initializes the appropriate data structures and data buffers in the 1553 device's memory, initializes the status response word for the remote terminal, initializes the subaddress's Filter Table entry, and enables the remote terminal for emulation. Multiple data buffers can be allocated for a subaddress. The procedure **set_global_control()** can be used to enable mapped monitor mode.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info** parameter is a record containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of this record and their descriptions are:

```
TYPE sa_info_type IS
  RECORD
    rt            :  valid_rt;
    sa            :  valid_sa;
    t_r           :  boolean;
    filter_bit    :  boolean;
    interrupt_bit :  boolean;
    swap_bit      :  boolean;
    num_buffers   :  uword RANGE 1..max_rt_buffers;
    buff_id       :  sa_buffer_id := (others =>0);
  END RECORD;
```

The **rt** and **sa** components define the remote terminal and subaddress.

The **t_r** component specifies whether the subaddress is to receive or transmit data. The enumerated type **message_type** has literal values of **receive** and **transmit**.

The **filter_bit** specifies whether sequential monitoring is enable for the subaddress.

The **interrupt_bit** specifies whether to interrupt when a message is received.

The **swap_bit** specifies whether to swap upon receipt of an error free message.

The **num_buffers** component specifies the number of data buffers to allocate for the subaddress.

The **buff_id** component is an array initialized by the call that contains an address offset to each subaddress data buffer that is allocated. These offsets allow access to the subaddress data buffers by the application program and are used by **read_sa_buffer()** and **write_sa_buffer()**.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.
DEFINE_RT_ERROR	The "Update Single RT Status Response Table Entry" host command failed.
DEFINE_RT_TIMEOUT	A timeout occurred during host command processing.

See Also

init_1553_device(), open_1553(), read_sa_buffer(), **set_filter_word(), set_global_control()**, **write_sa_buffer()**

read_filter_word Remote Terminal Routines

Read the transmit or receive filter word for an RT

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.read_filter_word( device  :  IN  device_handle;
                                sa_info :  IN  sa_info_type  )
                                RETURN long;
```

Description

read_filter_word() reads the transmit (offset 2) or receive (offset 1) filter word for an RT from the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info** parameter is a record containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of this record are listed in the description of **define_rt_sa()**.

Return Value

This function returns the 32-bit value of the specified filter word.

See Also

define_rt_sa(), init_1553_device(), open_1553(), set_filter_word()

read_mc17_data Remote Terminal Routines

Read the data word associated with mode code 17

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.read_mc17_data (  device  :  IN   device_handle;
                                rt      :  IN   valid_rt )
                                RETURN  uword;
```

Description

read_mc17_data() returns the data word associated with the Last Synchronize with Data Word Mode Code, mode code number 17.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the Last Synchronized Word Table.

Return Value

The value returned is the data word associated with the Last Synchronize with Data Word Mode Code sent to the corresponding remote terminal.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

init_1553_device(), open_1553()

read_rt_int_word — Remote Terminal Routines

Read the transmit or receive interrupt word for an RT

Syntax

```
WITH    abi_types,
        MILSTD1553_types,
        portable_types;
USE     abi_types,
        MILSTD1553_types,
        portable_types;

rt_management.read_rt_int_word(  device   :  IN   device_handle;
                                 sa_info  :  IN   sa_info_type  )
                                 RETURN long;
```

Description

read_rt_int_word() reads the transmit (offset 6) or receive (offset 5) interrupt word for an RT from the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info** parameter is a record containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of this record are listed in the description of **define_rt_sa()**.

Return Value

This function returns the 32-bit value of the specified Interrupt on Message Received word.

See Also

define_rt_sa(), init_1553_device(), open_1553(), set_rt_int_word()

read_sa_buffer — Remote Terminal Routines

Read the data buffer of a subaddress

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.read_sa_buffer( device    : IN    device_handle;
                              buffer_id : IN    long;
                              in_buffer : OUT   data_buffer;
                              status    : OUT   uword );
```

Description

read_sa_buffer() reads the contents of a subaddress data buffer.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **buffer_id** parameter is an address of the subaddress data buffer. The procedure **define_rt_sa()** creates this offset and returns it to the caller as **sa_info.buffer_id**.

The **in_buffer** parameter is a data buffer allocated by the caller. The contents of the subaddress data buffer are copied to this buffer. The procedure uses the word count field of the subaddress data buffer's Flag Word to determine the number of words to copy.

The **status** parameter returns the number of words read or zero if the subaddress data buffer is stale. Each time this routine is called for a particular subaddress, it clears the Flag Word (word 2) of the data buffer. The firmware will set the Flag Word after it writes new data to the buffer. A failure is declared if the routine reads a cleared Flag Word; in which case, 32 data words are copied to the user buffer and **status** will equal zero. Normally, **status** will return the number of words copied to the caller's data buffer.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR Invalid parameter in the subprogram's parameter list.

See Also

define_rt_sa(), init_1553_device(), open_1553(), write_sa_buffer()

read_status_word — Remote Terminal Routines

Read the Status Response Word for the specified remote terminal

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.read_status_word (  device  :  IN  device_handle;
                                  rt      :  IN  valid_rt )
                                  RETURN  status_response_type;
```

Description

read_status_word() returns the Status Response Word from the RT Status Response Table for a remote terminal.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the RT Status Response Table.

Return Value

The **status_response_type** parameter is a record representing the Status Response Word that will be transmitted by the remote terminal. See **load_status_word()** for a listing of the components of this record.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR Invalid parameter in the subprogram's parameter list.

See Also

define_rt_sa(), init_1553_device(), set_status_word(), open_1553()

read_swap_word — Remote Terminal Routines

Read the transmit or receive sequential monitor swap word for an RT

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.read_swap_word( device    : IN   device_handle;
                              sa_info   : IN   sa_info_type )
                              RETURN long;
```

Description

read_swap_word() reads the transmit (offset 3) or receive (offset 4) sequential monitor swap word for an RT from the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info** parameter is a record containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of this record are listed in the description of **define_rt_sa()**.

Return Value

This function returns the 32-bit value of the specified Swap on Message word.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The RT specified in **sa_info** has not been defined.

See Also

define_rt_sa(), init_1553_device(), open_1553(), set_swap_word()

read_transmit_sa_buffer Remote Terminal Routines

Read the data buffer of a transmit subaddress using VMEbus long word access

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.read_transmit_sa_buffer(  device     :  IN   device_handle;
                                        buffer_id  :  IN   valid_buffer_id;
                                        in_buffer  :  OUT  data_buffer;
                                        status     :  OUT  uword  );
```

Description

read_transmit_sa_buffer() reads the contents of a transmit subaddress data buffer using VMEbus long word (32-bit) access.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **buffer_id** parameter is an address offset pointing to the subaddress data buffer. The procedure **define_rt_sa()** creates this offset and returns it to the caller as **sa_info.buffer_id**.

The **in_buffer** parameter is a data buffer allocated by the caller. The contents of the subaddress data buffer are copied to this buffer. The procedure uses the word count field of the subaddress data buffer's Flag Word to determine the number of words to copy.

The **status** parameter returns the number of words read or zero if the contents of the subaddress data buffer are copied to this buffer.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_sa_buffer(), write_sa_buffer()

rt_wrap — Remote Terminal Routines

Enable or disable an RT to "wrap"

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.rt_wrap(  device     :  IN       device_handle;
                        sa_info_r  :  IN       sa_info_type;
                        sa_info_t  :  IN  OUT  sa_info_type;
                        wrap       :  IN       boolean   );
```

Description

The **rt_wrap()** routine allows a BC to test the RT by sending a BC-RT message immediately followed by a RT-BC message. The RT should then send back the same data that the BC sent to it.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info_r** parameter is a pointer to the **SA_INFO** record for the receive subaddress of the RT for which to enable or disable the wrap function.

The **sa_info_t** parameter is a pointer to the **SA_INFO** record for the transmit subaddress of the RT for which to enable or disable the wrap function. The **SA_INFO** structures for **sa_info_r** and **sa_info_t** should specify the same RT and SA.

The **wrap** parameter determines if the call will enable (TRUE) or disable (FALSE) the wrap function. If **wrap** is **TRUE**, the **rt_wrap** routine will set the transmit buffer pointer of the RT/SA specified by **sa_info_t** to match the receive buffer pointer of the RT/SA specified by **sa_info_r**. If **wrap** is **FALSE**, the **rt_wrap** routine will set the transmit buffer pointer of the RT/SA specified by **sa_info_t** back to the transmit buffer which was originally allocated (given in the **sa_info_t->buffer_id[0]** field).

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
DEVICE_NOT_OPEN	The **device** parameter contains a NULL value.
UNDEFINED_RT	There is no RT definition table for the specified remote terminal.
UNDEFINED_SUBADDRESS	There is no control buffer for the specified subaddress.

See Also

init_1553_device(), define_rt_sa()

set_filter_word — Remote Terminal Routines

Writes a value to the transmit or receive filter word in the RT Definition Table

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_filter_word(  device      :  IN  OUT  device_handle;
                                sa_info     :  IN       sa_info_type;
                                filter_word :  IN       long  );
```

Description

set_filter_word() writes a value to the transmit (offset 1) or receive (offset 2) filter word in the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info** parameter is a record containing information relative the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the **sa_info** record are listed in the description of **define_rt_sa()**.

The **filter_word** parameter contains the value to be written to the filter word in the RT definition table.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_filter_word()

set_global_control — Remote Terminal Routines

Write a value to the Global Control word

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_global_control(  device  :  IN  OUT  device_handle;
                                   rt      :  IN       valid_rt;
                                   value   :  IN       global_control_type  );
```

Description

set_global_control() writes a value to the Global Control word for a specific remote terminal.

NOTE: **update_rt_stat_resp_entry** must be called for RT Status Response Table changes to take effect.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the Protocol Table.

The **value** parameter is a number representing an entry in the Protocol Table.

See Also

init_1553_device(), open_1553(), **update_rt_stat_resp_entry()**

set_mc16_data — Remote Terminal Routines

Initialize the data word for mode code 16

Syntax

```
WITH    abi_types,
        MILSTD1553_types,
        portable_types;
USE     abi_types,
        MILSTD1553_types,
        portable_types;

rt_management.set_mc16_data( device :  IN  device_handle;
                             rt     :  IN  valid_rt;
                             tvw    :  IN  uword  );
```

Description

set_mc16_data() initializes the data word for the Transmit Vector Word Mode Code, mode code number 16.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the Transmit Vector Word Table.

The **tvw** parameter contains the data word associated with the Transmit Vector Word Mode Code that will be transmitted from the corresponding remote terminal.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

init_1553_device(), open_1553()

set_mc19_data — Remote Terminal Routines

Initialize the data word for mode code 19

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_mc19_data(  device   :  IN  device_handle;
                              rt       :  IN  valid_rt;
                              bit_word :  IN  uword );
```

Description

set_mc19_data() initializes the data word for the Transmit BIT Word Mode Code, mode code number 19.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the BIT Word Table.

The **bit_word** parameter contains the data word associated with the Transmit BIT Word Mode Code that will be transmitted from the corresponding remote terminal.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist

See Also

init_1553_device(), open_1553()

set_mim — Remote Terminal Routines

Set the mode command interrupt mask

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_mim(  device     :  IN   device_handle;
                        rt         :  IN   valid_rt;
                        mode_code  :  IN   mode_code_type );
```

Description

set_mim() sets the mode command interrupt mask (offset 7) for the specified mode code. This mask is used to generate an interrupt whenever a specific mode command message is received.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the mode command interrupt mask.

The **mode_code** parameter is an enumerated type representing the mode code that will be masked. Valid values for **mode_code_type** and their MIL-STD-1553B functions are:

dyn_bus_cntrl	Dynamic Bus Control
synch	Synchronize
xmit_status_word	Transmit Status Word
i**nit_self_test**	Initiate Self-Test
xmittr_shutdown	Transmitter Shutdown
override_xmittr_shutdown	Override Transmitter Shutdown
inhibit_flag_bit	Inhibit Terminal Flag Bit
override_inhibit_flag_bit	Override Terminal Inhibit Flag Bit

reset_rt	Reset Remote Terminal
xmit_vector_word	Transmit Vector Word
synch_w_data	Synchronize with Data Word
xmit_last_command	Transmit Last Command
xmit_bit_word	Transmit Built-in-Test Word
select_xmittr_shutdown	Selected Transmitter Shutdown
override_select_xmittr_shutdown	Override Selected Transmitter Shutdown

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT is not allocated.

See Also

init_1553_device(), open_1553(), **set_filter_word()**

set_rt_dma — Remote Terminal Routines

Write DMA control information to an RT data control buffer

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_rt_dma(  device      :  IN   device_handle;
                           buffer_id   :  IN   valid_buffer_id;
                           dma_control :  IN   dma_control_ptr  );
```

Description

set_rt_dma() writes DMA control, ABI address, VME address, and DMA word count values in the RT Data Control Buffer at offsets 3, 4, 5, and 6, respectively.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **buffer_id** parameter contains the offset to the RT Data Control Buffer.

The **dma_control** parameter is a record that is used to initialize the DMA information.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_BUFFER	The specified RT Data Control Buffer does not exist.

See Also

init_1553_device(), open_1553()

set_rt_int_word — Remote Terminal Routines

Writes a value to the transmit or receive interrupt on message received word in the RT Definition Table

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_rt_int_word(  device    :  IN   device_handle;
                                sa_info   :  IN   sa_info_type;
                                int_word  :  IN   long   );
```

Description

set_filter_word() writes a value to the transmit (offset 5) or receive (offset 6) interrupt on message received word in the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info** parameter is a record containing information relative the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the record are listed in the description of **define_rt_sa()**.

The **int_word** parameter contains the value that is copied to the RT Definition Table.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_rt_int_word()

set_status_word — Remote Terminal Routines

Set the Status Response Word for the specified remote terminal

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_status_word( device      : IN  device_handle;
                               rt          : IN  valid_rt;
                               status_word : IN  status_response_type );
```

Description

set_status_word() sets the Status Response Word in the RT Status Response Table for a remote terminal. The procedure **define_rt_sa()** will also initialize the Status Response Word for the remote terminal whenever it is called.

NOTE: **update_rt_stat_resp_entry()** must be called for RT Status Response Table entry changes to take effect.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the RT Status Response Table.

The **status_word** parameter is a record representing the Status Response Word that will be transmitted by the remote terminal. The **rt_address** component will be set by the call. The components of this record are:

```
TYPE status_response_type IS
  RECORD
    terminal_flag              : boolean  := FALSE;
    bus_control_accept         : boolean  := FALSE;
    subsystem_flag             : boolean  := FALSE;
    busy                       : boolean  := FALSE;
```

```
  broadcast_command_received  : boolean  := FALSE;
  unused                      : unused_3 := 0;
  service_request             : boolean  := FALSE;
  instrumentation             : boolean  := FALSE;
  message_error               : boolean  := FALSE;
  rt_address                  : valid_rt;
END RECORD;
```

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553()

set_swap_word — Remote Terminal Routines

Writes a value to the transmit or receive sequential monitor swap word in the RT Definition Table

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_swap_word( device    : IN  device_handle;
                             sa_info   : IN  sa_info_type;
                             swap_word : IN  long );
```

Description

set_swap_word() writes a value to the transmit (offset 3) or receive (offset 4) sequential monitor swap word in the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa_info** parameter is a record containing information relative the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the **sa_info** record are listed in the description of **define_rt_sa()**.

The **swap_word** parameter contains the value that is copied to the RT Definition Table.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_swap_word()

set_trx_status_control Remote Terminal Routines

Write an entry to the RT Status Transmit Control word for an RT

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.set_trx_status_control(  device  :  IN  device_handle;
                                       rt      :  IN  valid_rt;
                                       value   :  IN  uword  );
```

Description

set_trx_status_control() writes a value to the RT Status Transmit Control word for an RT.

NOTE: **update_rt_stat_resp_entry** must be called for RT Status Response Table changes to take effect.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the RT Status Transmit Control word.

The **value** parameter contains the hexadecimal value that is written to the RT Status Transmit Control word.

See Also

update_rt_stat_resp_entry()

update_rt_stat_resp_entry Remote Terminal Routines

Updates the RT Status Response Table entry for an RT

Syntax

```
WITH    abi_types,
        MILSTD1553_types,
        portable_types;
USE     abi_types,
        MILSTD1553_types,
        portable_types;

rt_management.update_rt_stat_resp_entry(  device  :  IN  device_handle;
                                          rt      :  IN  valid_rt   );
```

Description

update_rt_stat_resp_entry() issues host command 7 to update the RT Response Table entry for an RT. This routine is required after a call to **set_status_word()**, **set_trx_status_control()**, or **set_global_control()**.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal address for the Status Response Table update.

See Also

set_status_word(), set_trx_status_control(), **set_global_control()**

write_sa_buffer — Remote Terminal Routines

Write data to a subaddress data buffer

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

rt_management.write_sa_buffer( device      : IN   device_handle;
                               word_count  : IN   valid_word_count;
                               buffer_id   : IN   long;
                               out_buffer  : IN   data_buffer;
                               status      : OUT  uword );
```

Description

write_sa_buffer() copies data to a subaddress data buffer.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **word_count** parameter defines the number of words to write to the subaddress data buffer.

The **buffer_id** parameter is an address offset pointing to the subaddress data buffer. The procedure **define_rt_sa()** creates this offset and returns it to the caller as **sa_info.buffer_id**.

The **out_buffer** parameter is a data buffer allocated by the caller. The contents of this buffer are copied to the subaddress data buffer.

The **status** parameter returns the number of words written or zero if the subaddress data buffer is stale. Each time this routine is called for a particular subaddress, it clears the Flag Word (word 2) of the data buffer. The firmware will set the Flag Word after it transmits

data from the buffer. A failure is declared if the routine reads a cleared Flag Word; in which case, data is copied from the user buffer and **status** will equal zero. Normally, **status** will return the number of words copied from the caller's data buffer.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_sa_buffer()

add_link_to_chain — Bus Controller Routines

Add a link to a bus controller chain program

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.add_link_to_chain (  device  :  IN OUT  device_handle;
                                   link    :  IN OUT  link_handle );
```

Description

add_link_to_chain() is used to build a bus controller program (also called a chain). A chain consists of multiple bus controller command blocks (also called links), each representing one bus controller command. An application can define any number of chain programs. The number of links in a chain is limited by the amount of memory available on the 1553 device or by the application. One link will be added to a chain each time this routine is called.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **link** parameter is an access type referring to a record of type **chain_link_info_type.** This record defines the bus controller command to add to the chain program. The type **chain_link_info_type** is a variant record depending upon the type of bus command, i.e., mode code, bc-rt, rt-rt, etc. The components of this record and their descriptions are:

```
TYPE chain_link_info_type (bc_command : bc_command_type) IS
  RECORD
    chain_id          :  valid_chain_id;
    rt1               :  valid_rt;
    sa1               :  valid_sa;
    word_count        :  valid_DATA_FIELD;   -- Unused when bc_command = mode
    control           :  bc_control_word;
    link_id           :  valid_link_id;
    loop_flag         :  boolean;
```

```
    error_injection   :  error_injection_word;
    buff_id           :  bc_buffer_id  : = (others =>0);
    num_buffers       :  uword;
    CASE bc_command IS
      WHEN mode    =>
        mode_code     :  mode_code_type;
      WHEN rtrt | rtrt_broadcast  =>
        rt2           :  valid_rt;
        sa2           :  valid_sa;
      WHEN bc_delay =>
        time_delay    :  valid_time_delay;
      WHEN OTHERS   =>
        null;
    END CASE;
  END RECORD;
```

The enumerated values of **bc_command** are:

```
TYPE bc_command_type IS
  ( mode,            --  Mode Code Command
    bcrt,            --  Bus Controller to Remote Terminal
    rtbc,            --  Remote Terminal to Bus Controller
    rtrt,            --  Remote Terminal to Remote Terminal
    bc_delay,        --  Timed Delay
    bcrt_broadcast,  --  BC to RT Broadcast Command
    rtrt_broadcast); --  RT to RT Broadcast Command
```

The **chain_id** component defines the chain program where the link will be added. If the chain is undefined, the link will become the head of a new chain.

The **rt1** and **sa1** components define the remote terminal and subaddress for the command.

The **word_count** component specifies the number of words to be transferred for the command.

The **control** component represents the BC control word found in word 14 of a chain program command. The control word is used to specify the response of the 1553 device during certain conditions. The components of this record are:

```
FOR bc_control_word USE
     RECORD
        continue_on_error      at 0 range 0..0;
        bc_retry               at 0 range 1..1;
        trigger_out            at 0 range 2..2;
        trigger_in             at 0 range 3..3;
        mframe_start           at 0 range 4..4;
        no_op                  at 0 range 5..5;
        capture_rtrt_data      at 0 range 6..6;
        bit_7                  at 0 range 7..7;
        int_on_error           at 0 range 8..8;
        int_on_blk_complete    at 0 range 9..9;
        int_on_bc_halt         at 0 range 10..10;
        int_on_retry_attempt   at 0 range 11..11;
        int_on_masked_status   at 0 range 12..12;
```

```
      bits_13_14             at 0 range 13..14;
      bus_b                  at 0 range 15..15;
      retry_count            at 0 range 16..20;
      retry_success          at 0 range 21..21;
      bits_22_30             at 0 range 22..30;
      retry_bus_on_success   at 0 range 31..31;
   END RECORD;
```

The **link_id** component is a unique link identifier that is initialized by the call. It is used by **read_link_data()**, **read_link_status()**, and **write_link_data()** procedures to refer to this link.

The **loop_flag** component specifies whether or not the link is to loop back to the head of the chain after execution or terminate. A value of **TRUE** will cause the chain program to loop back to the top and execute continuously. A value of **FALSE** will stop the chain program after the last block is executed. The **loop_flag** is only valid for the last link in a chain.

The **error_injection** component is a union of a structure and a word. The structure represents the error injection word found at offset 17 from the beginning of a chain program command. The error word is used to specify the errors defined for the 1553 BC command. The components of this record are:

```
TYPE error_injection_word IS
     RECORD
        bit_16           : unused_1;
        timeout          : boolean := false;
        word_count       : boolean := false;
        illegal_command  : boolean := false;
        no_gap           : boolean := false;
        bit_21           : unused_1;
        long_message     : boolean := false;
        short_message    : boolean := false;
        unexpected_gap   : boolean := false;
        bit_25           : boolean := false;
        manchester_error : boolean := false;
        too_few_bits     : boolean := false;
        too_many_bits    : boolean := false;
        parity_error     : boolean := false;
        sync_error       : boolean := false;
        bit_31           : boolean := false;
     END RECORD;
```

The **buff_id** component contains the buffer address for each buffer that is defined.

The **num_buffers** component specifies the number of data buffers to allocate for the link.

The **rt2** and **sa2** components define the remote terminal and subaddress for the second command word for rt-rt transfers.

The **time_delay** component defines a delay interval in 100 nano-second counts. The component is valid when **bc_command** is equal to **bc_delay**.

When **bc_command** is **mode**, the **mode_code** component defines the mode code and the **word_count** component becomes invalid. When **bc_command** is **rtrt**, the **rt1** and **sa1** components define the destination remote terminal and subaddress, and the **rt2** and **sa2** components define the source remote terminal and subaddress for the command. When **bc_command** is **bc_delay**, the **time_delay** component defines a delay interval in 100 nano-second counts.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
MODE_CODE_INIT_FAIL	Improper mode code subaddress. The link is still added to the chain; however, the subaddress will equal zero.
OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.
UNDEFINED_MODE_CODE	Invalid 1553B mode code. The link is still added to the chain.

See Also

init_1553_device(), open_1553(), read_link_data(), read_link_status(), write_link_data()

delink_chain — Bus Controller Routines

Uncombine a chain program to form two chains

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.delink_chain( device   : IN   device_handle;
                            chain_1  : IN   valid_chain_id
                            chain_2  : IN   valid_chain_id  );
```

Description

delink_chain() uncombines a chain program to form two chains. This procedure would be used for chains that have been previously linked together using the **link_chain()** call.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **chain_1** parameter contains an identifier that refers to the first chain program. The chain is delinked by clearing the tail pointer of its last command block. Thus, a chain that was initialized to loop for continuous execution can be delinked to execute only once on its next cycle.

The **chain_2** parameter contains an identifier that refers to the second chain program. The chain is delinked by clearing the tail pointer of its last command block. Thus, a chain that was initialized to loop for continuous execution can be delinked to execute only once on its next cycle.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

UNDEFINED_CHAIN The **chain_id** parameter is invalid.

See Also

init_1553_device(), link_chain(), open_1553()

disable_frame_timer Bus Controller Routines

Disables the minor frame timer

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.disable_frame_timer( device : IN device_handle );
```

Description

disable_frame_timer() is used to disable the minor frame timer by clearing the Enable Timer bit in the BC Frame Timer Register (offset 0E from the channel control base address).

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

See Also

enable_frame_timer()

enable_frame_timer Bus Controller Routines

Enables the BC Minor Frame Timer and sets the frame interval

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.enable_frame_timer( device : IN device_handle;
                                  time   : IN long );
```

Description

enable_frame_timer() is used to enable the minor frame timer (by setting the Enable Timer bit) and set the frame interval in the BC Frame Timer Register (offset E from the channel control base address).

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **time** parameter contains the time to use for the frame interval.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
MAX_TIME_EXCEEDED	The frame interval is limited to 23 bits.

See Also

disable_frame_timer()

link_chain — Bus Controller Routines

Combine two chain programs to form one

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.link_chain(  device   :  IN   device_handle;
                           chain_1  :  IN   valid_chain_id;
                           chain_2  :  IN   valid_chain_id  );
```

Description

link_chain() combines two chain programs to form one larger chain. The chains can later be uncombined using the procedure **delink_chain()**.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **chain_1** parameter contains an identifier that refers to the first chain program. This chain is linked to the second by setting the tail pointer of its last command block to point to the first command block of the second chain.

The **chain_2** parameter contains an identifier that refers to the second chain program. This chain is linked after the first chain and will execute after the last command block of the first chain has completed. A chain that was initialized to execute once can be linked to execute continuously by setting **chain_1** and **chain_2** to refer to the same chain program.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_CHAIN	A chain id parameter is invalid.

See Also

delink_chain(), init_1553_device(), open_1553()

load_chain Bus Controller Routines

Prepare a chain program for execution

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.load_chain(  device    :  IN   device_handle;
                           chain_id  :  IN   valid_chain_id  );
```

Description

load_chain() prepares a chain program for execution by issuing host command 5, Start BC Chain. A chain must have been previously constructed using one or more **add_link_to_chain()** calls. The chain will execute immediately if bus processing has been started (i.e., RESP, offset 3 from the control base address, is constantly changing). Otherwise, a call to **start_io()** will execute the chain if it has been loaded.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **chain_id** parameter contains an identifier that refers to a specific chain.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_CHAIN	The **chain_id** parameter is invalid.
LOAD_CHAIN_ERROR	A "load chain" host command error occurred.

LOAD_CHAIN_TIMEOUT A timeout occurred during host command processing.

See Also

add_link_to_chain(), init_1553_device(), open_1553(), start_io()

read_bc_data_buff_control Bus Controller Routines

Read the buffer pointer/control word for a specified BC buffer

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.read_bc_data_buff_control( device  : IN  device_handle;
                                         buff_id : IN  valid_buffer_id );
```

Description

read_bc_data_buff_control() reads the pointer/control word associated with the specified BC data buffer. The primary use of this function is to check bit 31; when set, this bit indicates that an error occurred in the message and that the data in the BC receive buffer may be incomplete. This function can also be used to check the status of the control bits for interrupt on buffer processed, enable database swap, and DMA enable. See Figure BC-7 of the ABI-V5 Reference Manual for details on these bits.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **buff_id** parameter is an offset to a BC data buffer. The function **add_link_to_chain()** generates this offset and assigns the value to **chain_link_info.buff_id[n]**.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

set_bc_data_buff_cntl(), add_link_to_chain()

read_link_data — Bus Controller Routines

Read the data buffer of a chain link

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.read_link_data( device    : IN   device_handle;
                              link_id   : IN   valid_link_id;
                              buff_id   : IN   valid_buffer_id;
                              in_buffer : OUT  data_buffer  );
```

Description

read_link_data() reads the data buffer of a previously added link in a chain program. This call may be used to read the data word associated with a mode code.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** procedure.

The **buff_id** parameter is an offset to the BC control block. The procedure **add_link_to_chain()** creates this offset and returns it to the caller as **chain_link_info.buff(n)**.

The **in_buffer** parameter is a data buffer allocated by the caller. The contents of the chain link's data buffer are copied to this buffer. The number of words copied to the buffer is determined from the word count field in the chain link's command word.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR Invalid parameter in the subprogram's parameter list.

See Also

add_link_to_chain(), init_1553_device(), open_1553(), write_link_data()

read_link_status — Bus Controller Routines

Read the status response words associated with a chain link

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.read_link_status ( device   :  IN    device_handle;
                                 link_id  :  IN    valid_link_id;
                                 status   :  OUT   status_block_type  );
```

Description

read_link_status() reads the status response words of the specified chain link. This call will return two status response words for rt-rt chain links; otherwise, it will return only one.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** procedure.

The **status** parameter is a record that contains one or more status response words from the previous transfer. The number of status response words updated by the call depends upon the type of transfer. The components of this record are:

```
TYPE status_block_type IS
  RECORD
    word_1  :  status_response_type;
    word_2  :  status_response_type;
  END RECORD;
```

See **load_status_word()** for a description of **status_response_type**.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR Invalid parameter in the subprogram's parameter list.

See Also

add_link_to_chain(), init_1553_device(), open_1553(), set_status_word_mask()

set_bc_control — Bus Controller Routines

Set the control word for a specified chain link

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.set_bc_control(  device    :  IN   device_handle;
                               link_id   :  IN   valid_link_id;
                               controls  :  IN   bc_control_word  );
```

Description

set_bc_control() writes a value to the control word (offset 14) of a BC Control Block.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** procedure.

The **controls** parameter is a record representing the control word and is used to specify the response of the 1553 device during certain conditions. See **add_link_to_chain()** for a listing of the components within this record.

See Also

add_link_to_chain(), init_1553_device(), open_1553()

set_bc_data_buff_control Bus Controller Routines

Set or clear bits 28, 29, and 30 of the specified BC data buffer pointer/control word

Syntax

```
WITH  abi_types,
      MILSTD1553_types,
      portable_types;
USE   abi_types,
      MILSTD1553_types,
      portable_types;

bc_management.set_bc_data_buff_control(  device  :  IN device_handle;
                                         link_id :  IN valid_link_id;
                                         buff_id :  IN valid_buffer_id;
                                         mask    :  IN data_buff_ptr_control  );
```

Description

set_bc_data_buff_control() sets or clears bits 28, 29, and 30 of the specified BC data buffer pointer/control word. Bits 28, 29, and 30 of the **mask** word are the values assigned. If the specified buffer is the currently active buffer for the associated BC control block (specified by **link_id**), bits 28, 29, and 30 of the current data buffer pointer/control (offset 19) in the BC control block are also set to the values specified by the **mask** word.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **link_id** parameter contains the unique identifier that refers to a specific link. This value is assigned by the **add_link_to_chain()** routine and is stored in **chain_link_info.link_id**.

The **buff_id** parameter is an address offset pointing to a BC data buffer. This value is assigned by the **add_link_to_chain()** routine and is stored in **chain_link_info.buff_id[n]**.

The **mask** parameter specifies which of the data control bits to set.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_CHAIN_DATA_POINTER	The data buffer specified by buff_id was not found.
UNDEFINED_NEXT_POINTER	The next pointer at the address specified by **buff_id** was a NULL pointer.

See Also

read_bc_data_buff_control(), add_link_to_chain()

set_bc_dma — Bus Controller Routines

Writes DMA control information to a BC Control Block

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.set_bc_dma(  device       : IN   device_handle;
                           block_id     : IN   valid_link_id;
                           dma_control  : IN   dma_control_ptr  );
```

Description

set_bc_dma() writes DMA Control, ABI Address, VME Address, and DMA word count values in the BC Control Block at offsets 4, 5, 6, and 7, respectively.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **block_id** parameter contains the offset to the BC Control Block.

The **dma_control** parameter is a pointer to a record used to initialize the DMA information.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_BLOCK	The specified BC Control Block does not exist

See Also

init_1553_device(), open_1553()

set_error_word — Bus Controller Routines

Set the error injection control word for a specified link

Syntax

```
WITH    abi_types,
        MILSTD1553_types,
        portable_types;
USE     abi_types,
        MILSTD1553_types,
        portable_types;

bc_management.set_error_word(  device   :  IN   device_handle;
                               link_id  :  IN   valid_link_id;
                               errors   :  IN   error_injection_word  );
```

Description

set_error_word() writes a value to the error injection control (offset 17) of a chain link.

Parameters

The **device** parameter is an access type referring to a record of type **device_record**. It is returned from the **open_1553()** or **init_1553_device()** function.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** procedure.

The **errors** parameter is a record representing the error injection control word and is used to inject 1553 errors. See **add_link_to_chain()** for a listing of the components of the record.

See Also

add_link_to_chain(), init_1553_device(), open_1553()

stop_chain — Bus Controller Routines

Halts the execution of a bus controller chain program

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.stop_chain (   device :  IN  device_handle );
```

Description

stop_chain() halts the execution of a bus controller chain program by issuing host command 4, Stop BC Chain.

Parameters

The **device** parameter contains the **device_handle** access type returned from **open_1553()** or **init_1553_device()**.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

init_1553_device(), load_chain(), open_1553()

update_bc_control_info Bus Controller Routines

Informs the firmware of global BC changes

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.update_bc_control_info( device : IN device_handle );
```

Description

update_bc_control_info() informs the firmware of changes to the BC intermessage gap value (BCIMGP), BC retry information (BCRETRY), maximum allowable status response time (RSPGPA), and Frame Timer (FRMTMR) registers by issuing host command 6, Update Global BC Control Information.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UPDATE_BC_ERROR	A host command error occurred during processing of host command 6, Update BC Global Control Information
UPDATE_BC_TIMEOUT	A timeout occurred during processing of host command 6, Update BC Global Control Information

See Also

enable_frame_timer(), disable_frame_timer()

write_link_data — Bus Controller Routines

Write data to a chain link data buffer

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bc_management.write_link_data (  device     :  IN   device_handle;
                                 link_id    :  IN   valid_link_id;
                                 buff_id    :  IN   valid_buffer_id;
                                 out_buffer :  IN   data_buffer  );
```

Description

write_link_data() copies user data to a chain link's data buffer. This call may be used to update the data word associated with a mode code.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** procedure.

The **buff_id** parameter is an offset to the BC control block. The procedure **add_link_to_chain()** creates this offset and returns it to the caller as **chain_link_info.buff(n)**.

The **out_buffer** parameter is a data buffer allocated by the caller. This routine will copy all valid data found in this buffer to the chain link's data buffer. The number of words copied to the buffer is determined from the word count field of the chain link's command word.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

add_link_to_chain(), init_1553_device(), open_1553(), read_link_data()

create_bsm_buffer — Bus Monitor Routines

Create a sequential monitor buffer

Syntax

```
WITH  abi_types,
      MILSTD1553_types,
      portable_types;
USE   abi_types,
      MILSTD1553_types,
      portable_types;

bm_management.create_bsm_buffer(  device      :  IN  OUT  device_handle;
                                  buffer_size :  IN       long  );
```

Description

create_bsm_buffer() allocates memory on the 1553 device for a sequential monitor buffer. This procedure is required for proper firmware initialization. A call to this procedure is not necessary if **init_1553_device()** is used for 1553 device initialization.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **buffer_size** parameter contains the length in words of a sequential monitor buffer. Two buffers of this length are created. When one becomes full, the buffers swap, the monitor buffer swap interrupt is generated, and the second buffer begins to fill. The size of the sequential monitor buffer is limited only by the amount of memory available on the 1553 device.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.

See Also

force_buffer_swap(), init_1553_device(), open_1553()

force_buffer_swap — Bus Monitor Routines

Swap the sequential monitor buffers

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bm_management.force_buffer_swap(  device          :  IN  device_handle;
                                  interrupt_flag  :  IN  boolean : = FALSE  );
```

Description

force_buffer_swap() swaps the two sequential monitor buffers. This routine can be used to flush the current sequential monitor buffer and start capturing data in the second or read the current sequential monitor buffer without waiting for it to fill.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **interrupt_flag** parameter determines whether or not to generate an interrupt when the swaps occurs. A value of **TRUE** will generate an interrupt.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

create_bsm_buffer(), init_1553_device(), open_1553()

read_bsm_buffer — Bus Monitor Routines

Read the sequential monitor buffer

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bm_management.read_bsm_buffer(  device      :  IN   device_handle;
                                new_buffer  :  OUT  monitor_buffer  );
```

Description

read_bsm_buffer() reads the contents of a sequential monitor buffer. The buffer not being filled by the firmware is the one read by the procedure.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **new_buffer** parameter is a data buffer allocated by the caller. The entire contents of the sequential monitor buffer are copied to this buffer.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

init_1553_device(), open_1553()

seq_mon_all_clear — Bus Monitor Routines

Disables sequential monitoring for all subaddresses for all RTs.

Syntax

```
WITH    abi_types,
        MILSTD1553_types,
        portable_types;
USE     abi_types,
        MILSTD1553_types,
        portable_types;

bm_management.seq_mon_all_clear(  device  :  IN   device_handle  );
```

Description

seq_mon_all_clear() disables sequential monitoring for all subaddresses for all remote terminals by clearing bit 1 of the MONCTRL register and all bits in both SA SM Filter words in the RT Definition Tables of all defined RTs.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

See Also

init_1553_device(), open_1553()

seq_mon_all_rt_sa — Bus Monitor Routines

Enables sequential monitoring for all remote terminals and subaddresses

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bm_management.seq_mon_all_rt_sa(  device  :  IN  device_handle  );
```

Description

seq_mon_all_rt_sa() enables sequential monitoring for all remote terminals and subaddresses by setting bit 1 of the MONCTRL register.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

See Also

init_1553_device(), open_1553()

seq_mon_all_sa — Bus Monitor Routines

Enables sequential monitoring for all subaddresses for an RT

Syntax

```
WITH    abi_types,
        MILSTD1553_types,
        portable_types;
USE     abi_types,
        MILSTD1553_types,
        portable_types;

bm_management.seq_mon_all_sa(  device  :  IN  device_handle;
                               rt      :  IN  valid_rt  );
```

Description

seq_mon_all_sa() enables sequential monitoring for all subaddresses for a remote terminal by setting all bits in the SA SM Filter word in the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter contains the remote terminal number.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
OUT_OF_MEMORY	There is not enough memory to allocate the RT record.

See Also

init_1553_device(), open_1553()

seq_mon_clear_all_sa — Bus Monitor Routines

Disables sequential monitoring for all subaddresses of an RT

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bm_management.seq_mon_clear_all_sa( device : IN device_handle;
                                    rt     : IN valid_rt   );
```

Description

seq_mon_clear_all_sa() disables sequential monitoring for all subaddresses of a remote terminal by clearing all bits in the SA SM Filter word in the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **rt** parameter specifies on which RT to disable sequential monitoring.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

init_1553_device(), open_1553()

seq_mon_clear_one_sa — Bus Monitor Routines

Disables sequential monitoring for one subaddress of a remote terminal

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bm_management.seq_mon_clear_one_sa( device : IN device_handle;
                                    sa     : IN sa_info_type );
```

Description

seq_mon_clear_one_sa() disables sequential monitoring for one subaddress of a remote terminal by clearing the corresponding bit in the SA SM Filter word in the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa** parameter is a record containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the record are listed in the description of **define_rt_sa()**.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_RT	The specified RT does not exist.

See Also

init_1553_device(), open_1553()

seq_mon_one_sa — Bus Monitor Routines

Enables sequential monitoring for one subaddress of a remote terminal

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bm_management.seq_mon_one_sa(  device :  IN  device_handle;
                               sa     :  IN  sa_info_type  );
```

Description

seq_mon_one_sa() enables sequential monitoring for one subaddress of a remote terminal by setting the corresponding bit in the SA SM Filter word in the RT Definition Table.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sa** parameter is a record containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the record are listed in the description of **define_rt_sa()**.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
OUT_OF_MEMORY	There is not enough memory to allocate the RT record.

See Also

init_1553_device(), open_1553()

set_bsm_dma — Bus Monitor Routines

Writes DMA control information to a sequential monitor buffer

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bm_management.set_bsm_dma(  device      :  IN  device_handle;
                            block_id    :  IN  valid_buffer_id;
                            dma_control :  IN  dma_control_ptr  );
```

Description

set_bsm_dma() writes DMA Control, ABI Address, VME Address, and DMA word count values in the sequential monitor buffer at offsets 7, 8, 9, and 10, respectively.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **block_id** parameter contains the offset to the sequential monitor buffer.

The **dma_control** parameter is a record used to initialize the DMA information.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
UNDEFINED_BUFFER	The specified sequential monitor buffer does not exist

See Also

init_1553_device(), open_1553()

set_int_on_swap — Bus Monitor Routines

Enables and disables Interrupt on Buffer Swap

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bm_management.set_int_on_swap( device          : IN  device_handle;
                               sqm_buffer_id   : IN  valid_buffer_id;
                               interrupt_flag  : IN  boolean   );
```

Description

set_int_on_swap() sets or clears the Interrupt on Buffer Swap bit in the Buffer Control Length word of the sequential monitor buffer.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **sqm_buffer_id** parameter contains the offset to the sequential monitor buffer.

The **interrupt_flag** parameter is "0" to clear Interrupt on Buffer Swap and "1" to set the bit.

See Also

create_bsm_buffer()

disable_1553_interrupts Interrupt Management Routines

Disable 1553 device interrupts to the host

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

interrupt_management.disable_1553_interrupts( device : IN device_handle );
```

Description

disable_1553_interrupts() disables the 1553 device from interrupting the host. Interrupts are disabled by clearing ENA INT, bit 3, in the Control Status Register located at word offset 0.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

enable_1553_interrupts(), init_1553_device(), open_1553()

enable_1553_interrupts Interrupt Management Routines

Enable 1553 device interrupts to the host

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

interrupt_management.enable_1553_interrupts( device : IN device_handle );
```

Description

enable_1553_interrupts() enables interrupts from the 1553 device to the host. Interrupts are enabled by setting ENA INT, bit 3, in the Control Status Register located at word offset 0.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

disable_1553_interrupts(),init_1553_device(), open_1553()

handle_1553_interrupt Interrupt Management Routines

Handle interrupts generated by the 1553 device

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

interrupt_management.handle_1553_interrupt( device : IN device_handle );
```

Description

handle_1553_interrupt() services interrupts generated by the 1553 device. It first determines the cause of the interrupt and then branches to an application specific handler for that interrupt if one exists. Each interrupt is logged into an interrupt queue resident on the host that is used for test purposes. This routine is callable from an Interrupt Service Routine written specifically for the caller's operating system, or it may be called directly from the application program if polling is used instead of interrupts.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

init_1553_device(), open_1553()

init_interrupt_queue Interrupt Management Routines

Create and initialize the interrupt queues

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

interrupt_management.init_interrupt_queue( device       : IN OUT device_handle;
                                           queue_length : IN     long );
```

Description

init_interrupt_queue() creates a double buffered interrupt queue and allocates memory on the 1553 device.

This procedure is required for proper firmware initialization. A call to this procedure is not necessary if **init_1553_device()** is used for 1553 device initialization.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **queue_length** parameter contains the maximum number of interrupts entries that are to be stored in the interrupt queue. Two circular queues of **queue_length** are allocated so that if an interrupt is generated during the execution of the host's interrupt handler, the firmware can log the interrupt into the second interrupt queue.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.

See Also

init_1553_device(), handle_1553_interrupt(), open_1553()

perform_bit BIT Management Routines

Executes Built-in-test routines

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

bit_management.perform_bit(  device      :  IN OUT  device_handle )
                             bit_number  :  IN      test_number;
                             loops       :  IN      number_of_loops;
                             bit_status  :  OUT     bit_status_type  );
```

Description

perform_bit() executes the built-in-test routines. These consist of the following:

- **test_1()** 29K Sanity Test. This test performs a read/write test on the I/O bus and memory bus.
- **test_2()** I/O Port Test. This test performs a read/write test for the I/O bus.
- **test_3()** Data RAM Test. This test performs a complemented data test.
- **test_4()** 29K Burst Mode Test. This test performs 29K-specific tests for data bursts.
- **test_5()** System Timer Test. This test performs a verification of the 48-bit system timer, including clock accuracy, counter, interval timing, and local interrupt tests.
- **test_6()** DSP/1553B Test. This test performs a short test including a FIFO wrap test, an interrupt test for transmit data, and verification of status flags for the 29K and DSP. This test also performs a long test which includes the 1553B "walking ones" test in addition to the short test.

Note: The built-in-tests must be run from BIT mode. Therefore, if the ABI is in application mode, the firmware must be reloaded using **load_firmware** before the built-in-test can be performed.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **bit_numbe**r parameter specifies the test to perform. The components of this record are:

```
TYPE test_number IS
   (test_1, test_2, test_3, test_4, test_5, test_6, all_tests);
```

The **loops** parameter specifies the number of iterations of the test to perform.

The **bit_status** parameter specifies whether the test passed or failed.

Return Value

The **bit_result** parameter returns **PASS** or **FAIL**.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

init_1553_device(), open_1553()

close_1553 Low Level Routines

Close a device

Syntax

```
WITH   abi_types,
       MILSTD1553_types,
       portable_types;
USE    abi_types,
       MILSTD1553_types,
       portable_types;

low_level.close_1553( device : IN OUT device_handle );
```

Description

close_1553() disconnects the process ID from the 1553 device.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** function.

See Also

open_1553()

fetch_from_address Low Level Routines

Read a value from 1553 device memory

Syntax

```
WITH   abi_types,
       low_level,
       MILSTD1553_types,
       portable_types;
USE    MILSTD1553_types;

TYPE target IS PRIVATE;
low_level.fetch_from_address ( device   :  IN   device_handle;
                               offset   :  IN   portable_types.uword )
                               RETURN   target;
```

Description

fetch_from_address() returns a value from 1553 device memory. This function is generic so that the size of the data value to be returned; i.e., 16 bit or 32 bit; may be determined by the user. For example, to return a 16-bit data value use the following instantiation:

```
FUNCTION get_ram IS NEW
low_level.fetch_from_address(portable_types.uword);
```

Parameter

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **offset** parameter is a word offset to the location in memory that will be read.

Return Value

The generic **target** parameter contains the data value returned from a location in the 1553 device memory. An instance of the generic unit must be created for the desired type.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.

See Also

init_1553_device(), open_1553(), write_to_address()

get_ram32 Low Level Routines

Read a value from 1553 device memory using VMEbus long word access

Syntax

USE MILSTD1553_types;

low_level.get_ram32(device : IN device_handle;
offset : IN long;);
RETURN long;

Description

get_ram32() returns a long word value from 1553 device memory. It has been written as both a macro and a subroutine. The macro call is much more efficient than the subroutine version and should be used where possible. To use the subroutine call, declare the **NO_MACRO** directive.

The **get_ram32()** macro is declared in prototype.h as:

```
#define get_ram32(device, offset) \
((unsigned long)(*((unsigned long *)device->base_address + offset)))
```

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **offset** parameter is a longword offset to the location in the 1553 device's memory that will be read.

Return Value

A long word value of the 1553 device's memory located at the word offset is returned.

See Also

init_1553_device(), open_1553(), put_ram32()

initialize_shared_memory Low Level Routines

Sets a pointer for access to the board

Syntax

USE MILSTD1553_types;

low_level.initialize_shared_memory(device : IN device_handle);

Description

initialize_shared_memory() obtains a shared memory segment from the operating system and overlays it at the physical memory location at which the board resides.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** function.

See Also

init_1553_device()

load_ram Low Level Routines

Perform software download of the data RAM and instruction RAM files

Syntax

USE MILSTD1553_types;

low_level.load_ram(device : IN device_handle);

Description

load_ram() loads the data RAM and 29K instruction RAM areas of the ABI-V5 with data from the download files.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** function.

See Also

open_1553(), init_1553_device()

malloc_1553 Low Level Routines

Allocates space in the board RAM

Syntax

```
USE     MILSTD1553_types;

low_level.malloc_1553(  device        :  IN     device_handle;
                        size          :  IN     long;
                        block_offset  :  OUT    long   );
```

Description

malloc_1553() allocates a number of words in the board RAM space for a given device and returns the offset to the beginning of that space.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** function.

The **size** parameter specifies the number of words to allocate.

The **block_offset** parameter specifies the long word offset allocated to the RAM space.

Return Value

A long word offset to the allocated RAM space.

Diagnostics

Exceptions as a result of this function are:

CONSTRAINT_ERROR	Invalid parameter in the subprogram's parameter list.
MEMORY_ALLOCATION	The device number specified is higher than **MAX_DEV**.

OUT_OF_MEMORY	There is not enough free memory to allocate the requested size.

See Also

define_rt_sa(), init_interrupt_queue(), create_bsm_buffer(), add_link_to_chain()

put_ram32 Low Level Routines

Write a value to 1553 device memory using VMEbus long word access

Syntax

```
USE     MILSTD1553_types;

low_level.put_ram32(   device   :   IN   device_handle;
                       offset   :   IN   long;
                       value    :   IN   long   );
```

Description

put_ram32() writes a value to 1553 device memory.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **offset** parameter is a longword offset to the location in memory that will be written.

The **value** parameter is a longword to be written to the 1553 device's memory.

See Also

get_ram32(), init_1553_device(), open_1553()

set_interrupt_level Low Level Routines

Select the desired interrupt level

Syntax

```
WITH    abi_types,
        MILSTD1553_types,
        portable_types;
USE     abi_types,
        MILSTD1553_types,
        portable_types;

low_level.select_interrupt_level( device : IN device_handle );
```

Description

select_irq sets the IRQ level on the V5 board to match the "int_vector=xx" specification in the *1553dev.cfg* file. The function is called from the **open_device** routine in *low_lvl.c*.

Parameters

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** function.

See Also

open_1553()

write_to_address — Low Level Routines

Write a value to 1553 device memory

Syntax

```
WITH   abi_types,
       low_level,
       MILSTD1553_types,
       portable_types;
USE    MILSTD1553_types;

TYPE target IS PRIVATE;
low_level.write_to_address ( device  :  IN  device_handle;
                             offset  :  IN  portable_types.uword;
                             t       :  IN  target );
```

Description

write_to_address() writes a value to 1553 device memory. This function is generic so that the size of the data value to be written; i.e., 16 bit or 32 bit; may be determined by the user. For example, to write a 16-bit data value use the following instantiation:

```
PROCEDURE put_ram32 IS NEW low_level.write_to_address(portable_types.uword );
```

Parameter

The **device** parameter contains the **device_handle** access type returned from the **open_1553()** or **init_1553_device()** function.

The **offset** parameter is a word offset to the location in memory that will be written.

The generic **t** parameter contains the data value that will be written to a location in the 1553 device memory. An instance of the generic unit must be created for the desired type.

Diagnostics

Exceptions as a result of this procedure are:

CONSTRAINT_ERROR Invalid parameter in the subprogram's parameter list.

See Also

fetch_from_address(), init_1553_device(), open_1553()

C LIBRARY REFERENCE

SBS Technologies' MIL-STD-1553 C support library routines provide a variety of operations useful to the application program. This section contains the C library programming commands for the ABI-V5 MIL-STD-1553 interface board, including a detailed table of contents and reference pages for the V5 library.

Contents

ABI-V5 Table of Contents	6-i
ABI-V5 C Library Reference	6-1 through 6-101

This page intentionally left blank.

ABI-V5 C Library Reference

Table of Contents

Device Management Routines

close_1553 6-1
init_1553_device 6-2
init_1553_device2 6-4
initialize_channel 6-6
issue_host_cmd 6-7
load_firmware 6-8
open_1553 6-9
read_time 6-11
reset_channel 6-12
set_host_dma 6-14
start_firmware 6-15
start_io 6-16
stop_io 6-17
stub_select 6-18

Remote Terminal Routines

define_mon_rt_sa 6-19
define_rt_sa 6-20
read_filter_word 6-22
read_mc17_data 6-23
read_rt_int_word 6-24
read_sa_buffer 6-25
read_status_word 6-26
read_swap_word 6-27
read_transmit_sa_buffer 6-28
rt_wrap 6-29
set_filter_word 6-31
set_global_control 6-32
set_illegal_mode_code 6-33
set_illegal_sa 6-35
set_int_on_rt_processed 6-37
set_mc16_data 6-38
set_mc19_data 6-39
set_mim 6-40
set_rt_dma 6-42

set_rt_int_word 6-43
set_status_word 6-44
set_swap_word 6-45
set_trx_status_control 6-46
update_rt_stat_resp_entry 6-47
write_sa_buffer 6-48

Bus Controller Routines

add_link_to_chain 6-49
delink_chain 6-53
disable_frame_timer 6-55
enable_frame_timer 6-56
link_chain 6-57
load_chain 6-58
read_bc_data_buff_cntl 6-59
read_link_data 6-61
read_link_status 6-62
set_bc_control 6-63
set_bc_data_buff_cntl 6-64
set_bc_dma 6-66
set_error_word 6-67
stop_chain 6-68
update_bc_control_info 6-69
write_link_data 6-70

Bus Monitor Routines

create_bsm_buffer 6-71
force_buffer_swap 6-72
read_bsm_buffer 6-73
seq_mon_all_clear 6-74
seq_mon_all_rt_sa 6-75
seq_mon_all_sa 6-76
seq_mon_clear_all_sa 6-77
seq_mon_clear_one_sa 6-78
seq_mon_one_sa 6-79
set_bsm_dma 6-80
set_int_on_swap 6-81

Interrupt Management Routines

disable_1553_interrupts 6-82
enable_1553_interrupts 6-83
handle_1553_interrupt 6-84

init_interrupt_queue 6-85

BIT Management Routines

bit_1 6-86
bit_2 6-87
bit_3 6-88
bit_4 6-89
bit_5 6-90
bit_6 6-91
bit_all 6-92

Low Level Routines

close_device 6-93
get_ram 6-94
get_ram32 6-95
initialize_shared_memory 6-96
load_ram 6-97
malloc_1553 6-98
open_device 6-99
put_ram 6-100
put_ram32 6-101

This page intentionally left blank.

close_1553 Device Management Routines

Close access to a 1553 device

Syntax

#include "sbs_incl.h"

void close_1553 (device)
DEVICE_HANDLE device;

Description

close_1553() breaks the logical connection between the operating system and the 1553 device after it has been opened with the **open_1553()** or **init_1553_device()** call

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

See Also

init_1553_device(), open_1553()

init_1553_device Device Management Routines

Open and initialize the first channel of the 1553 device

Syntax

```
#include "sbs_incl.h"

DEVICE_HANDLE init_1553_device ( device_index, queue_length, bsm_length )
int             device_index;
unsigned int    queue_length;
unsigned int    bsm_length;
```

Description

init_1553_device() prepares the first channel of the 1553 device for access by the application. This function calls a set of library subprograms that provides the required initialization for proper firmware operation. The subprograms called are **open_1553()**, **load_firmware()**, **start_firmware()**, and **initialize_channel()**, and optionally **init_interrupt_queue()** and **create_bsm_buffer()**. They may be called separately if desired. This function (init_1553_device), init_1553_device2, or the four required subprograms must be called for the first channel on the board before 1553 processing can occur on the channel.

Parameters

The **device_index** parameter indicates which 1553 device to open. It is used as an index by the parser utility when the file *1553dev.cfg* is read. For embedded systems without access to the file *1553dev.cfg*, system information can be obtained from the *dev_conf.h* header file at compile time. For this case, the directive **NO_PARSE** must be declared so that **device_index** will be used as an index into an array of structures of type **DEVICE_CONFIG**. This type is defined and initialized in *dev_conf.h* and contains system dependent fields necessary to open the 1553 device and attach it to a shared memory segment if required by the operating system.

The **queue_length** parameter contains the number of interrupt queue entries desired for each interrupt queue. Two interrupt queues, for double buffering purposes, will be initialized if the queue length is nonzero. Interrupt queue initialization is only required if interrupts are to be used by the application program. If the queue length is zero, no interrupt queues will be allocated.

The **bsm_length** parameter contains the desired length in words for each bus sequential monitor buffer. Two buffers, for double buffering purposes, will be initialized if bsm_length is nonzero. If bsm_length is zero, no sequential monitor buffers will be allocated.

Return Value

The return value, of **DEVICE_HANDLE** type, is a pointer referring to a structure of type **DEVICE_RECORD**. This structure contains 1553 board specific information that is used by other library calls. The components of this structure are listed in the description of **open_1553()**.

Diagnostics

A **NULL** value is returned if the call fails and the global variable **error_exception** will contain one of the following exceptions:

DEVICE_PROBE_FAIL	The write/read probe test failed. The board was not accessed properly. The probable causes are: power is off or the device is addressed incorrectly.
MEMORY_ALLOCATION	Allocation of heap space for the device structure failed.
OPEN_FAIL	Opening the 1553 device failed. The function **open_device()** in *low_lvl.c* raises this exception.
OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.
SHARED_MEMORY_FAIL	Attachment to mapped memory failed. The function **initialize_shared_memory()** in *low_lvl.c* raises this exception..
LOAD_ERROR	The RAM load failed.
APP_MODE_ERROR	The switch to application mode failed.
CHANNEL_INIT_ERROR	The channel initialization failed.

See Also

close_1553(), create_bsm_buffer(), init_1553_device2(), init_interrupt_queue(), initialize_channel(), open_1553(), start_firmware()

init_1553_device2 Device Management Routines

Open and initialize the second channel of the 1553 device

Syntax

```
#include "sbs_incl.h"

DEVICE_HANDLE init_1553_device2 ( device_index, queue_length, bsm_length )
int             device_index;
unsigned int    queue_length;
unsigned int    bsm_length;
```

Description

init_1553_device2() prepares the second channel of the 1553 device for access by the application. This function calls a set of library subprograms that provides the required initialization for proper firmware operation. The subprograms called are **open_1553()** and **initialize_channel()**, and optionally **init_interrupt_queue()** and **create_bsm_buffer()**. They may be called separately if desired. This function (init_1553_device2), init_1553_device, or the two required subprograms must be called for the second channel on the board before 1553 processing can occur on the channel.

Parameters

The **device_index** parameter indicates which 1553 device to open. It is used as an index by the parser utility when the file *1553dev.cfg* is read. For systems with only one board, the value should always be equal to one or two. For embedded systems without access to the file *1553dev.cfg*, system information can be obtained from the *dev_conf.h* header file at compile time. For this case, the directive **NO_PARSE** must be declared so that **device_index** will be used as an index into an array of structures of type **DEVICE_CONFIG**. This type is defined and initialized in *dev_conf.h* and contains system dependent fields necessary to open the 1553 device and attach it to a shared memory segment if required by the operating system.

The **queue_length** parameter contains the number of interrupt queue entries desired for each interrupt queue. Two interrupt queues, for double buffering purposes, will be initialized if the queue length is nonzero. Interrupt queue initialization is only required if interrupts are to be used by the application program. If the queue length is zero, no interrupt queues will be allocated.

The **bsm_length** parameter contains the desired length in words for each bus sequential monitor buffer. Two buffers, for double buffering purposes, will be initialized if bsm_length is nonzero. If bsm_length is zero, no sequential monitor buffers will be allocated.

Return Value

The return value, of **DEVICE_HANDLE** type, is a pointer referring to a structure of type **DEVICE_RECORD**. This structure contains 1553 board specific information that is used by other library calls. The components of this structure are listed in the description of **open_1553()**.

Diagnostics

A **NULL** value is returned if the call fails and the global variable **error_exception** will contain one of the following exceptions:

DEVICE_PROBE_FAIL	The write/read probe test failed. The board was not accessed properly. The probable causes are: power is off or the device is addressed incorrectly.
MEMORY_ALLOCATION	Allocation of heap space for the device structure failed.
OPEN_FAIL	Opening the 1553 device failed. The function **open_device()** in *low_lvl.c* raises this exception.
OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.
SHARED_MEMORY_FAIL	Attachment to mapped memory failed. The function **initialize_shared_memory()** in *low_lvl.c* raises this exception.
CHANNEL_INIT_ERROR	The channel initialization failed.

See Also

close_1553(), create_bsm_buffer(), init_1553_device(), init_interrupt_queue(), initialize_channel(), open_1553()

initialize_channel Device Management Routines

Initializes internal registers applicable to the channel

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  initialize_channel ( device )
DEVICE_HANDLE     device;
```

Description

initialize_channel() causes the ABI channel to initialize internal registers according to the current memory allocation.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

A **NULL** value is returned if the call fails and the global variable **error_exception** will contain one of the following exceptions:

CHANNEL_INIT_ERROR	The channel initialization failed.
CHANNEL_TIMEOUT_ERROR	A timeout occurred during channel initialization.
CHANNEL_ALREADY_INITIALIZED	The channel is already initialized.

See Also

init_1553_device()

issue_host_cmd Device Management Routines

Initiates host command processing

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  issue_host_cmd ( device, host_command, data_wd )
DEVICE_HANDLE     device;
VALID_HOST_CMD    host_command;
VALID_DATA_WORD   data_wd;
```

Description

issue_host_cmd() writes the host data word to offset 1 from the control base address (if applicable), and then writes the host command to the control base address.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **host_command** parameter specifies the number of the host command to be processed.

The **data_wd** parameter contains the data word associated with the host command.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain one of the following:

HOST_CMD_TIMEOUT	A timeout occurred during host command processing
HOST_CMD_ERROR	A host command error occurred

load_firmware Device Management Routines

Loads data RAM and instruction RAM

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  load_firmware ( device )
DEVICE_HANDLE     device;
```

Description

load_firmware() loads the instruction RAM and data RAM download files to the respective ABI RAM areas; writes a "1" to the CSR to reset the hardware; writes a "2" to the CSR to start the firmware; and leaves the ABI in BIT mode.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

PUP_STATUS_TIMEOUT	Power-up Status Timeout occurred
PUP_RESULT_ERROR	Power-up Result register has errors

See Also

init_1553_device(), open_1553()

open_1553 Device Management Routines

Open one channel of the 1553 device and attach a shared memory segment to its local memory

Syntax

#include "sbs_incl.h"

DEVICE_HANDLE open_1553 (device_index)
int device_index;

Description

open_1553() opens one channel of the 1553 device for access by the application. This function is required for proper firmware initialization and must be called before further 1553 device initialization can proceed. A call to this function is not necessary if **init_1553_device()** is used to initialize the 1553 device. A write/read test is performed to verify that the board is present.

Parameters

The **device_index** parameter indicates which 1553 device to open. It is used as an index by the parser utility when the file *1553dev.cfg* is read. For systems with only one board, the value should always be equal to one or two. For embedded systems without access to the file *1553dev.cfg*, system information can be obtained from the *dev_conf.h* header file at compile time. For this case, the directive **NO_PARSE** must be declared so that **device_index** will be used as an index into an array of structures of type **DEVICE_CONFIG**. This type is defined and initialized in *dev_conf.h* and contains system dependent fields necessary to open the 1553 device and attach it to a shared memory segment if required by the operating system.

Return Value

The **DEVICE_HANDLE** parameter is a pointer to a structure of type **DEVICE_RECORD.** This structure contains 1553 board specific information that is used by other library calls. The components of this structure are:

```
typedef struct device_record
{
  BOOLEAN       is_open;
  UWORD         device_number;
  char          device_name[81];
  short int     channel_number;
```

```
  ULONG         control_base_addr;
  char          product_name[21];
  DEVICE_TYPE   kind;
  BASENAME      *base_address;
  BASENAME      *physical_address;
  ULONG         ram_size;
  UWORD         file_descriptor;
  UWORD         io_base;
  UWORD         int_vector;
  char          extra_name[81];
  int           global_sec_name;
  ULONG         next_free_space;
  UWORD         page_size;
  UWORD         number_of_pages;
  UWORD         active_rt_page;
#ifdef DPMI
  UWORD         selector;
#endif
} DEVICE_RECORD;

typedef DEVICE_RECORD *DEVICE_HANDLE;
```

Diagnostics

A **NULL** is returned if the call fails, and the global variable **error_exception** will contain one of the following exceptions:

DEVICE_PROBE_FAIL	The write/read probe test failed. The board was not accessed properly. The probable causes are: power is off or the device is addressed incorrectly.
MEMORY_ALLOCATION	Allocation of heap space for device structure failed.
OPEN_FAIL	Opening the 1553 device failed. The function **open_device()** in *low_lvl.c* raises this exception.
SHARED_MEMORY_FAIL	Attachment to mapped memory failed. The function **initialize_shared_memory()** in *low_lvl.c* raises this exception.

See Also

close_1553(), init_1553_device()

read_time Device Management Routines

Obtain the current time from the internal timer registers

Syntax

#include "sbs_incl.h"

INT_TIMER read_time(device)
DEVICE_HANDLE device;

Description

read_time() accesses the 1553 device internal timer registers and returns the value.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

The return value is of type **INT_TIMER**, defined as follows for the standard time format:

```
typedef struct int_timer
{
     UWORD  high 16;
     UWORD  mid 16;
     UWORD  low 16;
}    INT_TIMER;
```

If the IRIG option is selected in the *system.h* file, INT_TIMER is defined as follows:

```
typedef struct int_timer
{
     ULONG  high 32;
     ULONG  low 32;
}    INT_TIMER;
```

reset_channel Device Management Routines

Reopen and reinitialize a channel of the 1553 device

Syntax

```
#include "sbs_incl.h"

DEVICE_HANDLE reset_channel(device_index, queue_length, seq_mon_buf_length)
int             device_index;
unsigned int    queue_length;
unsigned int    seq_mon_buf_length;
```

Description

reset_channel() reinitializes one channel of the 1553 bus device for operation by calling all required initialization subprograms. The subprograms are **open_1553()**, **init_interrupt_queue()**, **create_bsm_buffer()**, **start_firmware()**, **start_application()**, and **initialize_channel()**.

NOTE: **open_1553()**, **init_interrupt_queue()**, and **create_bsm_buffer()** are only called if necessary. **start_firmware()** and **start_application()** are only called if the application has not been previously started.

Parameters

The **device_index** parameter indicates which 1553 device to open. It is used as an index by the parser utility when the file *1553dev.cfg* is read. For embedded systems without access to this file, system information can be obtained from the *dev_conf.h* header file at compile time. In this case, the directive **NO_PARSE** must be declared so that **device_index** will be used as an index into an array of structures of type **DEVICE_CONFIG**. This type is defined and initialized in *dev_conf.h* and contains system dependent fields necessary to open the 1553 device and attach it to a shared memory segment, if required by the operating system.

The **queue_length** parameter contains the number of interrupt queue entries desired for each interrupt queue. Two interrupt queues, for double buffering purposes, will be initialized if the queue length is nonzero. Interrupt queue initialization is only required if interrupts are to be used by the application program. If the queue length is zero, no interrupt queues will be allocated.

The **seq_mon_buf_length** parameter contains the desired length in words for each bus sequential monitor buffer. Two buffers, for double buffering purposes, will be initialized if

this parameter is nonzero. If this parameter is zero, no sequential monitor buffers will be allocated.

Return Value

The return value, of type **DEVICE_HANDLE**, is a pointer referring to a structure of type **DEVICE_RECORD**. This structure contains 1553 board specific information that is used by other library calls. The components of this structure are listed in the description of **open_1553()**.

Diagnostics

A **NULL** value is returned if the call fails and the global variable **error_exception** will contain the following:

DEVICE_PROBE_FAIL	The write/read probe test failed. The board was not accessed properly. Probable causes are: power is off or the device is addressed incorrectly.
MEMORY_ALLOCATION	Allocation of heap space for the device structure failed.
OPEN_FAIL	Opening the 1553 device failed. The function **open_1553()** in *low_lvl.c* raises this exception.
OUT_OF_MEMORY	Allocation of memory on the 1553 device has failed.
SHARED_MEMORY_FAIL	Attachment to mapped memory failed. The function **initialize_shared_memory()** in *low_lvl.c* raises this exception.
LOAD_ERROR	The RAM load failed.
APP_MODE_ERROR	The switch to application mode failed.
CHANNEL_INIT_ERROR	The channel initialization failed.

See Also

open_1553(), **init_interrupt_queue()**, **create_bsm_buffer()**, **start_firmware()**, **start_application()**, **initialize_channel()**

set_host_dma Device Management Routines

Writes the host DMA control information to the DMA Software Control Registers

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_host_dma ( device, dma_control )
DEVICE_HANDLE     device;
DMA_CONTROL       dma_control;
```

Description

set_host_dma() writes the DMA Control, ABI Address, VME Address, and DMA word count values to the Software Control registers at offsets 15, 16, 17, and 18, respectively.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **dma_control** parameter is a structure used to initialize the DMA information.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

See Also

init_1553_device(), open_1553()

start_firmware Device Management Routines

Start the execution of the firmware

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN start_firmware (device)
DEVICE_HANDLE device;

Description

start_firmware() starts the execution of the firmware by switching from BIT mode to application mode.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

A **NULL** value is returned if the call fails and the global variable **error_exception** will contain one of the following exceptions:

APP_MODE_TIMEOUT	A timeout occurred during the switch to application mode.

See Also

init_1553_device(), open_1553()

start_io Device Management Routines

Start 1553 input and output operations on a channel

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN start_io (device)
DEVICE_HANDLE device;

Description

start_io() enables the 1553 device to perform bus operations by issuing Host Command 2. This procedure should be called to start 1553 bus processing, usually after the bus controller, remote terminal, and bus monitor data structures have been defined.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

A **NULL** value is returned if the call fails and the global variable **error_exception** will contain one of the following exceptions:

CHANNEL_NOT_INITIALIZED	The channel must be initialized by calling initialize_channel before 1553 processing can be started.
ENABLE_CHANNEL_ERROR	An enable channel host command error occurred.
ENABLE_CHANNEL_TIMEOUT	A timeout occurred during host command processing.

See Also

init_1553_device(), open_1553(), stop_io()

stop_io Device Management Routines

Stop 1553 input and output operations on a channel

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN stop_io (device)
DEVICE_HANDLE device;

Description

stop_io() disables the 1553 device by issuing Host Command 3.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

A **NULL** value is returned if the call fails and the global variable **error_exception** will contain one of the following exceptions:

DISABLE_CHANNEL_TIMEOUT	A timeout occurred during host command processing.
DISABLE_CHANNEL_ERROR	A disable channel host command error occurred.

See Also

init_1553_device(), open_1553(), start_io()

stub_select Device Management Routines

Change the stub selection

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  stub_select ( device, stub_value )
DEVICE_HANDLE     device;
VALID_STUB_VALUE  stub_value;
```

Description

stub_select() provides the capability to change the stub length between short stub (stub_value = nonzero) and long stub (stub_value = 0). Long stub is the default.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **stub_value** parameter selects long stub (0) or short stub (1).

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

define_mon_rt_sa Remote Terminal Routines

Allocate a transmit subaddress for a remote terminal mapped monitor

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN define_mon_rt_sa ( device, sa_info )
DEVICE_HANDLE        device;
SA_HANDLE            sa_info;
```

Description

define_mon_rt_sa() allocates a subaddress for a remote terminal mapped monitor. It defines the appropriate data structures and sets the filter table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a pointer to a structure containing information relative to the initialization of the subaddress. The components of this structure are listed in the description of **define_rt_sa()**.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553()

define_rt_sa — Remote Terminal Routines

Define a remote terminal subaddress

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN define_rt_sa (device, sa_info)
DEVICE_HANDLE device;
SA_HANDLE sa_info;

Description

define_rt_sa() defines and sets up a remote terminal subaddress. This routine allocates and initializes the appropriate data structures and data buffers in the 1553 device's memory, initializes the status response word for the remote terminal, and enables the remote terminal for emulation. Multiple data buffers can be allocated for a subaddress. The routine **set_global_control()** can be used to enable mapped monitor mode.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a structure containing information relative to the initialization of the subaddress. This structure must be properly initialized by the user before making the call. The components of this structure and their descriptions are:

```
typedef struct sa_info
{
  VALID_RT            rt;
  VALID_SA            sa;
  MESSAGE_TYPE        t_r;
  UWORD               num_buffers;
  ULONG               buff_id[MAX_BUFFER];
  int                 filter_bit;
  int                 swap_bit;
  int                 interrupt_bit;

}  SA_INFO_TYPE;

typedef SA_INFO_TYPE  *SA_HANDLE;
```

The **rt** and **sa** components define the remote terminal and subaddress.

The **t_r** component specifies whether the subaddress is to receive or transmit data. The enumerated type **MESSAGE_TYPE** has literal values of **RECEIVE** and **TRANSMIT**.

The **num_buffers** component specifies the number of data buffers to allocate for the subaddress.

The **buffer_id** component is an array initialized by the call that contains an address offset to each control buffer that is allocated. These offsets allow access to the control buffers which contain offsets to the data buffers. The data buffers are used by **read_sa_buffer()** and **write_sa_buffer()**.

The **filter_bit** specifies whether sequential monitoring is enable for the subaddress.

The **swap_bit** specifies whether to swap upon receipt of an error free message.

The **interrupt_bit** specifies whether to interrupt when a message is received.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain one of the following:

OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.
DEFINE_RT_ERROR	The "Update Single RT Status Response Table Entry" host command failed.
DEFINE_RT_TIMEOUT	A timeout occurred during host command processing.

See Also

init_1553_device(), open_1553(), read_sa_buffer(), **set_global_control(), set_trx_status_control(), update_rt_stat_resp_entry(), write_sa_buffer32()**

read_filter_word Remote Terminal Routines

Read the transmit or receive filter word for an RT

Syntax

```
#include "sbs_incl.h"

ULONG  read_filter_word ( device, sa_info )
DEVICE_HANDLE    device;
SA_HANDLE        sa_info;
```

Description

read_filter_word() reads the transmit (offset 2) or receive (offset 1) filter word for an RT from the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a structure containing information relative to the initialization of the subaddress. This structure must be properly initialized by the user before making this call. The components of this structure are listed in the description of **define_rt_sa()**.

Return Value

This function returns the 32-bit value of the specified filter word.

See Also

define_rt_sa(), init_1553_device(), open_1553(), set_filter_word()

read_mc17_data Remote Terminal Routines

Read the data word associated with mode code 17

Syntax

```
#include "sbs_incl.h"

UWORD  read_mc17_data ( device, rt )
DEVICE_HANDLE    device;
VALID_RT         rt;
```

Description

read_mc17_data() returns the data word associated with the Synchronize Mode Code, mode code 17.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal number for the Synchronize Mode Code.

Return Value

The value returned is the data word associated with the Synchronize Mode Code sent to the corresponding remote terminal.

Diagnostics

The global variable **error_exception** will contain one of the following:

NO_EXCEPTION	No error.
UNDEFINED_RT	The RT to be read does not exist.

See Also

init_1553_device(), open_1553()

read_rt_int_word Remote Terminal Routines

Read the transmit or receive interrupt word for an RT

Syntax

```
#include "sbs_incl.h"

ULONG  read_rt_int_word ( device, sa_info )
DEVICE_HANDLE    device;
SA_HANDLE        sa_info;
```

Description

read_rt_int_word() reads the transmit (offset 6) or receive (offset 5) interrupt word for an RT from the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a structure containing information relative to the initialization of the subaddress. This structure must be properly initialized by the user before making this call. The components of this structure are listed in the description of **define_rt_sa()**.

Return Value

This function returns the 32-bit value of the specified Interrupt on Message Received word.

See Also

define_rt_sa(), init_1553_device(), open_1553(), set_rt_int_word()

read_sa_buffer Remote Terminal Routines

Read the data buffer of a receive subaddress using VMEbus long word access

Syntax

```
#include "sbs_incl.h"

int  read_sa_buffer ( device, ctl_buffer_id, buffer )
DEVICE_HANDLE    device;
ULONG            ctl_buffer_id;
DATA_BUFFER      buffer;
```

Description

read_sa_buffer() reads the contents of a receive subaddress data buffer using VMEbus long word (32-bit) access.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **ctl_buffer_id** parameter is an address offset pointing to the RT control buffer which contains the data buffer offset. The procedure **define_rt_sa()** creates this offset and returns it to the caller as **sa_info.buffer_id**.

The **buffer** parameter is a data buffer allocated by the caller. The contents of the subaddress data buffer are copied to this buffer. The procedure uses the word count field of the subaddress data buffer's Flag Word to determine the number of words to copy.

Return Value

This function returns the number of words read or 0 if the subaddress data buffer is stale.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_sa_buffer(), write_sa_buffer()

read_status_word Remote Terminal Routines

Read the Status Response Word for the specified remote terminal

Syntax

```
#include "sbs_incl.h"

STATUS_RESPONSE_TYPE  read_status_word ( device, rt )
DEVICE_HANDLE    device;
VALID_RT         rt;
```

Description

read_status_word() returns the Status Response Word from the RT Status Response Table for a remote terminal.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal address for the RT Status Response Table.

Return Value

The **status_word** parameter is a structure representing the Status Response Word that will be transmitted by the remote terminal. See **set_staus_word()** for a listing of the components of this structure.

See Also

define_rt_sa(), init_1553_device(), set_status_word(), open_1553()

read_swap_word — Remote Terminal Routines

Read the transmit or receive sequential monitor swap word for an RT

Syntax

```
#include "sbs_incl.h"

ULONG  read_swap_word ( device, sa_info )
DEVICE_HANDLE    device;
SA_HANDLE        sa_info;
```

Description

read_swap_word() reads the transmit (offset 3) or receive (offset 4) sequential monitor swap word for an RT from the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a structure containing information relative to the initialization of the subaddress. This structure must be properly initialized by the user before making this call. The components of this structure are listed in the description of **define_rt_sa()**.

Return Value

This function returns the 32-bit value of the specified Swap on Message word.

Diagnostics

If an error is detected, the return value will be FFFFFFFFh and the global variable **error_exception** will contain the following:

UNDEFINED_RT	The RT specified in **sa_info** has not been defined.

See Also

define_rt_sa(), init_1553_device(), open_1553(), set_swap_word()

read_transmit_sa_buffer Remote Terminal Routines

Read the data buffer of a transmit subaddress using VMEbus long word access

Syntax

```
#include "sbs_incl.h"

int  read_transmit_sa_buffer ( device, buffer_id, buffer )
DEVICE_HANDLE    device;
VALID_BUFFER_ID  buffer_id;
DATA_BUFFER      buffer;
```

Description

read_transmit_sa_buffer() reads the contents of a transmit subaddress data buffer using VMEbus long word (32-bit) access.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **buffer_id** parameter is an address offset pointing to the subaddress data buffer. The procedure **define_rt_sa()** creates this offset and returns it to the caller as **sa_info.buffer_id**.

The **buffer** parameter is a data buffer allocated by the caller. The contents of the subaddress data buffer are copied to this buffer. The procedure uses the word count field of the subaddress data buffer's Flag Word to determine the number of words to copy.

Return Value

This function returns the number of words read or 0 if the subaddress data buffer is stale.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_sa_buffer(), write_sa_buffer()

rt_wrap Remote Terminal Routines

Enable or disable an RT to "wrap"

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  rt_wrap ( device, sa_info_r, sa_info_t, wrap )
DEVICE_HANDLE     device;
SA_HANDLE         sa_info_r;
SA_HANDLE         sa_info_t;
BOOLEAN           wrap;
```

Description

The **rt_wrap()** routine allows a BC to test the RT by sending a BC-RT message immediately followed by a RT-BC message. The RT should then send back the same data that the BC sent to it.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info_r** parameter is a pointer to the **SA_INFO** structure for the receive subaddress of the RT for which to enable or disable the wrap function.

The **sa_info_t** parameter is a pointer to the **SA_INFO** structure for the transmit subaddress of the RT for which to enable or disable the wrap function. The **SA_INFO** structures for **sa_info_r** and **sa_info_t** should specify the same RT and SA.

The **wrap** parameter determines if the call will enable (TRUE) or disable (FALSE) the wrap function. If **wrap** is **TRUE**, the **rt_wrap** routine will set the transmit buffer pointer of the RT/SA specified by **sa_info_t** to match the receive buffer pointer of the RT/SA specified by **sa_info_r**. If **wrap** is **FALSE**, the **rt_wrap** routine will set the transmit buffer pointer of the RT/SA specified by **sa_info_t** back to the transmit buffer which was originally allocated (given in the **sa_info_t->buffer_id[0]** field).

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain one of the following:

DEVICE_NOT_OPEN	The **device** parameter contains a NULL value.
UNDEFINED_RT	There is no RT definition table for the specified remote terminal.
UNDEFINED_SUBADDRESS	There is no control buffer for the specified subaddress.

See Also

init_1553_device(), define_rt_sa()

set_filter_word — Remote Terminal Routines

Writes a value to the transmit or receive filter word in the RT Definition Table

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_filter_word ( device, sa_info, filter_word )
DEVICE_HANDLE     device;
SA_HANDLE         sa_info;
ULONG             filter_word;
```

Description

set_filter_word() writes a value to the transmit (offset 1) or receive (offset 2) filter word in the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a structure containing information relative the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the **sa_info** structure are listed in the description of **define_rt_sa()**.

The **filter_word** parameter contains the value to be written to the filter word in the RT definition table.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_filter_word()

set_global_control Remote Terminal Routines

Write a value to the Global Control word

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_global_control ( device, rt, value )
DEVICE_HANDLE     device;
VALID_RT          rt;
UWORD             value;
```

Description

set_global_control() writes a value to the Global Control word for a specific remote terminal.

NOTE: **update_rt_stat_resp_entry** must be called for RT Status Response Table changes to take effect.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal address for the Protocol Table.

The **value** parameter is a number representing an entry in the Protocol Table.

Return Value

EXCEPTION_RETURN returns **SUCCESS** or **FAILURE**.

See Also

init_1553_device(), open_1553(), **update_rt_stat_resp_entry()**

set_illegal_mode_code Remote Terminal Routines

Set or clear a specified bit in the Illegal Mode Code word

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN set_illegal_mode_code ( device, mode_code, mc_state )
DEVICE_HANDLE       device;
MODE_CODE_TYPE      mode_code;
BOOLEAN             mc_state;
```

Description

THIS FUNCTION ONLY APPLIES TO ABI-V5 FIRMWARE WHICH INCLUDES THE ILLEGAL COMMAND OPTION.

set_illegal_mode_code() sets or clears a specified bit representing a mode code in the Illegal Transmit or Receive Mode Code Word. It then calls issue_host_cmd() to activate the change. For example, the following function declares mode code 2 illegal:

```
set_illegal_mode_code(device, 2, TRUE);
```

Mode code 2 is declared legal again by the following:

```
set_illegal_mode_code(device, 2, FALSE);
```

See Appendix A of the ABI-V5 Reference Manual for more information on this option.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **mode_code** parameter is of type **MODE_CODE_TYPE**, and can be any valid mode code number.

The **mc_state** parameter is of type **BOOLEAN**. If it is **TRUE**, the specified mode code will be made illegal. If it is **FALSE**, the specified mode code will be made legal.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_MODE_CODE	The specified mode code does not exist.
INVALID_FW_FOR_OPERATION	The loaded firmware does not support this function.
HOST_CMD_TIMEOUT	A timeout occurred during host command processing.
HOST_CMD_ERROR	A host command error occurred.

See Also

issue_host_cmd()

set_illegal_sa Remote Terminal Routines

Set or clear a specified bit in the Illegal SA Word in the appropriate RT Definition Table entry

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN set_illegal_sa ( device, sa_info, bit_state )
DEVICE_HANDLE       device;
SA_HANDLE           sa_info;
BOOLEAN             bit_state;
```

Description

THIS FUNCTION ONLY APPLIES TO ABI-V5 FIRMWARE WHICH INCLUDES THE ILLEGAL COMMAND OPTION.

set_illegal_sa() sets or clears a specified bit representing a SA in the Illegal Transmit or Receive SA Word in the appropriate RT Definition Table entry. It then calls issue_host_cmd() to activate the change. For example, if **sa_info** has been set up to specify RT 5, SA 2, the following function declares SA 2 illegal for RT 5:

```
set_illegal_sa(device, sa_info, TRUE);
```

SA 2 is declared legal again for RT 5 by the following:

```
set_illegal_sa(device, sa_info, FALSE);
```

See Appendix A of the ABI-V5 Reference Manual for more information on this option.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a pointer to a structure containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the **sa_info** structure are listed in the description of **define_rt_sa()**.

The **bit_state** parameter is of type **BOOLEAN**. If it is **TRUE**, the specified SA will be made illegal for the specifed RT. If it is **FALSE**, the specified SA will be made legal for the specified RT.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.
INVALID_FW_FOR_OPERATION	The loaded firmware does not support this function.
HOST_CMD_TIMEOUT	A timeout occurred during host command processing.
HOST_CMD_ERROR	A host command error occurred.

See Also

issue_host_cmd(), **define_rt_sa()**

set_int_on_rt_processed Remote Terminal Routines

Sets the RT buffer control word

Syntax

#include "sbs_incl.h"

void set_int_on_rt_processed (device, **buffer_id, interrupt_flag)**
DEVICE_HANDLE device;
VALID_BUFFER_ID buffer_id;
BOOLEAN interrupt_flag;

Description

set_int_on_rt_processed() sets the Buffer Control word in the RT Control Buffer to "1", enabling the RT Buffer Processed interrupt.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **buffer_id** parameter contains the offset to the RT Data Control Buffer.

The **interrupt_flag** parameter contains the state of the RT Buffer Processed interrupt. If the value is **TRUE**, the interrupt will be enabled

See Also

define_rt_sa()

set_mc16_data — Remote Terminal Routines

Set the data word for mode code 16

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_mc16_data ( device, rt, tvw )
DEVICE_HANDLE     device;
VALID_RT          rt;
UWORD             tvw;
```

Description

set_mc16_data() sets the data word for the Transmit Vector Word Mode Code, mode code number 16.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal address for the Transmit Vector Word Mode Code.

The **tvw** parameter contains the data word associated with the Transmit Vector Word Mode Code that will be transmitted from the corresponding remote terminal.

Return Value

EXCEPTION_RETURN returns **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.

See Also

init_1553_device(), open_1553()

set_mc19_data — Remote Terminal Routines

Set the data word for mode code 19

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_mc19_data ( device, rt, bit_word )
DEVICE_HANDLE     device;
VALID_RT          rt;
UWORD             bit_word;
```

Description

set_mc19_data() sets the data word for the Transmit BIT Word Mode Code, mode code number 19.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal address for the Transmit BIT Word Mode Code.

The **bit_word** parameter contains the data word associated with the Transmit BIT Word Mode Code that will be transmitted from the corresponding remote terminal.

Return Value

EXCEPTION_RETURN returns **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.

See Also

init_1553_device(), open_1553()

set_mim Remote Terminal Routines

Set the mode command interrupt mask

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_mim ( device, rt, mode_code )
DEVICE_HANDLE     device;
VALID_RT          rt;
MODE_CODE_TYPE    mode_code;
```

Description

set_mim() sets the mode command interrupt mask (offset 7) for the specified mode code. This mask is used to generate an interrupt whenever a specific mode command message is received.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal address for the mode command interrupt mask.

The **mode_code** parameter is an enumerated type representing the mode code that will be masked. Valid values for **mode_code_type** and their MIL-STD-1553B functions are:

dyn_bus_cntrl	Dynamic Bus Control
synch	Synchronize
xmit_status_word	Transmit Status Word
i**nit_self_test**	Initiate Self-Test
xmittr_shutdown	Transmitter Shutdown
override_xmittr_shutdown	Override Transmitter Shutdown
inhibit_flag_bit	Inhibit Terminal Flag Bit
override_inhibit_flag_bit	Override Terminal Inhibit Flag Bit
reset_rt	Reset Remote Terminal
xmit_vector_word	Transmit Vector Word
synch_w_data	Synchronize with Data Word

xmit_last_command	Transmit Last Command
xmit_bit_word	Transmit Built-in-Test Word
select_xmittr_shutdown	Selected Transmitter Shutdown
override_select_xmittr_shutdown	Override Selected Transmitter Shutdown

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT is not allocated.

See Also

init_1553_device(), open_1553()

set_rt_dma — Remote Terminal Routines

Write DMA control information to an RT data control buffer

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_rt_dma( device, buffer_id, dma_control )
DEVICE_HANDLE     device;
VALID_BUFFER_ID   buffer_id;
DMA_CONTROL_BLOCK dma_control;
```

Description

set_rt_dma() writes DMA control, ABI address, VME address, and DMA word count values in the RT Data Control Buffer at offsets 3, 4, 5, and 6, respectively.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **buffer_id** parameter contains the offset to the RT Data Control Buffer.

The **dma_control** parameter is a structure that is used to initialize the DMA information.

Return Value

EXCEPTION_RETURN returns **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_BUFFER	The specified RT Data Control Buffer does not exist.

See Also

init_1553_device(), open_1553()

set_rt_int_word — Remote Terminal Routines

Writes a value to the transmit or receive interrupt on message received word in the RT Definition Table

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_rt_int_word ( device, sa_info, value )
DEVICE_HANDLE     device;
SA_HANDLE         sa_info;
ULONG             value;
```

Description

set_rt_int_word() writes a value to the transmit (offset 5) or receive (offset 6) interrupt on message received word in the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a structure containing information relative the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the structure are listed in the description of **define_rt_sa()**.

The **value** parameter contains the value that is copied to the RT Definition Table.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_rt_int_word()

set_status_word — Remote Terminal Routines

Set the Status Response Word for the specified remote terminal

Syntax

#include "sbs_incl.h"

void set_status_word (device, rt, status_word)
DEVICE_HANDLE device;
VALID_RT rt;
STATUS_RESPONSE_TYPE status_word;

Description

set_status_word() sets the Status Response Word in the RT Status Response Table for a remote terminal. The procedure **define_rt_sa()** will also initialize the Status Response Word for the remote terminal whenever it is called.

NOTE: **update_rt_stat_resp_entry()** must be called for RT Status Response Table entry changes to take effect.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal address for the RT Status Response Table.

The **status_word** parameter is a record representing the Status Response Word that will be transmitted by the remote terminal. The **rt_address** component will be set by the call. The components of this record are:

```
typedef struct status_response_type {
    unsigned int     rt_address                  : 5;
    unsigned int     message_error               : 1;
    unsigned int     instrumentation             : 1;
    unsigned int     service_request             : 1;
    unsigned int     unused                      : 3;
    unsigned int     broadcast_command_received  : 1;
    unsigned int     busy                        : 1;
    unsigned int     sub_bus_system_flag         : 1;
    unsigned int     bus_control_accept          : 1;
    unsigned int     tf_bit                      : 1;
} STATUS_RESPONSE_TYPE;
```

See Also

define_rt_sa(), init_1553_device(), open_1553()

set_swap_word — Remote Terminal Routines

Writes a value to the transmit or receive sequential monitor swap word in the RT Definition Table

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  set_swap_word ( device, sa_info, swap_word )
DEVICE_HANDLE     device;
SA_HANDLE         sa_info;
ULONG             swap_word;
```

Description

set_swap_word() writes a value to the transmit (offset 3) or receive (offset 4) sequential monitor swap word in the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa_info** parameter is a structure containing information relative the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the **sa_info** structure are listed in the description of **define_rt_sa()**.

The **swap_word** parameter contains the value that is copied to the RT Definition Table.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_swap_word()

set_trx_status_control Remote Terminal Routines

Write an entry to the RT Status Transmit Control word for an RT

Syntax

```
#include "sbs_incl.h"

void  set_trx_status_control ( device, rt, value )
DEVICE_HANDLE    device;
VALID_RT         rt;
UWORD            value;
```

Description

set_trx_status_control() writes a value to the RT Status Transmit Control word for an RT.

NOTE: **update_rt_stat_resp_entry** must be called for RT Status Response Table changes to take effect.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal address for the RT Status Transmit Control word.

The **value** parameter contains the hexadecimal value that is written to the RT Status Transmit Control word.

See Also

update_rt_stat_resp_entry()

update_rt_stat_resp_entry Remote Terminal Routines

Updates the RT Status Response Table entry for an RT

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  update_rt_stat_resp_entry ( device, rt )
DEVICE_HANDLE     device;
VALID_RT          rt;
```

Description

update_rt_stat_resp_entry() issues host command 7 to update the RT Response Table entry for an RT. This routine is required after a call to **set_status_word()**, **set_trx_status_control()**, or **set_global_control()**.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal address for the Status Response Table update.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**

See Also

set_status_word(), set_trx_status_control(), **set_global_control()**

write_sa_buffer Remote Terminal Routines

Write data to a transmit subaddress data buffer using VMEbus long word access

Syntax

```
#include "sbs_incl.h"

int write_sa_buffer ( device, word_count, ctl_buffer_id, out_buffer )
DEVICE_HANDLE       device;
VALID_WORD_COUNT    word_count;
VALID_BUFFER_ID     ctl_buffer_id;
DATA_BUFFER         out_buffer;
```

Description

write_sa_buffer() copies data to a transmit subaddress data buffer using VMEbus long word (32-bit) access.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **word_count** parameter defines the number of words to write to the subaddress data buffer.

The **ctl_buffer_id** parameter is a unique pointer to the control buffer. This pointer is established by the routine **define_rt_sa().**

The **out_buffer** parameter is a data buffer allocated by the caller. The contents of this buffer are copied to the subaddress data buffer.

Return Value

This function returns the number of words written or 0 if the subaddress data buffer is stale.

See Also

define_rt_sa(), init_1553_device(), open_1553(), read_sa_buffer()

add_link_to_chain — Bus Controller Routines

Add a link to a bus controller chain program

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN add_link_to_chain (device, link)
DEVICE_HANDLE device;
LINK_HANDLE link;

Description

add_link_to_chain() is used to build a bus controller program (also called a chain). A chain consists of multiple bus controller command blocks (also called links) each representing one bus controller command. An application can define any number of chain programs. The number of links in a chain is limited by the amount of memory available on the 1553 device or by the application. One link will be added to a chain each time this routine is called.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **link** parameter is a pointer referring to a structure of type **CHAIN_LINK_INFO**. This structure defines the bus controller command to add to the chain program. The contents of this structure are:

```
typedef struct chain_link_info
{
 VALID_BC_COMMAND   command;
 VALID_CHAIN_ID     chain_id;
 VALID_RT           rt1;
 VALID_SA           sa1;
 VALID_WORD_COUNT   word_count;
 VALID_LINK_ID      link_id;
 BOOLEAN            loop_flag;
 VALID_RT           rt2;
 VALID_SA           sa2;
 ULONG              bc time delay;
 UWORD              num_buffers;
 ULONG              buff_id [ MAX_BUFFER ];
 {
  BC_CTL_WORD_TYPE  flags;
```

```
  unsigned long     flags_;
 } flags;
 union
 {
  ERROR_WORD_TYPE   errors;
  unsigned short    errors_;
 } errors;
} CHAIN_LINK_INFO;

typedef CHAIN_LINK_INFO  *LINK_HANDLE;
```

The **command** component defines the type of transfer. When **bc_command** is **mode**, place the **mode_code** value in the **word_count** field. When **bc_command** is **rtrt**, the **rt1** and **sa1** components define the destination remote terminal and subaddress, and the **rt2** and **sa2** components define the source remote terminal and subaddress for the command.

The **chain_id** component defines the chain program where the link will be added. If the chain is undefined, the link will become the head of a new chain.

The **rt1** and **sa1** components define the remote terminal and subaddress for the command.

The **word_count** component specifies the number of words to be transferred for the command.

The **link_id** component is a unique link identifier that is initialized by the call. It is used by **read_link_data()**, r**ead_link_status()**, **set_flag_word()**, and **write_link_data32()** procedures to refer to this link.

The **loop_flag** component specifies whether or not the link is to loop back to the head of the chain after execution or terminate. A value of **TRUE** will cause the chain program to loop back to the top and execute continuously. A value of **FALSE** will stop the chain program after the last block is executed. The **loop_flag** is only valid for the last link in a chain.

The **rt2** and **sa2** components define the remote terminal and subaddress for the second command word for rt-rt transfers.

The **bc_time_delay** component specifies the 16-bit time for a delay block. The delay time is represented in microseconds.

If **bc_time_delay** is nonzero for blocks other than delay blocks, this is the 23-bit minor frame schedule time. Use the following formula to determine the value to use in **bc_time_delay**:

bc_time_delay *= (minor frame time in* µs*) - (schedule time in* µs*)*

The **num_buffers** component specifies the number of data buffers to allocate for the link.

The **buff_id** component contains the buffer address for each buffer that is defined.

The **dma_enable** component specifies whether or not DMA is enabled for the first data buffer.

The **enable_db_swap** component specifies whether or not the Enable Database Swap bit is set in the Data Buffer Pointer/Control word.

The **buff_processed_int** component specifies whether or not the Interrupt on Buffer Processed bit is set in the Data Buffer Pointer/Control word.

The **flags** component is a union of a structure and a word. The structure represents the control word found in word 14 of a chain program command. The control word is used to specify the response of the 1553 device during certain conditions. This routine uses the **flags** component for initialization. The routine **set_flag_word()** will also set a new value in a link's control word. The components of this structure are:

```
typedef struct bc_flags_type
{
     unsigned  continue_on_error      : 1;
     unsigned  bc_retry               : 1;
     unsigned  trigger_out            : 1;
     unsigned  trigger_in             : 1;
     unsigned  mframe_start           : 1;
     unsigned  no-op                  : 1;
     unsigned  unused3                : 2;
     unsigned  int_on_error           : 1;
     unsigned  int_on_blk_complete    : 1;
     unsigned  int_on_BC_halt         : 1;
     unsigned  int_on_retry_attempt   : 1;
     unsigned  int_on_masked_status   : 1;
     unsigned  unused2                : 2;
     unsigned  bus_b                  : 1;
     unsigned  retry_count            : 5;
     unsigned  retry_success          : 1;
     unsigned  unused1                : 9;
     unsigned  retry_bus_on_success   : 1;
} BC_CTL_WORD_TYPE;
```

NOTE: The bits are defined in reverse order when BIT_FIELD1 is included in 1553dev.cfg or dev_conf..h.

The **error_word** component is a union of a structure and a word. The structure represents the error injection word found at offset 17 from the beginning of a chain program command. The error word is used to specify the errors defined for the 1553 BC command.

```
typedef struct  error type
{
     unsigned  error_word             : 6;
     unsigned  long_message           : 1;
     unsigned  short_message          : 1;
     unsigned  insert_gap             : 1;
```

```
    unsigned  unused1               : 1;
    unsigned  manchester_error      : 1;
    unsigned  too_few_bits          : 1;
    unsigned  too_many_bits         : 1;
    unsigned  parity_error          : 1;
    unsigned  sync_error            : 1;
    unsigned  unused2               : 1;
} ERROR_WORD_TYPE;
```

NOTE: The bits are defined in reverse order when BIT_FIELD1 is included in 1553dev.cfg or dev_conf..h.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain one of the following:

MODE_CODE_INIT_FAIL	Improper mode code subaddress. The link is still added to the chain; however, the subaddress will equal zero.
OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.
UNDEFINED_MODE_CODE	Invalid 1553B mode code. The link is still added to the chain.

See Also

init_1553_device(), open_1553(), read_link_data(), read_link_status(), set_flag_word(), write_link_data()

delink_chain Bus Controller Routines

Uncombine a chain program to form two chains

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN delink_chain ( device, chain_1, chain_2 )
DEVICE_HANDLE    device;
VALID_CHAIN_ID   chain_1;
VALID_CHAIN_ID   chain_2;
```

Description

delink_chain() uncombines a chain program to form two chains. This procedure would be used for chains that have been previously linked together using the **link_chain()** call.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **chain_1** parameter contains an identifier that refers to the chain to delink. The chain is delinked by clearing the tail pointer of its last command block. Thus, a chain that was initialized to loop for continuous execution can be delinked to execute only once on its next cycle.

The **chain_2** parameter contains an identifier that refers to the chain to delink. The chain is delinked by clearing the tail pointer of its last command block. Thus, a chain that was initialized to loop for continuous execution can be delinked to execute only once on its next cycle.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_CHAIN The **chain_id** parameter is invalid.

See Also

init_1553_device(), link_chain(), open_1553()

disable_frame_timer Bus Controller Routines

Disables the minor frame timer

Syntax

#include "sbs_incl.h"

void disable_frame_timer (device)
DEVICE_HANDLE device;

Description

disable_frame_timer() is used to disable the minor frame timer by clearing the Enable Timer bit in the BC Frame Timer Register (offset 0E from the channel control base address).

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

See Also

enable_frame_timer()

enable_frame_timer Bus Controller Routines

Enables the BC Minor Frame Timer and sets the frame interval

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN  enable_frame_timer ( device, time )
DEVICE_HANDLE     device;
ULONG             time;
```

Description

enable_frame_timer() is used to enable the minor frame timer (by setting the Enable Timer bit) and set the frame interval in the BC Frame Timer Register (offset E from the channel control base address).

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **time** parameter contains the time to use for the frame interval.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable error_exception will contain:

MAX_TIME_EXCEEDED The frame interval is limited to 23 bits.

See Also

disable_frame_timer()

link_chain — Bus Controller Routines

Combine two chain programs to form one

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN link_chain (device, chain_1, chain_2)
DEVICE_HANDLE device;
VALID_CHAIN_ID chain_1;
VALID_CHAIN_ID chain_2;

Description

link_chain() combines two chain programs to form one larger chain. The chains can later be uncombined using the procedure **delink_chain()**.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **chain_1** parameter contains an identifier that refers to the first chain program. This chain is linked to the second by setting the tail pointer of its last command block to point to the first command block of the second chain.

The **chain_2** parameter contains an identifier that refers to the second chain program. This chain is linked after the first chain and will execute after the last command block of the first chain has completed.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_CHAIN The **chain_id** parameter is invalid.

See Also

delink_chain(), init_1553_device(), open_1553()

load_chain — Bus Controller Routines

Prepare a chain program for execution

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN load_chain ( device, chain_id );
DEVICE_HANDLE    device;
VALID_CHAIN_ID   chain_id;
```

Description

load_chain() prepares a chain program for execution by issuing host command 5, Start BC Chain. A chain must have been previously constructed using one or more **add_link_to_chain()** calls. The chain will execute immediately if bus processing has been started (i.e., RESP, offset 3 from the control base address, is constantly changing). Otherwise, a call to **start_io()** will execute the chain if it has been loaded.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **chain_id** parameter contains an identifier that refers to a specific chain.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_CHAIN	The **chain_id** parameter is invalid.
LOAD_CHAIN_ERROR	A "load chain" host command error occurred.
LOAD_CHAIN_TIMEOUT	A timeout occurrred during host command processing.

See Also

add_link_to_chain(), init_1553_device(), open_1553(), start_io()

read_bc_data_buff_cntl Bus Controller Routines

Read the buffer pointer/control word for a specified BC buffer

Syntax

```
#include "sbs_incl.h"

ULONG read_bc_data_buff_cntl ( device, buff_id );
DEVICE_HANDLE    device;
VALID_BUFFER_ID  buff_id;
```

Description

read_bc_data_buff_cntl() reads the pointer/control word associated with the specified BC data buffer. The primary use of this function is to check bit 31; when set, this bit indicates that an error occurred in the message and that the data in the BC receive buffer may be incomplete. This function can also be used to check the status of the control bits for interrupt on buffer processed, enable database swap, and DMA enable. See Figure BC-7 of the ABI-V5 Reference Manual for details on these bits.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **buff_id** parameter is an address offset pointing to a BC data buffer. The function **add_link_to_chain()** generates this offset and assigns the value to **chain_link_info.buff_id[n]**.

Return Value

The function returns the 32-bit value of the BC data buffer pointer/control word corresponding to the specified buffer.

If the specified buffer is not found, a zero value is returned.

Diagnostics

If a zero value is returned, the specified **buff_id** is invalid.

See Also

set_bc_data_buff_cntl(), add_link_to_chain()

read_link_data Bus Controller Routines

Read the data buffer of a chain link using VMEbus long word access

Syntax

```
#include "sbs_incl.h"

int read_link_data ( device, link_id, buff_id, in_buffer )
DEVICE_HANDLE    device;
VALID_LINK_ID    link_id;
VALID_BUFFER_ID  buff_id;
DATA_BUFFER      in_buffer;
```

Description

read_link_data() reads the data buffer of a previously added link in a chain program using VMEbus long word (32-bit) access. This call may be used to read the data word associated with a mode code.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** routine.

The **buff_id** parameter is an address offset pointing to the BC control block. The procedure **add_link_to_chain()** creates this offset and returns it to the caller as **chain_link_info.buff_id[n]**.

The **in_buffer** parameter is a data buffer allocated by the caller. The contents of the chain link's data buffer are copied to this buffer. The number of words copied to the buffer is determined from the word count field in the chain link's command word.

Return Value

The function returns the number of words copied from the link's data buffer into the caller's data buffer.

See Also

add_link_to_chain(), init_1553_device(), open_1553(), write_link_data()

read_link_status Bus Controller Routines

Read the status response words associated with a chain link

Syntax

#include "sbs_incl.h"

void read_link_status (device, link_id, status)
DEVICE_HANDLE device;
VALID_LINK_ID link_id;
STATUS_BLOCK_TYPE status;

Description

read_link_status() reads the status response words of the specified chain link. This call will retrieve two status response words for rt-rt chain links; otherwise, it will return only one.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** routine.

The **status** parameter is an array that will contain one or more status response words from the previous transfer. The number of status response words updated by the call depends upon the type of transfer. The type **STATUS_BLOCK_TYPE** is defined as:

```
typedef unsigned short STATUS_BLOCK_TYPE[2];
```

See Also

add_link_to_chain(), init_1553_device(), open_1553()

set_bc_control Bus Controller Routines

Set the control word for a specified chain link

Syntax

```
#include "sbs_incl.h"

void set_bc_control ( device, link_id, controls )
DEVICE_HANDLE       device;
VALID_LINK_ID       link_id;
BC_FLAG_WORD_TYPE   controls;
```

Description

set_bc_control() writes a value to the control word (offset 14) of a BC Control Block.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** procedure.

The **controls** parameter is a record representing the control word and is used to specify the response of the 1553 device during certain conditions. See **add_link_to_chain()** for a listing of the components within this record.

See Also

add_link_to_chain(), init_1553_device(), open_1553()

set_bc_data_buff_cntl Bus Controller Routines

Set or clear bits 28, 29, and 30 of the specified BC data buffer pointer/control word

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN set_bc_data_buff_cntl ( device, link_id, buff_id, mask )
DEVICE_HANDLE    device;
VALID_LINK_ID    link_id;
VALID_BUFFER _ID buff_id;
ULONG            mask;
```

Description

set_bc_data_buff_cntl() sets or clears bits 28, 29, and 30 of the specified BC data buffer pointer/control word. Bits 28, 29, and 30 of the **mask** word are the values assigned. If the specified buffer is the currently active buffer for the associated BC control block (specified by **link_id**), bits 28, 29, and 30 of the current data buffer pointer/control (offset 19) in the BC control block are also set to the values specified by the **mask** word.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **link_id** parameter contains the unique identifier that refers to a specific link. This value is assigned by the **add_link_to_chain()** routine and is stored in **chain_link_info.link_id**.

The **buff_id** parameter is an address offset pointing to a BC data buffer. This value is assigned by the **add_link_to_chain()** routine and is stored in **chain_link_info.buff_id[n]**.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain one of the following:

UNDEFINED_CHAIN_DATA_POINTER	The data buffer specified by buff_id was not found.
UNDEFINED_NEXT_POINTER	The next pointer at the address specified by **buff_id** was a NULL pointer.

See Also

read_bc_data_buff_cntl(), add_link_to_chain()

set_bc_dma — Bus Controller Routines

Writes DMA control information to a BC Control Block

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN set_bc_dma ( device, block_id, dma_control )
DEVICE_HANDLE    device;
VALID_LINK_ID    block_id;
DMA_CONTROL      dma_control;
```

Description

set_bc_dma() writes DMA Control, ABI Address, VME Address, and DMA word count values in the BC Control Block at offsets 4, 5, 6, and 7, respectively.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **block_id** parameter contains the offset to the BC Control Block.

The **dma_control** parameter is a pointer to a structure used to initialize the DMA information.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain one of the following:

UNDEFINED_BLOCK	The specified BC Control Block does not exist

See Also

init_1553_device(), open_1553()

set_error_word — Bus Controller Routines

Set the error injection control word for a specified link

Syntax

```
#include "sbs_incl.h"

void set_error_word ( device, link_id, errors )
DEVICE_HANDLE       device;
VALID_LINK_ID       link_id;
ERROR_WORD_TYPE     errors;
```

Description

set_error_word() writes a value to the error injection control (offset 17) of a chain link.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** procedure.

The **errors** parameter is a record representing the error injection control word and is used to inject 1553 errors. See **add_link_to_chain()** for a listing of the components of the record.

See Also

add_link_to_chain(), init_1553_device(), open_1553()

stop_chain Bus Controller Routines

Halts the execution of a bus controller chain program

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN stop_chain (device)
DEVICE_HANDLE device;

Description

stop_chain() halts the execution of a bus controller chain program by issuing host command 4, Stop BC Chain.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

See Also

init_1553_device(), load_chain(), open_1553()

update_bc_control_info Bus Controller Routines

Informs the firmware of global BC changes

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN update_bc_control_info (device)
DEVICE_HANDLE device;

Description

update_bc_control_info() informs the firmware of changes to the BC intermessage gap value (BCIMGP), BC retry information (BCRETRY), maximum allowable status response time (RSPGPA), and Frame Timer (FRMTMR) registers by issuing host command 6, Update Global BC Control Information.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain one of the following:

UPDATE_BC_ERROR	A host command error occurred during processing of host command 6, Update BC Global Control Information
UPDATE_BC_TIMEOUT	A timeout occurred during processing of host command 6, Update BC Global Control Information

See Also

enable_frame_timer(), disable_frame_timer()

write_link_data Bus Controller Routines

Write data to a chain link data buffer using VMEbus long word access

Syntax

#include "sbs_incl.h"

int write_link_data (device, **link_id, buff_id, out_buffer)**
DEVICE_HANDLE device;
VALID_LINK_ID link_id;
VALID_BUFFER_ID buff_id;
DATA_BUFFER out_buffer;

Description

write_link_data() copies user data to a chain link's data buffer using VMEbus long word (32-bit) access. This call may be used to update the data word associated with a mode code.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **link_id** parameter contains the unique identifier that refers to a specific link. The link identifier is established by the **add_link_to_chain()** routine.

The **buff_id** parameter is an address offset pointing to the BC data buffer. The procedure **add_link_to_chain()** creates this offset and returns it to the caller as **chain_link_info.buff_id[n]**.

The **out_buffer** parameter is a data buffer allocated by the caller. This routine will copy all valid data found in this buffer to chain link data buffer. The number of words copied to the buffer is determined from the word count field of the chain link's command word.

Return Value

The function returns the number of words copied from the link's data buffer into the caller's data buffer.

See Also

add_link_to_chain(), init_1553_device(), open_1553(), read_link_data()

create_bsm_buffer — Bus Monitor Routines

Create a sequential monitor buffer

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN create_bsm_buffer ( device, buffer_size )
DEVICE_HANDLE    device;
unsigned_int     buffer_size;
```

Description

create_bsm_buffer() allocates memory on the 1553 device for the sequential monitor buffers.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **buffer_size** parameter contains the length in words of a sequential monitor buffer. Two buffers of this length are created. When one becomes full, the buffers swap, the monitor buffer swap interrupt is generated, and the second buffer begins to fill. The size of the sequential monitor buffer is limited only by the amount of memory available on the 1553 device.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

OUT_OF_MEMORY	Allocation of memory on the 1553 device failed.

See Also

init_1553_device(), open_1553(), force_buffer_swap()

force_buffer_swap — Bus Monitor Routines

Swap the sequential monitor buffers

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN force_buffer_swap ( device )
DEVICE_HANDLE    device;
```

Description

force_buffer_swap() swaps the two sequential monitor buffers. This routine can be used to flush the current sequential monitor buffer and start capturing data in the second, or read the current sequential monitor buffer without waiting for it to fill.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

See Also

create_bsm_buffer(), init_1553_device(), open_1553()

read_bsm_buffer — Bus Monitor Routines

Read the sequential monitor buffer using VMEbus long word access

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN read_bsm_buffer ( device, new_buffer )
DEVICE_HANDLE       device;
MONITOR_BUFFER      new_buffer;
```

Description

read_bsm_buffer() reads the sequential monitor buffer using VMEbus long word (32-bit) access. The buffer not being filled by the firmware is the one read by the routine.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **new_buffer** parameter is a data buffer allocated by the caller. The entire contents of the sequential monitor buffer are copied to this buffer.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_SQM_BUFF	The sequential monitor buffer is not defined
NO_WORDS_IN_SQM_BUFF	There are no words in the sequential monitor buffer

See Also

create_bsm_buffer(), init_1553_device(), open_1553()

seq_mon_all_clear — Bus Monitor Routines

Disables sequential monitoring for all subaddresses for all RTs.

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN seq_mon_all_clear ( device )
DEVICE_HANDLE    device;
```

Description

seq_mon_all_clear() disables sequential monitoring for all subaddresses for all remote terminals by clearing bit 1 of the MONCTRL register and all bits in both SA SM Filter words in the RT Definition Tables of all defined RTs.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**

See Also

init_1553_device(), open_1553()

seq_mon_all_rt_sa Bus Monitor Routines

Enables sequential monitoring for all remote terminals and subaddresses

Syntax

```
#include "sbs_incl.h"

void seq_mon_all_rt_sa ( device )
DEVICE_HANDLE          device;
```

Description

seq_mon_all_rt_sa() enables sequential monitoring for all remote terminals and subaddresses by setting bit 1 of the MONCTRL register.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

See Also

init_1553_device(), open_1553()

seq_mon_all_sa — Bus Monitor Routines

Enables sequential monitoring for all subaddresses for an RT

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN seq_mon_all_sa ( device, rt )
DEVICE_HANDLE    device;
VALID_RT         rt;
```

Description

seq_mon_all_sa() enables sequential monitoring for all subaddresses for a remote terminal by setting all bits in the SA SM Filter word in the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter contains the remote terminal number.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

OUT_OF_MEMORY	There is not enough memory to allocate the RT structure.

See Also

init_1553_device(), open_1553()

seq_mon_clear_all_sa Bus Monitor Routines

Disables sequential monitoring for all subaddresses of an RT

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN seq_mon_clear_all_sa ( device, rt )
DEVICE_HANDLE    device;
VALID_RT         rt;
```

Description

seq_mon_clear_all_sa() disables sequential monitoring for all subaddresses of a remote terminal by clearing all bits in the SA SM Filter word in the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **rt** parameter specifies on which RT to disable sequential monitoring.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.

See Also

init_1553_device(), open_1553()

seq_mon_clear_one_sa — Bus Monitor Routines

Disables sequential monitoring for one subaddress of a remote terminal

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN seq_mon_clear_one_sa ( device, sa )
DEVICE_HANDLE    device;
SA_INFO_TYPE     sa;
```

Description

seq_mon_clear_one_sa() disables sequential monitoring for one subaddress of a remote terminal by clearing the corresponding bit in the SA SM Filter word in the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa** parameter is a structure containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the structure are listed in the description of **define_rt_sa()**.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_RT	The specified RT does not exist.

See Also

init_1553_device(), open_1553()

seq_mon_one_sa — Bus Monitor Routines

Enables sequential monitoring for one subaddress of a remote terminal

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN seq_mon_one_sa ( device, sa )
DEVICE_HANDLE    device;
SA_INFO_TYPE     sa;
```

Description

seq_mon_one_sa() enables sequential monitoring for one subaddress of a remote terminal by setting the corresponding bit in the SA SM Filter word in the RT Definition Table.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sa** parameter is a structure containing information relative to the initialization of the subaddress. This record must be properly initialized by the user before making this call. The components of the structure are listed in the description of **define_rt_sa()**.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

OUT_OF_MEMORY	There is not enough memory to allocate the RT structure.

See Also

init_1553_device(), open_1553()

set_bsm_dma — Bus Monitor Routines

Writes DMA control information to a sequential monitor buffer

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN set_bsm_dma ( device, buffer_id, dma_control )
DEVICE_HANDLE       device;
VALID_BUFFER_ID     buffer_id;
DMA_CONTROL         dma_control;
```

Description

set_bsm_dma() writes DMA Control, ABI Address, VME Address, and DMA word count values in the sequential monitor buffer at offsets 7, 8, 9, and 10, respectively.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **buffer_id** parameter contains the offset to the sequential monitor buffer.

The **dma_control** parameter is a structure used to initialize the DMA information.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

UNDEFINED_BUFFER	The specified sequential monitor buffer does not exist

See Also

init_1553_device(), open_1553()

set_int_on_swap — Bus Monitor Routines

Enables and disables Interrupt on Buffer Swap

Syntax

```
#include "sbs_incl.h"

void set_int_on_swap ( device, sqm_buffer_id, interrupt_flag )
DEVICE_HANDLE       device;
VALID_BUFFER_ID     sqm_buffer_id;
BOLEAN              interrupt_flag;
```

Description

set_int_on_swap() sets or clears the Interrupt on Buffer Swap bit in the Buffer Control Length word of the sequential monitor buffer.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **sqm_buffer_id** parameter contains the offset to the sequential monitor buffer.

The **interrupt_flag** parameter is "0" to clear Interrupt on Buffer Swap and "1" to set the bit.

See Also

create_bsm_buffer()

disable_1553_interrupts Interrupt Management Routines

Disable 1553 device interrupts to the host

Syntax

#include "sbs_incl.h"

void disable_1553_interrupts (device)
DEVICE_HANDLE device;

Description

disable_1553_interrupts() disables the 1553 device from interrupting the host. Interrupts are disabled by clearing bit 3 in the Control Status Register, located at word offset 0.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

See Also

enable_1553_interrupts(), init_1553_device(), open_1553()

enable_1553_interrupts Interrupt Management Routines

Enable 1553 device interrupts to the host

Syntax

#include "sbs_incl.h"

void enable_1553_interrupts (device)
DEVICE_HANDLE device;

Description

enable_1553_interrupts() enables interrupts from the 1553 device to the host. Interrupts are enabled by setting ENA INT, bit 3 in the Control Status Register, located at word offset 0.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

See Also

disable_1553_interrupts(), init_1553_device(), open_1553()

handle_1553_interrupt Interrupt Management Routines

Handle interrupts generated by the 1553 device

Syntax

#include "sbs_incl.h"

void handle_1553_interrupt (device)
DEVICE_HANDLE device;

Description

handle_1553_interrupt() services interrupts generated by the 1553 device. It first determines the cause of the interrupt and then branches to an application specific handler for that interrupt if one exists. Each interrupt is logged into an interrupt queue resident on the host that is used for test purposes. This routine is callable from an Interrupt Service Routine written specifically for the caller's operating system. It may also be called from the application program if polling is used instead of interrupts.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

See Also

init_1553_device(), open_1553()

init_interrupt_queue Interrupt Management Routines

Create and initialize the interrupt queue

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN init_interrupt_queue (device, queue_length)
DEVICE_HANDLE device;
int queue_length;

Description

init_interrupt_queue() creates a double buffered interrupt queue and allocates memory on the 1553 device.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **queue_length** parameter contains the maximum number of interrupts that are to be stored in the interrupt queue. Two circular queues of **queue_length** are allocated. If an interrupt is generated during the execution of the host's interrupt handler, the firmware will log the interrupt into the second interrupt queue.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

Diagnostics

If **FAILURE** is returned, the global variable **error_exception** will contain the following:

OUT_OF_MEMORY Allocation of memory on the 1553 device failed.

See Also

init_1553_device(), open_1553(), handle_1553_interrupt()

bit_1 BIT Management Routines

Executes Built in Test routine 1 - 29K Sanity Test

Syntax

#include "sbs_incl.h"

BIT_RESULT bit_1 (device)
DEVICE_HANDLE device;

Description

bit_1() executes built-in-test routine 1 which performs a read/write test on the I/O bus and memory bus.

Note: The built-in-test must be run from BIT mode. Therefore, if the ABI is in application mode, the firmware must be reloaded using **load_firmware** before the built-in-test can be performed.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

The **bit_result** parameter is a pointer to a **BIT_RESULT_TYPE** structure which contains the BIT results. The components of this structure and their descriptions are:

```
typedef struct bit_status_type
{
     BIT_STATUS_TYPE     status;
     char                message[40];
} BIT_RESULT_TYPE
```

The enumerated type for **BIT_STATUS_TYPE** is the following:

```
typedef     enum {BIT_PASS, BIT_FAIL}  BIT_STATUS_TYPE
```

See Also

init_1553_device(), open_1553()

bit_2 BIT Management Routines

Executes Built in Test routine 2 - I/O Port Test

Syntax

#include "sbs_incl.h"

BIT_RESULT bit_2 (device)
DEVICE_HANDLE device;

Description

bit_2() executes built-in-test routine 2 which performs a read/write test for the I/O bus.

Note: The built-in-test must be run from BIT mode. Therefore, if the ABI is in application mode, the firmware must be reloaded using **load_firmware** before the built-in-test can be performed.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

The **bit_result** parameter is a pointer to a **BIT_RESULT_TYPE** structure which contains the BIT results. See **bit_1()** for a description of this structure type.

See Also

init_1553_device(), open_1553(), **bit_1()**

bit_3 BIT Management Routines

Executes Built in Test routine 3 - Data RAM Test

Syntax

#include "sbs_incl.h"

BIT_RESULT bit_3 (device)
DEVICE_HANDLE device;

Description

bit_3() executes built-in-test routine 3 which performs a complemented data test.

Note: The built-in-test must be run from BIT mode. Therefore, if the ABI is in application mode, the firmware must be reloaded using **load_firmware** before the built-in-test can be performed.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

The **bit_result** parameter is a pointer to a **BIT_RESULT_TYPE** structure which contains the BIT results. See **bit_1()** for a description of this structure type.

See Also

init_1553_device(), open_1553(), bit_1()

bit_4 BIT Management Routines

Executes Built in Test routine 4 - 29K Burst Mode Test

Syntax

```
#include "sbs_incl.h"

BIT_RESULT   bit_4 ( device )
DEVICE_HANDLE    device;
```

Description

bit_4() executes built-in-test routine 4 which performs 29K-specific tests for data bursts.

Note: The built-in-test must be run from BIT mode. Therefore, if the ABI is in application mode, the firmware must be reloaded using **load_firmware** before the built-in-test can be performed.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

The **bit_result** parameter is a pointer to a **BIT_RESULT_TYPE** structure which contains the BIT results. See **bit_1()** for a description of this structure type.

See Also

init_1553_device(), open_1553(), **bit_1()**

bit_5 BIT Management Routines

Executes Built in Test routine 5 - System Timer Test

Syntax

#include "sbs_incl.h"

BIT_RESULT bit_5 (device)
DEVICE_HANDLE device;

Description

bit_5() executes built-in-test routine 5 which performs a verification of the 48-bit system timer, including clock accuracy, counter, interval timing, and local interrupt tests.

Note: The built-in-test must be run from BIT mode. Therefore, if the ABI is in application mode, the firmware must be reloaded using **load_firmware** before the built-in-test can be performed.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

The **bit_result** parameter is a pointer to a **BIT_RESULT_TYPE** structure which contains the BIT results. See **bit_1()** for a description of this structure type.

See Also

init_1553_device(), open_1553(), bit_1()

bit_6 BIT Management Routines

Executes Built in Test routine 6 - DSP/1553B Test

Syntax

```
#include "sbs_incl.h"

BIT_RESULT  bit_6 ( device )
DEVICE_HANDLE    device;
```

Description

bit_6() executes built-in-test routine 6 which performs a short test including a FIFO wrap test, an interrupt test for transmit data, and verification of status flags for the 29K and DSP. **bit_6** also performs a long test which includes the 1553B "walking ones" test in addition to the short test.

Note: The built-in-test must be run from BIT mode. Therefore, if the ABI is in application mode, the firmware must be reloaded using **load_firmware** before the built-in-test can be performed.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

The **bit_result** parameter is a pointer to a **BIT_RESULT_TYPE** structure which contains the BIT results. See **bit_1()** for a description of this structure type.

See Also

init_1553_device(), open_1553(), bit_1()

bit_all BIT Management Routines

Perform all six Built-In-Tests

Syntax

#include "sbs_incl.h"

BIT_RESULT bit_all (device)
DEVICE_HANDLE device;

Description

bit_all() performs all Built-In-Tests, as described on pages ST-6 and ST-7 of the ABI-V5 Reference Manual.

Note: The built-in-test must be run from BIT mode. Therefore, if the ABI is in application mode, the firmware must be reloaded using **load_firmware** before the built-in-test can be performed.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

Return Value

The **bit_result** parameter is a pointer to a **BIT_RESULT_TYPE** structure which contains the BIT results. The components of this structure and their descriptions are:

```
typedef enum {BIT_PASS, BIT_FAIL, NO_STATUS} BIT_STATUS_TYPE;

typedef struct bit_result_type {
     BIT_STATUS_TYPE     status;
     char                message[40];
} BIT_RESULT_TYPE;

typedef BIT_RESULT_TYPE *BIT_RESULT;
```

The return value will contain a status, which will be either **BIT_PASS** or **BIT_FAIL**. Nothing is returned from this function in the **message** field.

See Also

bit_1(), bit_2(), bit_3(), bit_4(), bit_5(), bit_6()

close_device Low Level Routines

Close a device

Syntax

#include "sbs_incl.h"

void close_device (device)
DEVICE_HANDLE device;

Description

close_device() disconnects the process ID from the 1553 device.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** function. It points to a **DEVICE_RECORD** structure.

See Also

open_1553()

get_ram Low Level Routines

Read a value from 1553 device memory

Syntax

#include "sbs_incl.h"

UWORD get_ram (device, offset)
DEVICE_HANDLE device;
ULONG offset;

Description

get_ram() returns a value from 1553 device memory. It has been written as both a macro and a subroutine. The macro call is much more efficient than the subroutine version and should be used where possible. To use the subroutine call, declare the **NO_MACRO** directive.

The **get_ram()** macro is declared in *prototyp.h* as:

```
#define get_ram(device, offset) \
((unsigned short)(*((unsigned short *)device->base_address + offset)))
```

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **offset** parameter is a longword offset to the location in the 1553 device's memory that will be read.

Return Value

A value of the 1553 device's memory located at the word offset is returned.

See Also

get_ram()32, init_1553_device(), open_1553(), put_ram()

get_ram32 Low Level Routines

Read a value from 1553 device memory using VMEbus long word access

Syntax

#include "sbs_incl.h"

ULONG get_ram32 (device, offset)
DEVICE_HANDLE device;
ULONG offset;

Description

get_ram32() returns a long word value from 1553 device memory. It has been written as both a macro and a subroutine. The macro call is much more efficient than the subroutine version and should be used where possible. To use the subroutine call, declare the **NO_MACRO** directive. This routine is the same as **get_ram()** except it takes advantage of the ABI-V5 product's long word access capability.

The **get_ram32()** macro is declared in *prototyp.h* as:

```
#define get_ram32(device, offset) \
((unsigned long)(*((unsigned long *)device->base_address + offset)))
```

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **offset** parameter is a longword offset to the location in the 1553 device's memory that will be read.

Return Value

A long word value of the 1553 device's memory located at the word offset is returned.

See Also

get_ram(), init_1553_device(), open_1553(), put_ram()32

initialize_shared_memory Low Level Routines

Sets a pointer for access to the board

Syntax

#include "sbs_incl.h"

BASENAME * initialize_shared_memory (device_index, device)
int device_index;
DEVICE_HANDLE device;

Description

initialize_shared_memory() obtains a shared memory segment from the operating system and overlays it at the phsical memory location at which the board resides.

Parameters

The **device_index** parameter is the device number. The **device** parameter is the pointer returned from the **open_1553()** function. It points to a **DEVICE_RECORD** structure.

See Also

init_1553_device()

load_ram — Low Level Routines

Perform software download of the data RAM and instruction RAM files

Syntax

```
#include "sbs_incl.h"

EXCEPTION_RETURN load_ram ( device )
DEVICE_HANDLE    device;
```

Description

load_ram() loads the data RAM and 29K instruction RAM areas of the ABI-V5 with data from the download files.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

See Also

open_1553(), init_1553_device()

malloc_1553 Low Level Routines

Allocates space in the board RAM

Syntax

#include "sbs_incl.h"

unsigned long malloc_1553 (device, size)
DEVICE_HANDLE device;
unsigned long size;

Description

malloc_1553() allocates a number of words in the board RAM space for a given device and returns the offset to the beginning of that space.

Parameters

The **device** parameter is the pointer returned from the **open_1553()** function. It points to a **DEVICE_RECORD** structure. The **size** parameter is the number of words to be allocated.

Return Value

A long word offset to the allocated RAM space.

Diagnostics

If an error is detected, the return value will be zero and the global variable **error_exception** will contain one of the following:

MEMORY_ALLOCATION	The device number specified is higher than **MAX_DEV**.
OUT_OF_MEMORY	There is not enough free memory to allocate the requested size.

See Also

define_rt_sa(), init_interrupt_queue(), create_bsm_buffer(), add_link_to_chain()

open_device Low Level Routines

Open a device

Syntax

#include "sbs_incl.h"

EXCEPTION_RETURN open_device (device_index, device)
int device_index;
DEVICE_HANDLE device;

Description

open_device() performs an operating system "open" for the device. **open_devic**e is operating system and device driver dependent.

Parameters

The **device_index** parameter is the device number. The **device** parameter is the pointer returned from the **open_1553()** function. It points to a **DEVICE_RECORD** structure.

Return Value

EXCEPTION_RETURN returns either **SUCCESS** or **FAILURE**.

See Also

open_1553(), init_1553_device()

put_ram — Low Level Routines

Write a value to 1553 device memory

Syntax

#include "sbs_incl.h"

void put_ram (device, offset, value)
DEVICE_HANDLE device;
ULONG offset;
UWORD value;

Description

put_ram() writes a value to 1553 device memory. It has been written as both a macro and a subroutine. The macro call is much more efficient than the subroutine version and should be used where possible. To use the subroutine call, declare the **NO_MACRO** directive.

The **put_ram()** macro is declared in *prototyp.h* as:

```
#define put_ram(device, offset, value) \
 (*((unsigned short *)device->base_address + offset)) = value;
```

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **offset** parameter is a longword offset to the location in memory that will be written.

The **value** parameter is the data word to be written to the 1553 device's memory.

See Also

get_ram(), init_1553_device(), open_1553(), put_ram32()

put_ram32 Low Level Routines

Write a value to 1553 device memory using VMEbus long word access

Syntax

#include "sbs_incl.h"

void put_ram32 (device, offset, value)
DEVICE_HANDLE device;
ULONG offset;
ULONG value;

Description

put_ram32() writes a value to 1553 device memory. It has been written as both a macro and a subroutine. The macro call is much more efficient than the subroutine version and should be used where possible. To use the subroutine call, declare the **NO_MACRO** directive. This routine is the same as **put_ram()** except it takes advantage of the ABI-V5 product's long word access capability.

The **put_ram32()** macro is declared in *prototyp.h* as:

```
#define put_ram32(device, offset, value) \
 (*((unsigned long *)device->base_address + offset)) = value;
```

Parameters

The **device** parameter is the pointer returned from the **open_1553()** or **init_1553_device()** function. It points to a **DEVICE_RECORD** structure.

The **offset** parameter is a longword offset to the location in memory that will be written.

The **value** parameter is a longword to be written to the 1553 device's memory.

See Also

get_ram(), init_1553_device(), open_1553(), put_ram(), **get_ram32()**

SYSTEM SOFTWARE SUPPORT

SBS Technologies provides additional software to support the interface between our 1553 Interface Libraries and a variety of computer systems. We currently offer it for the ABI-V5 product in the C language. The table below lists the C interface software currently available.

ABI C Language Support
System
Encore 90/91,UMAX V
Encore 90/91,UMPX (Available from Encore)
HP-RT
LynxOS
Radisys
Silicon Graphics, IRIX
UNIX V, Release 3
VxWorks

Our support software is system dependent and may include a device driver, interrupt handler, or both. A device driver links user software to the 1553 device and is often required for certain operating systems (especially UNIX). If the operating system does not require a device driver, an interrupt service routine is needed to response to 1553 interrupts. In many cases, SBS Technologies will help write interrupt handler software designed specifically for your operating system or application.

Device Drivers

A device driver ensures exclusive access and handles interrupts generated by the 1553 device. If exclusive access and interrupt handling are not requirements, a device driver is usually not necessary. In a UNIX-like environment, the device driver routines which may be needed are:

close Clears the open flag.

init Initializes the 1553 device at boot time.

intrpt Handles 1553 device interrupts. It can be simple or complex.

ioctl Performs I/O control commands. Not normally needed, but some systems may need one to return the virtual address pointer, set debugging flags, etc.

open Checks an open flag to see if the device has been opened previously, sets the open flag, and returns a file descriptor.

UNIX Device Driver Interrupt Routines. The simplest interrupt routine involves the use of UNIX "signals" where the user's program would set up a signal handler such as SIGUSR1. When the kernel receives an interrupt from the device, the *intrpt* routine in the device driver signals SIGUSR1. The signal handler in the user application is responsible for determining from which device the interrupt came, reading the interrupt data, and clearing the interrupt. Depending on the efficiency of the signal processing and the timing requirements of the application, this may or may not be an suitable solution.

Because of strict timing requirements, complex interrupt handlers were implemented on both a LYNX 2.0 operating system running on a PS/2 and a UNIX V 3.2 operating system running on a Harris Nighthawk. Part of the design was to create an *ioctl* call in the kernel which communicates with the *intrpt* routine via a semaphore and a circular queue of device interrupt data. When an interrupt is received, the *intrpt* routine determines who is the interrupting device, acknowledges the interrupt, disables device interrupts, places the device information in the circular queue, and "signals" the *ioctl* routine via the semaphore. The *ioctl* call, when invoked by a user application call, waits for the semaphore. When the semaphore is signaled from *intrpt*, the *ioctl* determines the device from the circular queue information, reads the interrupt data from the device, moves it to the argument data which is returned to the user application, enables the device interrupts, and returns to the user application.

It is important to note that the when the user application calls *ioctl* it will block until an interrupt is received. This wait can be avoided by placing a time-out in the *ioctl* routine and returning an error to the calling routine so it can do some other processing before returning to the *ioctl* call.

Encore Computer System

A device driver is needed for the Encore computer systems to map shared memory and handle interrupts. One is supplied with our software distribution tape which is found in the directory *SBS/drivers/encore*. This driver was written and documented for the UMAX operating system by Encore Computer. Below are the installation procedures to build the device driver. The original installation notes are found in the file named *SBS/drivers/encore/package/ins_abi_1.0*, and these notes are found in the file named *SBS/drivers/encore/readme*. Note that system responses and file names are in *italics* type and user responses are **bold** in type. Follow each user response with a carriage return or enter keystroke.

Introduction

These installation notes are for the person responsible for installing the Encore 90/91 Family ABI driver for the SBS Technologies 1553 VMEbus Interface. They provide information that is not included in other documents in the set. Read this document and the Encore 90/91 Family ABI/ASF Release Notes before installing the 1553 driver.

Contents of this Document. These notes contain the following sections:

- Introduction
- Contents of Document
- Hardware/Software Requirements
- Installation Procedures
- Media Contents
- Distribution Tape Format

Hardware/Software Requirements. The following section contains the minimum hardware and software requirements for installation of the ABI driver.

Hardware Requirements:

- One compatible device for the distribution medium:
  TSR-04 9-track tape drive or a TSC-02 cartridge tape drive

- One compatible auxiliary input device for batch processing:
  UMAX V terminal

- One Encore Series 90 computer

Software Requirements:

- UMAX V Operating System Revision 3.0.5 or later

Installation Procedures

This section provides procedures for installing the 1553 device driver. You must be running UMAX V Revision 3.0.5 or later to install this software package. Before you begin installing the 1553 device driver you must be logged in with superuser privilege. Installation consists of the following phases:

1. Installation of the software onto the target system
2. Reconfiguration of the UNIX kernel to include the 1553 device driver
3. Building the reconfigured kernel

Load the contents of the tape on your system as follows:

1. Place the distribution tape in the Series 90/91 tape drive.
2. Enter the following commands to load the contents of the tape and to sync your disks. If the */usr/abi* directory does not exist, first create it using the mkdir command. The name of the 0h file is system dependent. You must specify the appropriate entry for your system. The *add_abi* script prompts for the directories in which to install the remaining files from the tape. The default directories are recommended.

   # cd /usr/abi
   # tar -xvf /dev/rmt/0h
   # ./add_abi
   # sync; sync; sync

3. Verify that the */etc/master* file contains the following abi device (located below the other SCSI,enet, and VMEbus devices). If it does not, edit the */etc/master* file to include the devices (a copy of the devices is included in /usr/abi/bin/master.abi).

```
#-------------------------------------------------------------------------
# 1553 SBS VME board
#-------------------
abi   vme  soch  cMC  abi   0   39     2    2
#      ^   ^^^^  ^^^   ^    ^    ^     ^    ^
#      :   ::::  :::   :    :    :     :    :.. max devices
#      :   ::::  :::   :    :    :     :        per channel
#      :   ::::  :::   :    :    :     :
#      :   ::::  :::   :    :    :     :.. max channel
#      :   ::::  :::   :    :    :
#      :   ::::  :::   :    :    :.. character device number
#      :   ::::  :::   :    :        (major number)
#      :   ::::  :::   :    :
#      :   ::::  :::   :    :.. block device number (not used)
#      :   ::::  :::   :
```

```
#      :   ::::  :::   :.. driver name prefix
#      :   ::::  :::
#      :   ::::  :::.. (C) generate 'int abi_cnt = 1;' in conf.c
#      :   ::::  ::        (number derived from units)
#      :   ::::  ::
#      :   ::::  ::... (M) generate 'int abi_cmajor = 39;' in conf.c file
#      :   ::::  :
#      :   ::::  :.... (c) character device flag
#      :   ::::
#      :   :::..... (h) = generate #include "sys/abispace.h" in conf.c
#      :   :::..... (c) = driver has 'abiclose' function
#      :   ::....... (o) = driver has 'abiopen'  function
#      :   :........ (s) = driver has 'abiinit'  function
#      :
#      :.. device type
#-------------------------------------------------------------------------
```

4. Edit the */etc/system* file to include the devices contained in the file */usr/abi/bin/system.abi.* The first entry belongs below the "enet" entry. Note that the starting address on the first 1553 device is set to 0xBF000000 (VMEbus A32) and the STATUSID vector is 0xFF00. The 1553 board is jumpered to match these values. If they do not, change this entry to match the board's jumpers.

```
#name         minor#   channel
 enet         0         0
```

```
#-------------------------------------------------------------------------
# 1553 SBS 6U VMEbus board
#-----------------------
#name   index#  board   parameter 0     parameter 1    parameter 2
 abi    0       0       0xbf00          0xff00         0x0
#       ^       ^       ^               ^              (not used)
#       :       :       :               :
#       :       :       :               :.. board's STATUSID vector
#       :       :       :
#       :       :       :.. VMEbus a32 address (not really used)
#       :       :           The most significant 16 bits of the address
#       :       :           are given. This is due to a problem with
#       :       :           config treating these are signed parameters.
#       :       :
#       :       :.. This field is reflected in abi_devaddr[].v_chan
#       :           structure.
#       :
#       :.. This field generates the '#define ABI 1' in conf.c that
#           is used during driver compilation to generate the actual
#           number of abi board structures.
#-------------------------------------------------------------------------
```

5. The shell script *MAKEDEV.abi* creates all of the device entries needed to access the 1553 device driver. This script requires one or more ABI board specifiers of the form **abi?** where **?** is an 1553 board indicator from **0** to **3**. The following example will create a device entry for two 1553 boards.

/bin/sh -x /usr/abi/bin/MAKEDEV.abi abi0 abi1

6. Back up the existing version of UNIX in the root partition (/ directory) by entering the following commands:

   Note: The *cp* command overwrites the *unix.bak* file if it already exists. If you wish to maintain this file, back up the original *unix.bak* file to another file name before performing the following commands or back up the UNIX file to another file name instead of performing the following commands.

   # cd /
   # cp unix unix.bak

7. Change directories to /usr/src/uts/GEMINI by entering the following command:

   # cd /usr/src/uts/GEMINI

8. Edit the Makefile. The variable *LIBLIST* must be changed to include libabi.a. This entry belongs directly before *libio.a*.

9. Execute the Makefile in */usr/src/uts/GEMINI*

   # make

10. Enter the following command to move the *UNIX*, just created, to */unix*:

    # mv UNIX._ /unix

    where **UNIX._** is the name of the new *UNIX* file created by executing the Makefile.

11. Reboot the new kernel by executing **sysadm reboot** or ***/etc/shutdown***.

12. As the system reboots, the following message indicating that an ABI was mapped into Series 90/91 address space is displayed on the console:

    NOTICE: ABI Board 0: interrupt vector {vector}.

    The message may vary depending on the number of 1553 boards configured in your system.

Media Contents

This tape contains the following tar images:

Image 1 — Installation Notes file, the Release Notes file (and any late-breaking information), and the script add_abi, which automatically extracts and installs the files from the installation package.

Image 2 Binary product files for support of this product.

Image 3 ABI driver library.

Image 4 ABI test and library source files.

Image 5 Source files.

Distribution Tape Format. The following is a list of files and directories contained in this installation package. To avoid overwriting any needed files, review this list before extracting the files:

NOTE: The driver was originally tested with the ABI library source code located in Image 4. A newer ABI library is found in *SBS/abi/c_src*; however, this library was never tested with this driver. We recommend that you use this one because there have been several useful changes and bug fixes.

Tar Image 1 Files	Recommended Directory	Contents
./SRN_abi_1.0	/usr/abi	Software Release Notes
./add_abi	/usr/abi	Script to install this product
./ins_abi_1.0	/usr/abi	Installation Notes

Tar Image 2 Files	Recommended Directory	Contents
./MAKEDEV.abi	/usr/abi/bin	Script to make ABI devices
./master.abi	/usr/abi/bin	Example file with ABI device entries
./system.abi	/usr/abi/bin	Example file with ABI device entries

Tar Image 3 Files	Recommended Directory	Contents
./libabi.a	/usr/src/uts/GEMINI	ABI driver object

Tar Image 4 Files	Recommended Directory	Contents
./test/Makefile	/usr/abi/test	Builds unit_test
./test/README	/usr/abi/test	Notes on unit_test
./test/{files}	/usr/abi/test	source files

Tar Image 5 Files	Recommended Directory	Contents
./src/abi.c	/usr/abi/src	Driver source
./src/abi..h	/usr/abi/src	Driver include (structures, macros, constants)
./src/abispace.h	/usr/abi/src	Driver controller structures

HP-RT Device Driver

HP-RT Device Driver Files

The following files necessary for building and installing the HP-RT device drivers and SBS libraries are found under *SBS/drivers/HP-RT/*:

make_sbs	make file for SBS libraries
makedrvr	make file for interrupt driver and interrupt driver test programs
config/sbs_abi.cfg	kernel configuration file
include/sbs_abi.h	header file for device driver
include/sbs_info.h	header file for device driver
obj/	temporary directory for object files
src/sbs_abi.c	C source code for device driver
src/sbs_info.c	C source code for device driver
tests/test_int.c	C source code for device driver test program
tests/gen_it.c	C source code for generating and interrupt

Building The Device Drivers

The files contained under *SBS/drivers/HP-RT/src*, *SBS/drivers/HP-RT/include*, and *SBS/drivers/HP-RT/tests* contain the source and include files necessary for building the device drivers and sample test programs for testing the device drivers. A make file is provided (*SBS/drivers/HP-RT/makedrvr*) to build the drivers, copy the object files to the correct system directories, and build the test programs. Run this make file prior to modifying the kernel configuration.

Kernel Memory And Device Driver Declarations

Before installation of an ABI in the system, memory for the ABI must be declared in the kernel. The file *SBS/drivers/HP-RT/config/sbs_abi.cfg* contains samples of device driver and memory allocation information. The file *sbs_abi.cfg* contains device driver and memory allocation information for a single ABI and one interrupt driver.

1. Edit */HP-RT/etc/conf/cfg/CONFIG.TBL*

2. Include the following lines at the end of the *CONFIG.TBL* file:

```
# SBS 1553 ABI cards
I:sbs_abi.cfg
```

3. This will include the information contained in *sbs_abi.cfg* in the *CONFIG.TBL* file. Alternatively, the information contained in *sbs_abi.cfg* may be included directly in *CONFIG.TBL*.

4. Rebuild the kernel and reboot.

5. The device driver test programs found under *SBS/drivers/HP-RT/test* may now be run to verify correct installation of the device drivers.

LYNXOS

A device driver is needed for LynxOS to handle interrupts and to map virtual memory. One is supplied with our software distribution tape or diskette which is found in the directory *SBS/drivers/lynx*. This driver was tested with version 2.1. Below are the installation procedures to build the device driver. These installation notes are also found in the file named *SBS/drivers/lynx/readme*.

Distribution Files

The LynxOS driver is composed of the following files:

makeabi	makefile that builds the ABI library.
makeasf	makefile that builds the ASF library.
makefile	compiles the driver source code in *sbsdrvr.c*, *sbsinfo.c*, and creates a test program from *testdrvr.c*.
mk1553dr	device driver install script.
readme	a text file of these notes.
sbsdrvr.c	device driver source code.
sbsdrvr.h	data definitions used by the device driver.
sbsinfo.c	variable declarations needed when the system is being configured.
testdrvr.c	a test program used to test the driver.
1553.cfg	a system file containing device information.

Driver Installation Procedures

First install the 1553 board into your system. Hardware installation instructions are described in your hardware manual. Follow the steps described below to install the device driver into the LynxOS system. You must have superuser privileges to install the driver.

1. Retrieve the software from the tape or disk. Change directory to where you want to locate the software then extract it using *tar*. For example *tar xvf /dev/rDevice_Name.*

2. Modify the CONFIG.TBL file located in /sys/lynx.os. Add the following line:

```
I:1553.cfg
```

3. Set the address and interrupt parameters for each board. Edit the file *sbsdrvr.h* and choose hexadecimal values for the directives corresponding to the board's base address, I/O address, and interrupt vector. For example if your system has one 1553 board, modify the following directives:

```
#define BA_ADDR0 xxxxxx
#define IO_ADDR0 xxx
#define INT_VECT0 xxx
```

4. Select values to match the I/O and base addresses and interrupt vector jumpered on the 1553 board. Note that the IO_ADDRx field does not apply to VMEbus based boards; set this value to 0 if you have one. If you have multiple boards, modify the directives associated with boards two, three, and so on to match your system configuration.

   Modify MAXDEVS to match the number of boards that will be in your system. It is set to 1 by default.

4. For multi-board systems, you must also edit the file *sbsinfo.c*. If you have just one board, skip to step 5. An array of structures named `sbsinfo` in *sbsinfo.c* must be initialized for each board in the system. Uncomment `sbsinfo` for each board. Add additional lines if you have more than five boards.

5. Build the kernel. Run the script file to install the 1553 driver.

```
mk1553dr
```

6. Reboot the system.

```
reboot -aN
```

7. Run the 1553 driver test. Type **testdrvr**. The output of the test will be displayed on the screen and should look like the following.

```
Opening driver.
  In sbsopen()
  Base addr: xxxxxxxx Level: zz
  Board 0 found.
File Descriptor : y
Mapped Base Address: xxxxxx
  In sbsioctl() - minor dev: 0
1553 Probe Test Pass
Begin WaitForIntr()
  In sbsioctl() - minor dev: 0
  In sbsioctl() - timeout set
T2 done
T1 done
Interrupt Count = 1
Closing driver.
```

```
In sbsclose()
```

The addresses displayed above (xxxxxx) will corresponds to a virtual address, the file descriptor (y) should be an integer, and the interrupt vector (zz) should correspond to the vector jumpered on the board. If the test fails, the message `1553 Probe Test Fail` will be displayed. If this occurs, the jumpered address on the board does not match the address in the driver file System. Double check these values.

If you wish to suppress annoying screen message such as *In sbsopen()* that are generated by the driver, edit the makefile, remove the **-DDEBUG** flag, and recompile the driver.

7. Compile the 1553 library. The library is located in the *./SBS/abi* or *./SBS/asf* source code directories.

Interrupt Operation

Device driver interrupts can be set up and handled using the `ioctl` system call. The first parameter to `ioctl` should be the file descriptor returned from the `open` call. The second parameter must be one of the following request codes:

INTR_INSTALL	Installs the interrupt handler.
INTR_WAIT	Waits for an interrupt. When the interrupt occurs, the user's interrupt handler is called.

The third parameter to the `ioctl` call depends upon the request code. For INTR_INSTALL, this parameter is not used, but a pointer to a character must be passed anyway. For INTR_WAIT, the third parameter should point to a structure of type `struct intr_entry` defined in *sbsdrvr.h.*

```
struct intr_entry
{
   void (* handle_1553_intr)();
   int  intr_timeout;
   char *arg;
}
```

This structure must be initialized by the caller. The element `handle_1553_intr` is a pointer to a function that will be executed with one argument, `arg`, when an interrupt occurs. The `ioctl` will wait `intr_timeout` hundredths of a second. A value of zero will cause the `ioctl` to wait indefinitely. Review the procedure in *testdrvr.c* for further details.

C Directives

To compile the ABI library, declare the following directives for your compiler and board type. They may be defined in the makefile or in *system.h*. See the Installation section for further details.

```
 /*  LYNXOS for 386/486 or MVME167 Platforms  */
#define  LYNX
#define  NORMAL
#define  BITFIELD_1

/*  You must declare your board type  */
#define ABI
```

Radisys

The ABI-V3 libraries have been tested using a Radisys Single Board Computer in a VMEbus system. Access to the 1553 board was through the Radisys BusManager driver. No special driver files are available.

C Directives

To compile the ABI library, declare the following directives for your compiler and board type. They may be defined in the makefile or in *system.h*. See the Installation section for further details.

```
/*  RADISYS VMEbus board and Borland C Compiler  */
#define  RADISYS
#define  BORLAND
#define  DOSREAL

/* You must declare your board type  */
#define ABI
```

Silicon Graphics Computer System

A device driver is needed for the Silicon Graphics computer systems to map shared memory and handle interrupts. A driver is supplied for systems running versions of IRIX 5.3. The driver software is found in the directory *SBS/drivers/irix/abiv5*. Below are the installation procedures to build the driver for your system. Note that system responses and file names are in *italics* type and user responses are **bold** in type. Follow each user response with a carriage return or enter keystroke.

NOTE: If you have trouble reading the distribution tape, try reading it without swapping the bytes. For example: *tar -xvf /dev/tape/ns*.

Distribution Files

The Silicon Graphics ABI-V5 driver is composed of the following files:

makefile	makefile to build the driver.
sbs1553	system file specifying driver information such as the major device number.
sbs1553.c	device driver source code.
sbs1553.o	device driver object file.
sbs1553.sm	system file containing hardware information for the device.
testabi.c	driver test file source code.

IRIX 5.x Installation

This driver will allow the user to map the ABI device memory into the user's address space and receive interrupts via the *ioctl()* interface (see *testabi.c*).

This driver assumes that you use the factory default VMEbus address (0x48000000), IPL (5), and interrupt vector (0xF0) for the first board. If it is necessary to change the base address and interrupt vector on succeeding boards you must edit the file *sbs1553.sm* and change these parameters to reflect your configuration.

After you have installed the 1553 board in the SGI (refer to the "Hardware Installation" section of your 1553B Reference Manual), use the following steps to install the driver:

1. login as root
2. This will make 4 nodes /dev/1553[A-D]

make node

3. Compile the driver code

   make sbs1553.o

4. Save original kernel. This will insure that the old version of unix is there in case things go wrong.

   ln /unix /unix.old

5. Copy everything to its proper place.

   make install

6. Make a new unix kernel.

   make newunix

   You should see that the *lboot(1M)* found at least one of your ABI boards in the system If not, you need to check the VME addresses.

7. Reboot the system.

   reboot

8. Test THE board.

   make TestAbi

C Directives

To compile the ABI library, declare the following directives for your compiler and board type. They may be defined in the makefile or in *system.h*. See the Installation section for further details.

```
/*  IRIX for Silicon Graphics   */
#define IRIX5_0         /*  IRIX 5.x  */

#define  NORMAL
#define  BITFIELD_1

/* You must declare your board type  */
#define ABI
```

UNIX System V

The C 1553 library was ported to a Motorola Delta system which ran UNIX System V, Release 3. For this system, a device driver was not needed to map the 1553 device memory to shared memory. A device driver is needed to handle interrupts but is not available at the moment. Thus, there are not any special driver files for this system. Makefiles for the ABI and ASF, named *makeabi* and *makeasf* respectively, are found in the directory *SBS/drivers/unix5_0*.

C Directives

To compile the ABI library, declare the following directives for your compiler and board type. They may be defined in the makefile or in *system.h*. See the Installation section for further details.

```
/*  UNIX System V system  */
#define  UNIXSYSVR3
#define  NORMAL
#define  BITFIELD_1

/* You must declare your board type   */
#define ABI
```

VxWorks

The section describes the VxWorks example driver that handles 1553 interrupts generated by an ABI or ASF board. This software was originally developed on a Sun 4 using OS version 4.1. The target system included an ABI-V and a Motorola MVME147 CPU which booted VxWorks from EEPROM. These system dependencies are reflected in the source code. User modifications will be required to run VxWorks on other system configurations. Note that system responses and file names are in *italics* type and user responses are **bold** in type. Follow each user response with a carriage return or enter keystroke.

Distribution Files

The VxWorks example driver is composed of the following files found in *SBS/drivers/vxworks*:

makeabi a makefile that compiles and links the ABI 1553 Interface Library and VxWorks software. At this point, VxWorks calls are undefined external references. These references are dynamically resolved during the download to the target.

makeasf a makefile that compiles and links the ASF 1553 Interface Library and VxWorks software.

readme a text file of these notes.

sbstsk.c a collection of tasks used to manage the 1553 interrupt handler. It interfaces to the VxWorks library.

swtypes.h data type definitions used by the software routines in the file named *sbstsk.c*.

Driver Routines

There are three VxWorks driver routines found in the file named sbstask.c.

sbs_taski() spawns the interrupt handling task *sbs_task()* and creates a semaphore connecting the 1553 interrupt to *sbs_task()*. Interrupts are enabled on the CPU for IRQ6*, vector number F0h. The directives *CINT_VEC_NUM* and *CINT_LVL* define the 1553 vector and interrupt level. The values of these two directives must match the hardware settings on the 1553 board.

sbs_taskd() disables the 1553 interrupt level on the CPU and deletes *sbs_task()* and its semaphore.

sbs_task() waits for the semaphore that will be generated for each 1553 interrupt. When the semaphore arrives, it calls the Library routine

handle_1553_interrupt(), increments a test counter, then waits for the next semaphore. The VMEbus base address of the 1553 board is passed to *handle_1553_interrupt()*. This address was hardcoded (very, very bad practice!) in *sbs_task()*. This address must be modified by the user.

Library Routines

One Library routine is used with the VxWorks driver.

handle_1553_interrupt() services interrupts on the 1553 device. Application specific 1553 interrupt handlers can be installed inside this routine. This routine can be called by VxWorks or by the application program if polling is used instead of interrupts. It is located in the file named *int_mgmt.c*.

Porting Notes

There is a definition block in the file *makevx* that is system dependent. The definitions CC, AS, and LD were set to values specific to the GNU C compiler. Also the *-traditional* flag set for CFLAGS is GNU specific. The MV147, ALL, CCP, CAT, and RM definitions may be deleted. VXINCL contains the path where VxWorks header files are located. SWINCL contains the path where user header files are located. The path names associated with these definitions are dependent upon the file structure of the development system. The appropriate target header files must be included in *sbstsk.c* that match the system configuration. The names of these files may vary between systems.

The directive VXWORKS and CPU must be defined in the library header file named *system.h* or installed in the makefile. Other directives may be needed. For this system, the following were declared:

```
/*  ABI-V in a VXWORKS environment running on a MVME-147 CPU board. */
#define  VXWORKS
#define  CPU   MC68020
#define  NORMAL
#define  BITFIELD_1
```

These directives may be declared in a *makefile* as well. The Installation Section of this manual contains more information regarding the installation of the libraries.

Installation

1. Copy the *makeabi* file to *makefile.*

2. Modify the software in the files *makefile*, *sbstsk.c*, and *system.h* to account for your system configuration.

3. Compile and link the software. Type **make**. The output file will be named *unit_test*.

4. From the VxWorks prompt on the target CPU, connect to the host and download *unit_test*.

5. Initialize the software by typing: *vxworks_prompt>* **sbs_taski**

6. Run *unit_test*: *vxworks_prompt>* **main**
   NOTE: If there are other *main* programs, you may need to change the name of the main procedure in the file *unit_test* from *main* to *unit_test*.

7. Use *unit_test* to set up the 1553 board and begin bus operations. If interrupts are enabled and an interrupt condition occurs, *sbs_task()* will manage it. Type option *'Handle 1553 Interrupt Queue'* from the Device Management Menu to view the interrupt log.

8. After exiting *unit_test* delete *sbs_task()* by typing: *vxworks_prompt>* **sbs_taskd**

Execution

The library routine *handle_1553_interrupt()* is called by the task *sbs_task()* whenever the 1553 board generates an interrupt. *Handle_1553_interrupt()* manages the 1553 board's interrupt structure and copies the interrupt entries from the 1553 board to a host interrupt log. The log is used for testing purposes and would not be necessary for an application program. The program UNIT_TEST is used to initialize the 1553 board and display interrupt information.

UNIT TEST APPLICATION

The Unit Test application uses the 1553 library to provide basic 1553 bus operations and the ability to exercise the library subprograms. The 1553 library requires some configuration and tuning which is dependent on the target hardware and operating system. Since the Unit Test application is a level above the library, configuring and tuning should not be required.

Unit Test uses ANSI escape sequences to control output to the screen. You must have an ANSI compatible terminal or driver. For PC systems, check your *config.sys* file and make sure the *ansi.sys* driver is installed. There should be a line that reads: **DEVICE=C:\DOS\ANSI.SYS**. Add this line if it is not present and reboot your computer.

The Unit Test application source code must be compiled and linked with the 1553 library.

The names of the files for the C 1553 library are listed in Table B-1.

File Name	Description
unit_tes.c	Main procedure of the Unit test Application.
dev_test.c	Procedure that tests the library subprograms in dev_mgmt.c.
rt_test.c	Procedure that tests the library subprograms in rt_mgmt.c.
bc_test.c	Procedure that tests the library subprograms in bc_mgm.c.
bm_test.c	Procedure that tests the library subprograms in bm_mgmt.c.

Table B-1. C Unit Test Files

A makefile is included with the C Library. Some tuning to this file may be necessary depending upon your software development environment. The makefile will compile all source files, link the object files, and create an executable filed named *unit_test*. See the section *Installation* for a discussion on makefiles.

The names of the files for the Ada 1553 library are listed in Table B-2.

File Name	Description
unit_tes.ada	Main procedure of the Unit test Application.
dev_test.ada	Procedure that tests the library subprograms in dev_pkg.ada.
rt_test.ada	Procedure that tests the library subprograms in rt_pkg.ada.
bc_test.ada	Procedure that tests the library subprograms in bc_pkg.ada.
bm_test.ada	Procedure that tests the library subprograms in bm_pkg.ada.

Table B-2. Ada Unit Test Files

An Alsyis ActivAda invoke file, UNITTEST.INV, is included with the Ada Library. The makefile will compile all source files, bind the object files, and create an executable file named *unit_tes*.

Using The ABI Unit Test Application

To run *unit_tes*t, type **unit_test<CR>** at your prompt (this command depends upon the operating system, e.g., in VMS the command is *run unit_test*). The main menu of *unit_tes*t, illustrated in Figure B-1, should appear on your screen. *Unit_test* is divided into four sections: 1) Device Management, 2) Remote Terminal Management, 3) Bus Controller Management, and 4) Bus Monitor Management. Each of the four selections will branch to another menu offering choices for the management of a particular function.

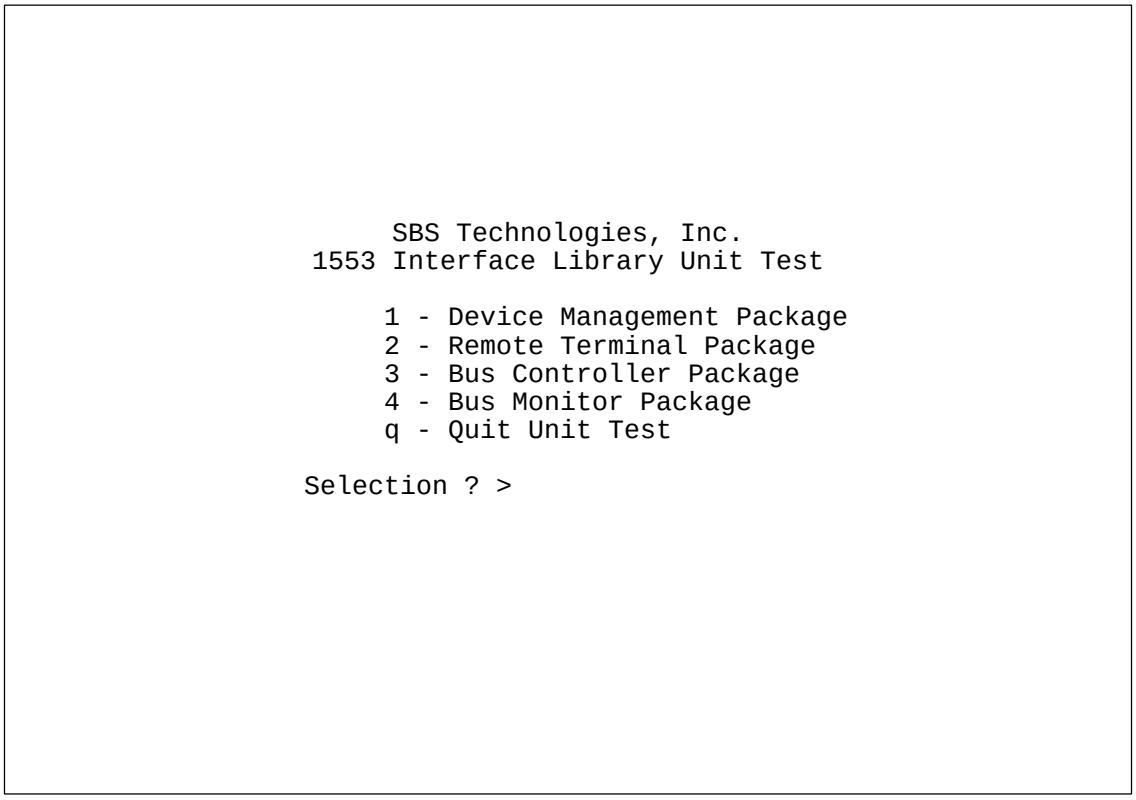


Figure B-1. Unit Test Main Menu

Device Management. Type **1<CR>** at the main menu prompt to branch to the Device Management menu illustrated in Figure B-2. Device Management provides several general routines that initialize and set up the board.

```
                    Device Management Unit Test

     1 - Initialize 1553 Device     a - Initialize Interrupt Queue
     2 - Open 1553 Device           b - Disable 1553 Interrupts
     3 - Close 1553 Device          c - Enable 1553 Interrupts
     4 - Load Firmware              d - Handle 1553 Interrupt Queue
     5 - Start Application          e - Get Device Information
     6 - Initialize Channel         f - Execute Built in Test
     7 - Start 1553 I/O             r - Read/Write RAM
     8 - Stop 1553 I/O              q - Return to Main Menu

--------------------------------------------------------------------------
Selection: Initialize 1553 Device
Device Number ?            >1
Queue Length ?             >4
Seqential Monitor Length ? >1000




Messages:
--------------------------------------------------------------------------
Menu Selection ? > 1
```

Figure B-2. ABI Device Management Menu

A description of each choice is listed below:

1 - Initialize 1553 Device	Performs a number of initialization functions all of which can be called separately if desired. The functions called are *open_1553(), load_firmware(), start_firmware(), initialize_channel(), init_interrupt_queue()*, and *create_bsm_buffer()*.
2 - Open 1553 Device	Opens the device and attaches a shared memory segment for access to the board if needed.
3 - Close 1553 Device	Closes access to the device.
4 - Load Firmware	Loads the data RAM and instruction RAM files.
5 - Start Application	Starts the application code.
6 - Initialize Channel	Initializes the selected channel.
7 - Start 1553 I/O	Starts operations on the 1553 bus.
8 - Stop 1553 I/O	Stops operations on the 1553 bus.

a - Initialize Interrupt Queue	Allocates memory and sets the number of entries in the interrupt queues.
b - Disable Device Interrupts	Disables 1553 device interrupts to the host.
c - Enable Device Interrupts	Enables 1553 device interrupts to the host.
d- Handle 1553 Interrupt Queue	Retrieves the interrupt queue entries.
e - Get Device Information	Returns 1553 device parameters such as the virtual base address, the type of device, the size of the onboard RAM, and the offset to free memory.
f - Execute Built in Test	Runs diagnostic tests one through seven.
r - Read/Write RAM	Displays a 128 word block of RAM on the 1553 device beginning at a user supplied offset. Modifying memory is possible. Figure B-3 illustrates a 128-word block of RAM beginning at offset 80h.
q - Return to Main Menu	Returns the user to the main menu (Figure B-1).

If an error occurs, a message will be displayed at the *Message:* prompt. The selection chosen by the user is displayed at the *Selection:* prompt. The steps below described how to initialize the ABI board. Program responses are in *italics* type, user responses are in **bold** type, and a <CR> symbol represents a carriage return or enter keystroke

Initializing the Device. Type **1<CR>** at the Device Management menu to execute a full device initialization procedure. This routine will prompt the user for the information below. Figure B-2 illlustrates this sequence.

Device Number ? **>1<CR>**

The device number corresponds to the board number you wish to open.

Interrupt Queue Length ? >**4<CR>**
Sequential Monitor Length ? >**1000<CR>**

The interrupt queue length and sequential buffer length are user selectable and their values will depend upon the application at hand. If you do not plan to used the sequential monitor, you may set its length to zero. If you do not plan to use the interrupt queue, you may set its length to zero.

```
Word Offset (hex)? >80
0080: 00000000 00000000 00000000 00000610 00000620 00000000 00000000 00000004
0088: 00000430 00000470 00000490 00000000 00000630 00000636 00000006 00000000
0090: 00000000 00000000 00000000 00000000 00000000 000004d0 000004f0 00000530
0098: 00000550 00000570 00000590 00000000 000005b0 00000000 000005d0 00000000
00a0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00a8: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00b0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00b8: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00c0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00c8: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00d0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00d8: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00e0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00e8: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00f0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00f8: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
'm'=More, 'w'=Write Memory, 'b'=New Base Offset, or 'q'=Quit
```

Figure B-3. Read/Write RAM Page

Since *unit_test* is designed for an environment requiring a device driver, the device(s) must be opened before access via the library will be granted. Each 1553 device (or board) in the system will be assigned a unique device number when the device is opened. Device numbers begin at 1. To access channel 2 for a particular board, an even device number must be assigned. Therefore, each new device will begin with an odd number. For example, if there is only one board in the system, the device numbers assigned will be 1 and 2. If there are two boards in the system, the device numbers will be 1, 2, 3, and 4. The device number will be requested by all other operations invoked by the user.

Initialize 1553 Device (choice 1) performs a number of initialization functions all of which can be called separately if desired. The functions called are Open Device (choice 2), Load Firmware (choice 4), Start Application (choice 5), Initialize Channel (choice 6), Initialize Interrupt Queue (choice a), and Create Sequential Monitor Buffer (choice 1 from the Bus Sequential Monitor Management menu). Each time *unit_test* is run, the device(s) must be opened before proceeding with any other command.

Remote Terminal Management. Type **2<CR>** at the main menu prompt to branch to the Remote Terminal Management menu illustrated in Figure B-4. Remote Terminal Management provides routines to initialize and allocate the subaddress data structures of a remote terminal.

```
                Remote Terminal Management Unit Test

1  -  Define RT Subaddress            8  -  Read Sync Data Word
2  -  Enable Mode Code Interrupt
3  -  Load Status Response Word       a  -  Set BIT Word Table
4  -  Read Filter Word                b  -  Set Transmit Vector Word
5  -  Set Filter Word                 c  -  Set Global Control
6  -  Read from Subaddress Buffer     r  -  Read/Write RAM
7  -  Write to Subaddress Buffer      q  -  Return to Main Menu

------------------------------------------------------------------------
Selection: Define RT Subaddress
Device Number ?            >1
Remote Terminal?           >1
Subaddress ?               >4
'r'= receive or 't'= transmit
Message Type ?             >r
Number of Buffers ?        >2
Monitor Filter (0 or 1)  ? >0

Messages:
------------------------------------------------------------------------
Menu Selection ? > 1
```

Figure B-4. ABI Remote Terminal Management Menu

A description of each routine is listed below:

1 - Define RT Subaddress	Initialize and allocate data structures and buffers for a specified remote terminal and subaddress. Multiple buffers can be allocated per subaddress.
2 - Enable Mode Code Interrupt	Set the mode command interrupt mask to selectively enable interrupts to the host for certain mode codes.
3 - Load Status Response Word	Set a value for a specified remote terminal's status response word.
4 - Read Filter Word	Reads the filter, interrupt, and swap word.
5 - Set Filter Word	Sets the filter, interrupt, and swap word.
6 - Read from Subaddress Buffer	Read data from a subaddress data buffer.
7 - Write to Subaddress Buffer	Write data to a subaddress data buffer.
8 - Read Sync Data Word	Reads the data word associated with Last Synchronize with data word mode code (mode code 17).

a - Set BIT Word Table	Set the data word for the Transmit BIT Word mode code (mode code 19).
b - Set Transmit Vector Word	Set the data word for the Transmit Vector Word mode code (mode code 16).
c - Set Protocol Table	Write a value to the Protocol Table for a specified remote terminal.
r - Read/Write RAM	Displays a 128 word block of RAM on the 1553 device beginning at a user supplied offset. Modifying memory is possible. Figure B-3 illustrates a 128-word block of RAM beginning at offset 80h.
q - Return to Main Menu	Returns the user to the main menu (Figure B-1).

If an error occurs, a message will be displayed at the *Message:* prompt. The selection chosen by the user is displayed at the *Selection:* prompt. The steps below described how to initialize a remote terminal subaddress on the ABI board. Program responses are in *italics* type, user responses are in **bold** type, and a <CR> symbol represents a carriage return or enter keystroke.

Setting up Remote Terminals. Type **1<CR>** at the Remote Terminal Management menu prompt to define a remote terminal subaddress. This call must be made for each subaddres you want to activate. The routine will prompt the user for the information below. Figure B-4 illustrates this sequence.

Device Number ? >**1<CR>**

The device number corresponds to the board number you wish to open.

Remote Terminal ? >**1<CR>**
Subaddress ? >**4<CR>**

Type the remote terminal and subaddress number you wish to set up. Remote terminal number 1 and subaddress 4 are set up for this example.

'r'= receive or 't'= transmit
Message Type ? >**r<CR>**

Type *r* for a receive message or *t* for a transmit message.

Number of Buffers ? **>2<CR>**

Type the number of data buffers you wish to be associated with this subaddress. Double buffering is set up for this message.

*Filter Table Entry (hex) ? >***0<CR>**

Enter a hexidecimal value that will be written to the Filter Table Entry for the subaddress.

Bus Controller Management. Type **3<CR>** at the main menu prompt to branch to the Bus Controller Management menu illustrated in Figure B-5. Bus Controller Management provides routines to initialize and allocate bus controller commands (also called links) to chain programs. There are routines to read and write data to and from the link's data buffers and to enable and disable interrupts associated with bus controller operations.

```
                Bus Controller Management Unit Test

1  -  Add Link to Chain          7  -  Stop Chain
2  -  Read Link Data             8  -  Write Link Data
3  -  Read Link Status           9  -  Display Links
4  -  Set Link Control Word      r  -  Read/Write RAM
5  -  Set Error Injection        q  -  Return to Main Menu
6  -  Start Chain

-------------------------------------------------------------------------
Selection: Add Link To Chain       Word Count/Mode Code ? >5
Device Number ?        >1          Control Word (hex)   ? >0
Chain Number ?         >1          Error Word  (hex)    ? >0
RT Number ?            >1          Loop (y/n) ?           >t
SA Number ?            >1
Transfer Type = BCRT
Transfer Type ?        >1
Messages:
-------------------------------------------------------------------------
Menu Selection ? > 1
```

Figure B-5. ABI Bus Controller Management Menu

1 - Add Link to Chain	Adds a link (or a bus controller command block) to a chain of links.
2 - Read Link Data	Reads data from a data buffer associated with a chain link.
3 - Read Link Status	Reads the status response word(s) associated with a chain link.
4- Set Link Control Word	Set the Flags word (word 5) for a specified chain link.

5 - Set Error Injection	Sets the error word associated with a chain link.
6 - Start Chain	Starts the execution of a bus controller chain program.
7 - Stop Chain	Halts the execution of a bus controller chain program.
8 - Write Link Data	Writes data to a data buffer associated with a chain link.
9 - Display Links	Displays the bus controller links that have be intialized.
r - Read/Write RAM	Displays a 128 word block of RAM on the 1553 device beginning at a user supplied offset. Modifying memory is possible. Figure B-3 illustrates a 128-word block of RAM beginning at offset 80h.
q - Return to Main Menu	Returns the user to the main menu (Figure B-1).

If an error occurs, a message will be displayed at the *Message:* prompt. The selection chosen by the user is displayed at the *Selection:* prompt. The steps below described how to initialize a bus controller chain on the ABI board. Program responses are in *italics* type, user responses are in **bold** type, and a <CR> symbol represents a carriage return or enter keystroke.

Setting up a Bus Controller. Type **1<CR>** at the Bus Controller Management menu prompt to add a link to a chain. This call is made once for each link. The routine will prompt the user for the information below. Figure B-5 illustrates this sequence.

Device Number ? >**1<CR>**

The device number corresponds to the board number you wish to open.

Chain Number ? >**1<CR>**

Type the chain number to which to attach this link. If a chain has not been defined, a new chain will be created with this link as its first member. This link is attached to chain number one in this example.

RT Number ? >**1<CR>**
SA Number ? >**1<CR>**

Type the remote terminal and subaddress number you wish to set up. Remote terminal number 1 and subaddress 1 are set up for this example.

0=mode 1=bcrt 2=rtbc 3=rtrt 4=delay 5=bcrt brdcst 6=rtrt brdcst
Transfer Type ? >**1<CR>**

Type the number corresponding to the type of bus transaction for this link. Valid bus command type codes are listed for the user. A BC to RT message (transmit) is set up for this link.

Word Count/Mode Code ? >**5<CR>**

Type the number of data words which are to be exchanged with the bus controller or the mode code number. Five data words will be transmitted when this link is executed.

Control Word (hex) ? >**0<CR>**

The control word includes interrupt bits corresponding the chain program. This value must be entered in hexadecimal. The flag word in this example is zero.

Error Word (hex) ? **>0<CR>**

The error word includes the error bits for the Error Injection control. The value must be entered in hexadecimal. The error word is zero in this example.

Loop (y/n) ? >**y<CR>**

The Loop will cause a chain to loop or halt after the last link has been executed. In this example, the chain program will loop continuously.

Messages: Link Number is 1 *Hit CR to Continue*

A unique identifier, called a link handle, is associated with each link and returned by the program. In this example, the link handle is 1. This identifier is used by other bus controller routines such as *Write link Data* and *Read Link Data* to reference this link. The menu selection Display Links will display a list of link handles.

Bus Sequential Monitor Management. Type **4<CR>** at the main menu prompt to branch to the Bus Sequential Monitor Management menu illustrated in Figure B-6. Bus Sequential Monitor Management provides routines to create and allocate a sequential monitor buffer. There are routines to enable monitoring on specifies subaddresses, read the buffer, and force a buffer swap.

1 - Create Sequential Monitor Buffer	Creates and allocates a sequential monitor buffer.
2 - Enable Subaddress Monitoring	Enables sequential monitoring on a specified subaddress.
3 - Disable Subaddress Monitoring	Disables sequential monitoring on a specified subaddress.
4 - Read Monitor Buffer	Reads the sequential monitor buffer which is not being used by the microcode.

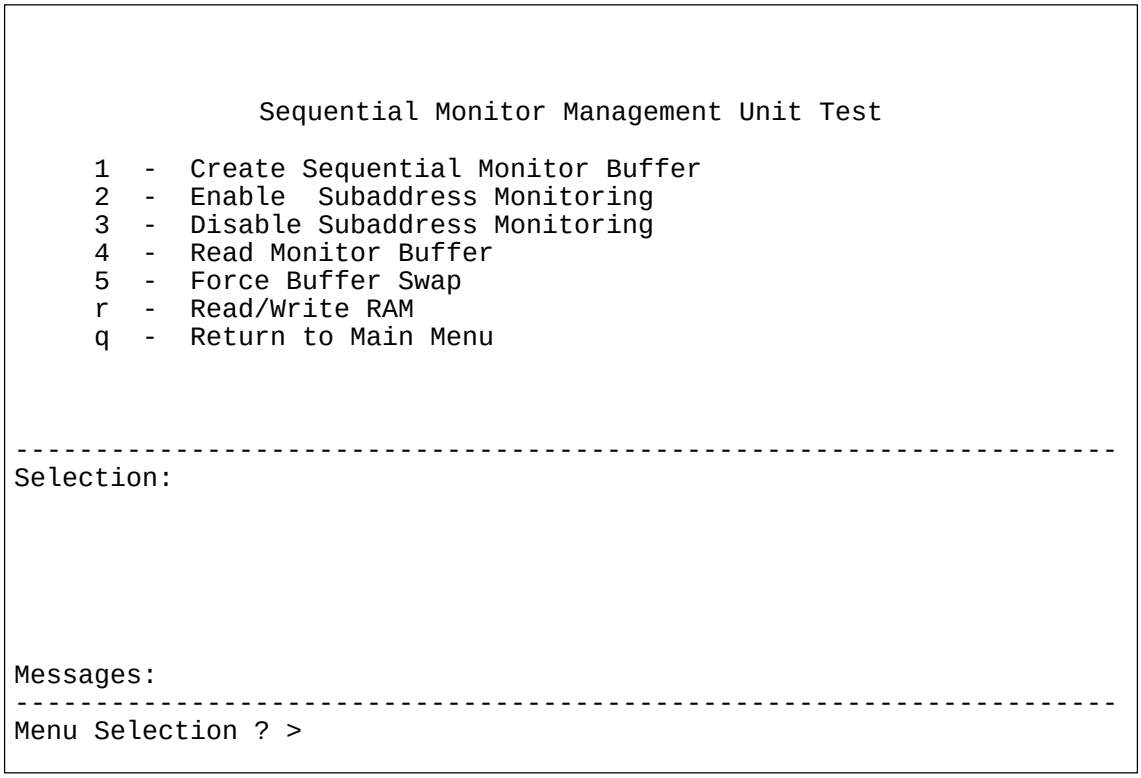


Figure B-6. ABI Bus Sequential Monitor Menu

5 - Force Buffer Swap	Forces the sequential monitoring buffers to swap.
r - Read/Write RAM	Displays a 128 word block of RAM on the 1553 device beginning at a user supplied offset. Modifying memory is possible. Figure B-3 illustrates a 128-word block of RAM beginning at offset 80h.
q - Return to Main Menu	Returns the user to the main menu (Figure B-1).

If an error occurs, a message will be displayed at the *Message:* prompt. The selection chosen by the user is displayed at the *Selection:* prompt.

Setting up the Sequential Monitor. The Sequential Monitor is set up as part of device initialization when *Initialize 1553 Device* is called executed. There is usually never a need to set up the monitor from this menu.

Starting the Bus. Once you have initialized and set up the board for your messages, the last step is to enable the ABI to respond on the bus. If the bus controller is enabled, the selection Start Chain will load a chain program and enable the bus. To start the bus for remote terminal and bus monitor, you must go to the Device Management Menu and execute *Start 1553 I/O*. Branch to the main menu and type **1<CR>** to go to Device Management Menu. Type **6<CR>** to start the

bus. This selection will request a device number. Respond with the device number established when the device was opened.

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