

ARTISAN®
TECHNOLOGY GROUP

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

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

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

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

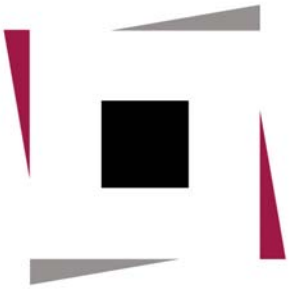
Custom engineering
so your equipment works exactly as you specify.

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

Expert team | Trust guarantee | 100% satisfaction

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

Find the *Curtiss-Wright / DY-4 DMV-IXA7* at our website: *Click HERE*



Dy 4 Systems

The future of high-integrity, embedded technology

Manual for the IXA7
Quad PowerPC Board

Version 2.5
August 2002

Address:	**Dy 4 Systems, Inc.** **741-G Miller Drive, SE** **Leesburg, VA 20175**
Telephone:	**(703) 779-7800**
FAX:	**(703) 779-7805**
Internet:	**www.dy4.com**

Information furnished by Dy 4 Systems, Inc. is believed to be accurate and reliable. However, Dy 4 Systems, Inc. assumes no liability resulting from any omissions in this document, or from the use of the information obtained therein; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Dy 4 Systems, Inc.

Dy 4 Systems, Inc. reserves the right to revise this document and to make changes from time to time in the content hereof without obligation of Dy 4 Systems, Inc. to notify any person or persons of such revision or changes.

No part of this document may be reproduced or copied in any tangible medium, or stored in a retrieval system, or transmitted in any form, or by any means, radio, electronic, mechanical, photocopying, recording, or facsimile, or otherwise, without prior written permission from Dy 4 Systems, Inc.

Trademarks used herein are the property of their respective companies.

PowerPC™ is a trademark of International Business Machines Corporation.

Universe II™ is a trademark of the Tundra Semiconductor Corporation.

VxWorks™, *Tornado*™, and *WindSh*™ are trademarks of Wind River Systems, Inc.

Dy 4 Systems™ is a trademark of Dy 4 Systems, Inc.

© 1999 - 2002 Dy 4 Systems, Inc.

ALL RIGHTS RESERVED PRINTED IN THE USA

Customer Support

Technical Support Hotline

If you have any questions about the IXA7, please contact Dy 4 Systems, Inc.

Telephone: (703) 779-7800

FAX: (703) 779-7805

email: support@dy4.com

Address: Dy 4 Systems, Inc.
741-G Miller Drive, SE
Leesburg, VA 20175

Sales Hotline

If you desire product pricing and availability, please contact Dy 4 Systems, Inc.

Telephone: (703) 779-7800

FAX: (703) 779-7805

email: sales@dy4.com

Address: Dy 4 Systems, Inc.
741-G Miller Drive, SE
Leesburg, VA 20175

Table of Contents

Chapter 1: Getting Started 1-1

1.1 Introduction 1-1
1.2 Assumptions 1-1
1.3 Conventions 1-1
1.4 Product Overview 1-1
1.5 Related Documentation 1-3

Chapter 2: Installation 2-1

2.1 Some Cautions 2-1
2.2 Board Layout 2-1
2.3 Configuring the Board's Jumpers 2-3
2.4 VMEbus Address Configuration 2-5
2.5 Installing PMCs 2-5
2.6 Installing theTransition Module Option 2-6
2.7 Installing the PMC-ENET1 Option 2-7
2.8 Verifying VME Backplane Power 2-8
2.9 Board Boot Operation 2-9
2.10 Power-up Diagnostics 2-12
2.11 Flash Recovery Procedure 2-13
2.12 JTAG/COP Connections 2-17
2.13 Configuring an Emulator for an IXA7 2-17
2.14 Configuring the VxWorks Boot Parameters 2-18
2.15 Installing IXAtools 2-21

Chapter 3: Hardware Architecture 3-1

3.1 Introduction 3-1
3.2 IOPlus 3-1
3.3 SPEs 3-2
3.4 SPE-PCI Bridge 3-2
3.5 Universe II 3-3
3.6 PCI Local Bus 3-3
3.7 Interrupt Controller 3-4
3.8 Board Semaphores 3-4
3.9 Global Memory 3-5
3.10 SPE Local Memory 3-5
3.11 Flash Memory 3-6
3.12 PMC Sites 3-6
3.13 VMEbus Interface 3-11

Chapter 4: Memory Maps 4-1

4.1 Introduction 4-1
4.2 IOPlus Memory Map 4-1
4.3 SPE Memory 4-3
4.4 Global Memory 4-5
4.5 VME Memory 4-6
4.6 Interrupt Controller Register Map 4-7

Chapter 5: Using the IOPlus SmartDMA 5-1

5.1 Introduction 5-1
5.2 SmartDMA Overview 5-1
5.3 When to Use SmartDMA 5-3
5.4 Using SmartDMA 5-4
5.5 Programming SmartDMA 5-8
ixa_cmd_close *5-9*
ixa_cmd_error *5-10*
ixa_cmd_open *5-11*
ixa_cmd_set_next, ixa_cmd_set_opcode, ixa_cmd_set_option, ixa_cmd_set_param *5-12*
ixa_cmd_start *5-14*
ixa_cmd_status *5-15*
ixa_cmd_stop *5-16*
5.6 SmartDMA Commands 5-17
CMD_GENERATE_INT *5-24*
CMD_MOVE_DATA *5-25*
CMD_READ_DATA *5-27*
CMD_RESET *5-28*
CMD_SUPPORT_QUERY *5-29*
CMD_TOGGLE_LED *5-30*
CMD_USER *5-31*
CMD_VME_RMW *5-32*
CMD_WAIT_INT *5-33*
CMD_WRITE_DATA *5-35*

Chapter 6: Programming the IOPlus 6-1

6.1 Introduction 6-1
6.2 VxWorks and the IOPlus 6-1
6.3 The IOPlus Application Programming Interface 6-9
ioplus_calloc *6-11*
ioplus_check_pci_dma_done *6-12*
ioplus_check_vme_dma_done *6-13*
ioplus_free *6-14*
ioplus_generate_interrupt *6-15*
ioplus_malloc *6-16*
ioplus_map_vme *6-17*
ioplus_move_data *6-18*
ioplus_pci_find_device *6-20*
ioplus_read_data *6-21*
ioplus_realloc *6-22*
ioplus_reset *6-23*
ioplus_toggle_led *6-24*
ioplus_unmap_vme *6-25*
ioplus_vme_peek *6-26*
ioplus_vme_poke *6-27*
ioplus_vme_read *6-28*
ioplus_vme_write *6-29*
ioplus_write_data *6-30*

Chapter 7: Programming the SPEs 7-1

7.1 SPE Software Development 7-1
7.2 The Common Boot Code 7-1
7.3 Performance Monitoring Capabilities 7-13
7.4 VxWorks and the SPEs 7-14
7.5 Function Reference 7-16
getchar *7-17*

ixa_cache_enable 7-18
ixa_cache_disable 7-19
ixa_cache_flush 7-20
ixa_cache_invalidate 7-21
ixa_cache_inv_all 7-22
ixa_cache_sync 7-23
ixa_cache_throttle_read 7-24
ixa_cache_throttle_write 7-25
ixa_cmd_VME_error 7-26
ixa_cmd_VME_read, ixa_cmd_VME_write 7-27
ixa_cmd_VME_status 7-29
ixa_delay 7-30
ixa_delay_msec, ixa_delay_sec, ixa_delay_usec 7-31
ixa_dma_init 7-32
ixa_dma_start 7-33
ixa_evt_disable 7-36
ixa_evt_enable 7-37
ixa_evt_get 7-38
ixa_evt_set 7-39
ixa_evt_restore 7-40
ixa_flash_delete, ixa_flash_read, ixa_flash_write 7-41
ixa_get_cluster_id 7-42
ixa_get_proc_id 7-43
ixa_get_proc_info 7-44
ixa_get_proc_rev 7-45
ixa_get_proc_type 7-46
ixa_get_sysproc_id 7-47
ixa_init 7-48
ixa_int_ack 7-49
ixa_int_disable 7-50
ixa_int_enable 7-51
ixa_int_getvect 7-52
ixa_int_lock 7-53
ixa_int_setpri 7-54
ixa_int_setvect 7-55
ixa_int_unlock 7-56
ixa_ipi_disable, ixa_ipi_enable 7-57
ixa_ipi_interrupt 7-58
ixa_led_blink 7-59
ixa_led_blink2 7-60
ixa_led_off 7-61
ixa_led_on 7-62
ixa_mmu_get_page_size 7-63
ixa_mmu_map_addr 7-64
ixa_mmu_map_block 7-65
ixa_mmu_map_page, ixa_mmu_map_pages 7-66
ixa_mmu_peek_l, ixa_mmu_poke_l 7-67
ixa_mmu_peek_p, ixa_mmu_poke_p 7-68
ixa_mmu_remap_block 7-69
ixa_mmu_remap_page, ixa_mmu_remap_pages 7-70
ixa_mmu_set_block_attr 7-71
ixa_mmu_set_page_attr 7-72
ixa_mmu_unmap_block 7-73
ixa_mmu_unmap_page, ixa_mmu_unmap_pages 7-74
ixa_PCI_config_read, ixa_PCI_config_write 7-75

ixa_PCI_io_read, ixa_PCI_io_write 7-76
ixa_pci_to_local, ixa_local_to_pci 7-77
ixa_local_to_pci_dev 7-78
ixa_PCI_read, ixa_PCI_write 7-79
ixa_pm_init 7-80
ixa_pm_reset 7-82
ixa_pm_term 7-83
ixa_pm_display_stats 7-84
ixa_pm_display_trace 7-85
ixa_pm_start 7-86
ixa_pm_stop 7-87
ixa_proc_is_iop 7-88
ixa_proc_is_750 7-89
ixa_proc_is_7400 7-90
ixa_proc_is_7410 7-91
ixa_sem_release 7-92
ixa_sem_request 7-93
ixa_tas_cluster 7-95
ixa_tas_local 7-96
ixa_tas_vme, ixa_tas_clear_vme 7-97
ixa_temp_read 7-98
ixa_timer_cancel 7-99
ixa_timer_create 7-100
ixa_timer_get_ticks_per_second 7-101
ixa_timer_get_TBL 7-102
ixa_timer_get_time 7-103
ixa_timer_get_usec 7-104
ixa_timer_get_timebase 7-105
ixa_timer_init 7-106
ixa_timer_msec_to_ticks, ixa_timer_usec_to_ticks, ixa_timer_sec_to_ticks 7-107
ixa_timer_set 7-108
ixa_timern_enable, ixa_timern_disable 7-109
ixa_timern_get_ticks_per_second, ixa_timern_get_timebase 7-110
ixa_timern_msec_to_ticks, ixa_timern_sec_to_ticks, ixa_timern_usec_to_ticks 7-111
ixa_timern_read 7-112
ixa_timern_set 7-113
ixa_timern_start, ixa_timern_stop 7-114
ixa_timern_trap 7-115
ixa_VME_dma_in, ixa_VME_dma_out 7-116
ixa_VME_open 7-119
ixa_VME_close 7-122
ixa_VME_map 7-123
ixa_VME_remap 7-124
ixa_VME_read, ... 7-125
ixa_VME_write 7-125
ixa_VME_peek, ixa_VME_poke 7-128
ixa_VME_probe 7-130
ixa_VME_int_enable, ixa_VME_int_disable 7-132
ixa_VME_getvect, ixa_VME_setvect 7-133
ixa_VME_int_gen 7-134
ixa_VME_mailbox_enable, ixa_VME_mailbox_disable 7-135
ixa_VME_mailbox_getvec, ixa_VME_mailbox_setvect 7-136
ixa_VME_mailbox_post, ixa_VME_mailbox_set 7-137
ixa_vme_to_local, ixa_local_to_vme 7-138
printf 7-139

putchar *7-140*
puts *7-141*
sprintf *7-142*

Chapter 8: Programming the Flash Memory 8-1

8.1 Introduction 8-1
8.2 The IXA Board Configuration Utility 8-1
8.3 VME Configuration 8-5
8.4 PCI Configuration 8-8
8.5 SPE Configuration 8-10
8.6 IOPlus Configuration 8-13
8.7 Firmware Configuration 8-16
8.8 Flash Page 8-18
8.9 Health and Status Page 8-19
8.10 Burning Flash using the Ethernet Burn Utility 8-20
8.11 The IXA Flash Burn Utility 8-24
8.12 Flash Validation 8-32

Chapter 9: Host Software 9-1

9.1 Introduction 9-1
9.2 Porting HostAPI 9-2
9.3 HostAPI Functions 9-10
host_board_close *9-11*
host_board_open *9-12*
host_board_reset *9-13*
host_board_status *9-14*
host_close_flash_params *9-15*
host_free *9-16*
host_get_board_type *9-17*
host_load_flash *9-18*
host_load_program *9-20*
host_malloc *9-21*
host_map_resource *9-22*
host_memory_read *9-24*
host_memory_write *9-25*
host_open_flash_params *9-26*
host_read_board_info *9-27*
host_read_error_log *9-28*
host_set_start_address *9-29*
host_unmap_resource *9-30*
host_vme_read *9-31*
host_vme_write *9-32*
host_write_config_param *9-33*

List of Figures

Figure 1.1 – Block diagram of the IXA7 1-2
Figure 2.1 – IXA7 board layout, top and front panel views 2-2
Figure 2.2 - Partial Bottom View of an IXA735 Baseboard Indicating the Location of Surface Mount LEDs 2-3
Figure 2.3 - Transition Module Layout 2-7
Figure 2.4 - PMC-ENET1 Layout (Bottom View) 2-8
Figure 3.1 - IXA7 Board Architecture 3-1
Figure 4.1 - Interrupt Mask Register Format for CPE (IOPlus) processor 4-9
Figure 4.2 - INT Interrupt Mask 1 Register Format for SPE processors 4-9
Figure 4.3 - INT Interrupt Mask 2 Register Format for SPE processors 4-10
Figure 4.4 - INT Interrupt Mask 3 Register Format for SPE processors 4-10
Figure 4.5 - Processor Interrupt Generation Registers 4-13
Figure 4.6 - SPE AB Status Registers 4-16
Figure 4.7 - SPE CD Status Registers 4-17
Figure 4.8 - IOPlus Interrupt Status Registers 4-18
Figure 4.9 - VME Geographical Address Register Format 4-19
Figure 5.1 - Command Packet 5-3
Figure 8.1 - Initial Screen (before connecting to board) 8-2
Figure 8.2 – "Connecting" message window 8-3
Figure 8.3 – Connection Timeout Dialog Box 8-3
Figure 8.4 – Old Burn Task 8-3
Figure 8.5 – Product Information Page (after successful connection) 8-4
Figure 8.6 - VME Configuration Page 8-6
Figure 8.7 - PCI Configuration Page 8-9
Figure 8.8 - SPE Configuration Page 8-11
Figure 8.9 - IOPlus Configuration Page 8-13
Figure 8.10 - Firmware Configuration Page 8-17
Figure 8.11 - Flash Page 8-19
Figure 8.12 – Health and Status Page 8-20
Figure 8.13 – Deleting Existing SPE Programs 8-21
Figure 8.14 – Changing Startup Code 8-21
Figure 8.15 – Changing VxWorks Boot ROM 8-21
Figure 8.16 – Changing VxWorks Boot ROM 8-22
Figure 8.17 – Invalid Checksum 8-22
Figure 8.18 – Burning Program 8-22
Figure 8.19 – Screen During Burn Operation 8-23
Figure 8.20 – Error Writing to Flash 8-23
Figure 8.21 – Burn Completed with Errors 8-24
Figure 8.22 – Burn Completed Successfully 8-24
Figure 9.1 - HostAPI Relationship Diagram 9-1
Figure A.1 - Command Packet A-1
Figure A.2 - Source and Destination Command Packet Fields A-4
Figure A.3 - Host Process ID Assignment with Two Host Processes Already Attached A-5
Figure A.4 - All List Address Table Entries in Global Memory A-7
Figure A.5 - List Address Table Entry Relocated to Local Memory A-8

List of Tables

Table 2.1 – Jumper Block Definition 2-4
Table 2.2 - Transition Module Jumper Definition 2-6
Table 2.3 - PMC-ENET1 Jumper Definitions 2-8
Table 2.4 - IXA7 Power Supply Tolerances 2-9
Table 2.5 - Front Panel LED Diagnostic Codes 2-12
Table 2.6 - Serial Port Connector Definition 2-19
Table 2.7 - Multi-Board IP Mapping Example 2-21
Table 3.1 - PMC to VME Connector Mapping 3-8
Table 3.2 - PMC Maximum Current Loads 3-11
Table 3.3 - P0 Connector Definitions 3-12
Table 3.4 - P1 Connector Definitions 3-13
Table 3.5 - P2 Connector Definition 3-14
Table 4.1 - IOPlus-Local and PCI Memory Map 4-1
Table 4.2 - SPE A/B Cluster Memory Map 4-3
Table 4.3 - SPE C/D Cluster Memory Map 4-4
Table 4.4 - PCI Memory Map 4-5
Table 4.5 - Global Memory Map 4-6
Table 4.6 - VME Memory Map 4-6
Table 4.7 – Interrupt Controller Memory Map for Cluster AB 4-7
Table 4.8 – Interrupt Controller Memory Map for Cluster CD 4-8
Table 4.9 – Interrupt Controller Memory Map for the IOPlus 4-8
Table 4.10 - Bit Field Definitions for the Interrupt Mask Registers 4-10
Table 4.11 - Relationship of General Interrupt Status Register Fields to Other Status Registers 4-15
Table 4.12 - VME Geographical Address Assignments 4-20
Table 4.13 – Format of USER_SEL_XX Registers 4-21
Table 4.14 –USER_SEL Register Settings 4-21
Table 5.1 – Command Table 5-17
Table 5.2 - Memory Section IDs 5-17
Table 5.3 - Format of Version Information Table (Table ID 24) 5-18
Table 5.4 - Format of Status Information Table (Table ID 25) 5-19
Table 5.5 - Format of Board Information Table (Table ID 26) 5-20
Table 6.1 - IOPlus VxWorks BSP Features 6-2
Table 6.2 - IOPlus VxWorks BSP Functions 6-3
Table 6.3 - RAM Map for VxWorks 6-5
Table 6.4 - Default BAT Configuration 6-5
Table 6.5 - Default PTE Configuration 6-6
Table 6.6 - VxWorks BSP Device Drivers 6-6
Table 6.7 - Interrupt Vectors and Priorities 6-7
Table 7.1 - SPE Local Memory Map with no CBC (64 MB) 7-2
Table 7.2 - SPE Local Memory Map with CBC (64 MB) 7-2
Table 7.3 - SPE VxWorks BSP Features 7-15
Table 7.4 - SPE VxWorks BSP Functions 7-15
Table 8.1 - *ldflash* Commands 8-26
Table 8.2 - Flash Sections that can be Modified by Users 8-27
Table 8.3 - Flash Sections that can be Modified by Users at the Direction of Dy 4 Systems 8-28
Table 8.4 - Flash Sections not Modifiable by Users 8-28
Table 8.5 - Flash Configuration Parameters 8-29
Table A.1 - Assignment of Processor IDs A-3
Table A.2 - Command Option Register A-12
Table A.3 - Status Register A-13

Table A.5 - Flash Memory Data Types 14
Table A.6 - Example Flash Directory 15

Chapter 1: Getting Started

1.1 Introduction

This manual describes the IXA7 and its operation. It is organized into three sections. The first section contains an introduction to the product, installation instructions, and some general operational information to enable you to get the board up and running. The next section provides a detailed reference on the hardware architecture. The last section provides a detailed reference for the IXA7 software.

1.2 Assumptions

We wrote this manual assuming that users would have a basic level of expertise. That expertise includes:

a solid foundation in C programming
an understanding of the basics of the VMEbus
an understanding of the basics of Digital Signal Processing

1.3 Conventions

This manual uses several conventions. They are:

An asterisk following the signal name (e.g. BERR*) indicates active low.
0x00000 indicates hexadecimal numbers and offset addressing.
Software listings use a courier font.

A gray highlighted box emphasizes an important point.

! An exclamation point and a gray highlighted box express caution.

1.4 Product Overview

The IXA7 is a 6U VME board featuring four Motorola PowerPC 750 or 7400 processors, a PowerPC referred to as the IOPlus, and two PMC sites. Figure 1.1 depicts the IXA7 architecture.

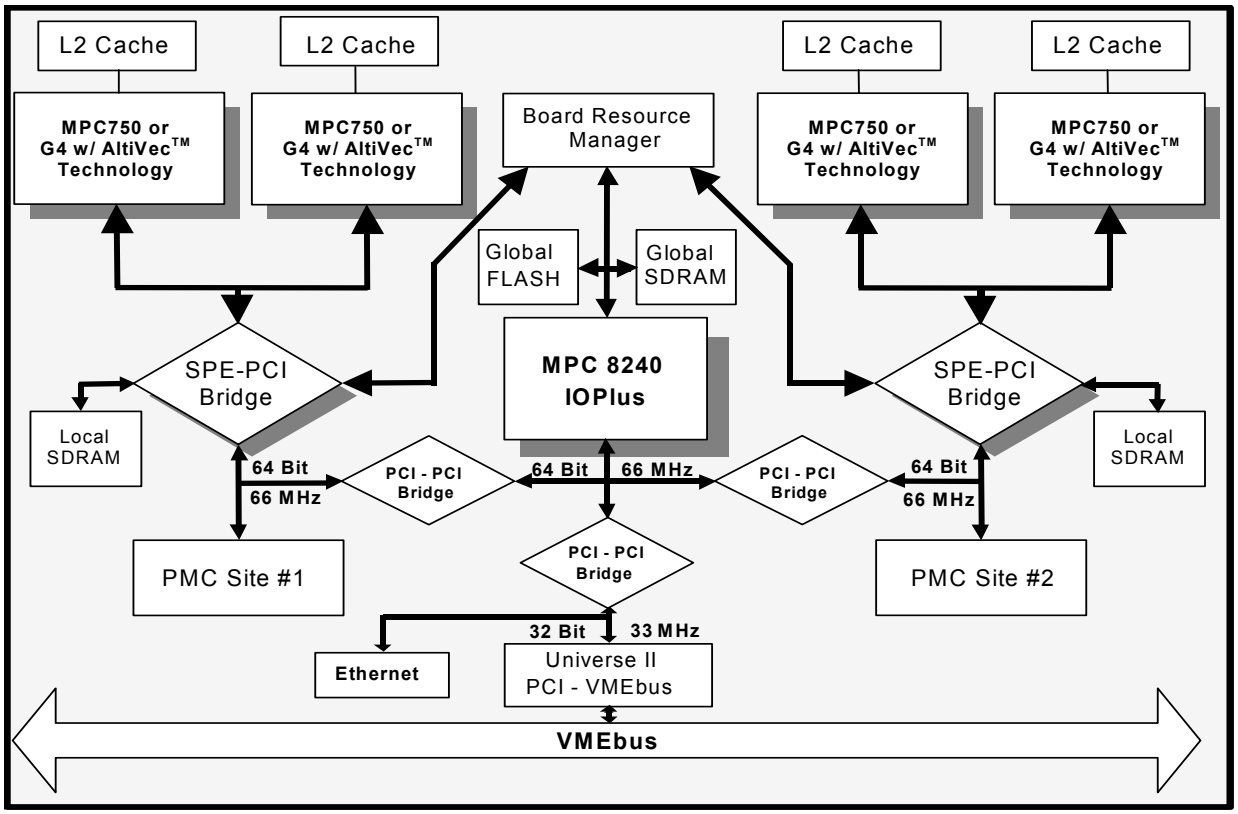


Figure 1.1 – Block diagram of the IXA7

The IXA7 board can be functionally divided into three sections, two Signal Processing Element (SPE) clusters and one Core Processing Element (CPE). The SPE clusters are the two "sides" of the design and each consist of two PowerPC processors, L2 cache, A SPE – PCI bridge, local SDRAM and a PMC site. The CPE consists of: the IOPlus processor, global SDRAM, global Flash, three PCI to PCI (PCI-PCI) bridges, the VME interface and the Board Resource Manager logic. As shown on the block diagram, the board resources are accessible to each SPE over a Port X interface from the SPE – PCI bridge. This means that the SPE processors can access particular board resources without tying up bandwidth on the PCI buses. Each SPE is connected to the CPE through a PCI-PCI bridge. The PCI-PCI bridges provide PCI bus isolation so that simultaneous data movement can occur over the PCI bus in each element.

The IOPlus is responsible for board initialization, Power On Self Test, VMEbus and board resource management functions. It can also be used for applications as long as the developer follows Dy 4 Systems' guidelines.

Both local and global memory is available on the IXA7 board. The local memory can be populated with up to 128 MB of SDRAM (100 MHz) per SPE. Global memory is accessible by the IOPlus and the four PowerPC processors, and can be populated with up to 64MB of SDRAM (100 MHz).

The internal 66MHz, 64-bit PCI buses allow for multiple, simultaneous high-speed transactions. Because of the segmented architecture of the PCI buses, both PMCs can be transferring data to local memory, while the VMEbus accesses global memory simultaneously. Please note that while the PCI buses are 64 bits wide, the PCI data width for the SPE-PCI bridges and the IOPlus is 32 bits (not 64 bits).

The IXA7 supports standard development environments such as VxWorks, and is further enhanced by Dy 4 Systems' IXAtools software package. This package contains VxWorks board support packages (BSPs) for both the IOPlus processor and the SPE processors (750, 7400, or 7410). IXAtools further provides a board support library of "C" functions that support: cache control, memory mapping, timer control, DMA control, interrupt services, LED control, PCI support, and IOPlus command control. It also provides a standard I/O library that contains some basic standard I/O calls, such as printf , sprintf, and putchar. Should the flash on the IXA7 board become corrupted, IXAtools includes the files necessary to reburn the flash to bring the board back to working condition. Additionally, a host resident C library is included in IXAtools that allows a host processor to load, start, and reset the SPEs across the VMEbus. The library of C functions is supplied in source code format so the developer can compile and link these functions into a host application. Host support is offered for a number of processor and operating system configurations. Contact Dy 4 Systems for a current list.

For those who wish to operate in a well-defined development environment, a VxWorks kernel may be run on each processor. Further, all processors are individually addressable over the Ethernet. This provides a flexible multiprocessing development and debug environment through the Tornado interface. A serial interface into the board is also provided.

The user of the IXA7 board may also choose to write application code, foregoing the use of any operating system or kernel. In this scenario, the user may use the JTAG port for code download, execution and debug. In another scenario, a user may run a VxWorks kernel on the IOPlus while using the SPE processors without a real-time kernel.

1.5 Related Documentation

The following publications provide additional reference information:

Conduction Cooled PCI Mezzanine Card (CCPMC) (VITA 20-199x) Draft Standard; Draft 1.11, November, 1999

Draft Standard Physical and Environmental Layers for PCI Mezzanine Cards: PMC; IEEE P1386.1 Draft 2.1; October 11, 1999

Draft Standard for a Common Mezzanine Card Family: CMC; IEEE P1386 Draft 2.1; October 11, 1999

PMC-P4 Pin Out Mapping to VME-P0 and VME64x-P2 Draft Standard (VITA 35-199x), DRAFT 0.3, 30 June 1999.

Draft Processor PMC Standard (VITA 32-199x) Draft 0.3, October 30, 1999.

PCI Local Bus Specification Revision 2.2, 18 December 1998.

PowerPC^{tm} Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, Motorola

American Nation Standard for VME64, ANSI/VITA 1-1994, 10 April 1995.

American Nation Standard for VME64 Extensions, ANSI/VITA 1.1-1997, 7 October 1998.

Universe II User Manual, Tundra Semiconductor Corporation, Spring 1998.

Universe II Manual Addendum, Tundra Semiconductor Corporation, 21 June 1999.

You can obtain a VMEbus specification through the VMEbus/Futurebus Industry trade Association (VITA) at:

VITA – USA
7825 East Gelding Drive, Suite 104
Scottsdale, AZ 85260
USA
http://www.vita.com
(602) 951-8866

VITA –Europe
PO Box 192
5300 AD Zaltbommel
The Netherlands

TEL: 31.4180.1466

Chapter 2: Installation

2.1 Some Cautions

The IXA7 was designed to provide a long reliable period of service. However, like most sophisticated electronic devices certain precautions must be taken when installing or handling the product. The following warnings should be heeded at all times.

> ! WARNING: The board components are static sensitive. Use care and static control when handling the circuit board.

> ! WARNING: Do not short the JTAG pins together.

> ! WARNING: The components require both 3.3v and 5v power. Do not plug an IXA7 board into a 5V-only chassis.

> ! WARNING: Each PMC site is factory configured for either 3.3 V or 5 V signaling bus levels (both PMC sites on the IXA board can be configured the same or differently). A PMC can be a 3.3 V board (uses 3.3 V signaling), a 5 V board (uses 5 V signaling), or a universal board (autodetects signaling level). Do not install a 5 V PMC on a 3.3 V PMC site. Do not install a 3.3 V PMC on a 5 V site. Installing incompatible PMCs can permanently damage the IXA baseboard and/or the PMC.

2.2 Board Layout

Figure 2.1 illustrates the physical layout of the IXA7 board as viewed from the top of the board and the front. This diagram is to be used with the rest of this chapter to locate the board jumpers, and status LEDs.

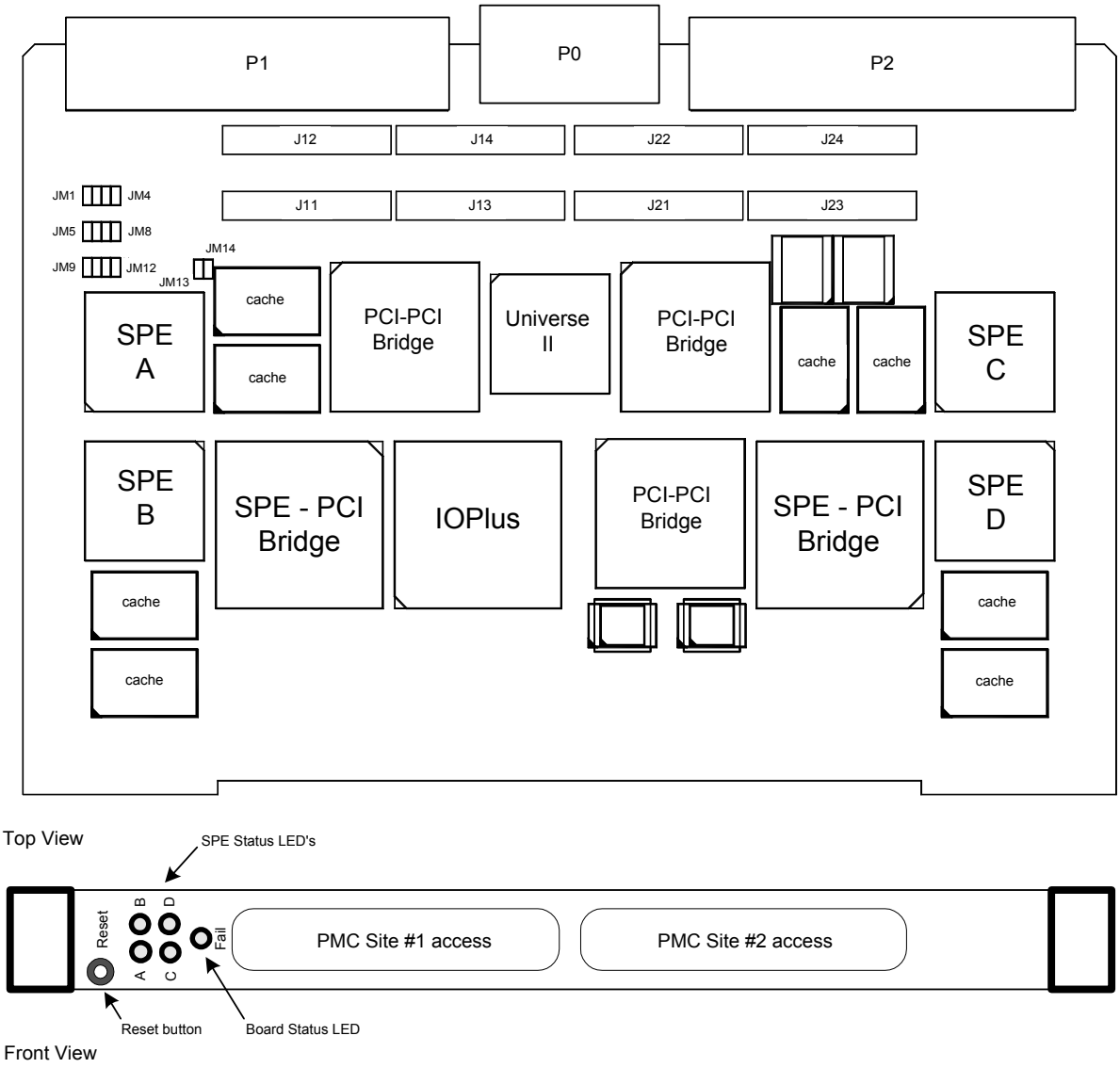


Figure 2.1 – IXA7 board layout, top and front panel views

The following paragraph applies to IXA735 Baseboards ONLY

A set of four surface mount LEDs on the bottom of the baseboard duplicate the function of the four front panel Green LEDs. The location of these LEDs is shown in the Figure 1.0. The LEDs, from left to right, are assigned to processors A, B, C, and D. These LEDs were added since L100 conduction cooled boards do not have the front panel green LEDs. Although in many cases, these LEDs may be hidden from the user by other boards in the VME chassis, they do provide a means to observe the LEDs in cases where the boards in the VME chassis can be separated by at least one slot.

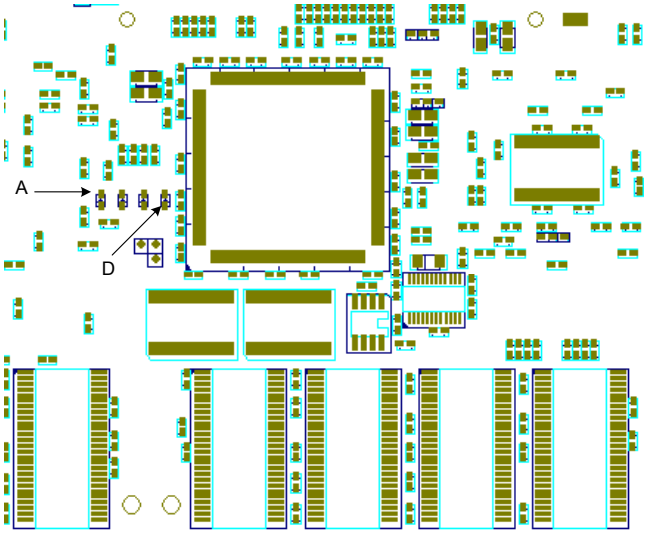


Figure 2.2 - Partial Bottom View of an IXA735 Baseboard Indicating the Location of Surface Mount LEDs

2.3 Configuring the Board's Jumpers

Table 2.1 defines the jumper block (JM1 – JM14) on the IXA7 that resides at the top rear of the board. These are the only jumpers on the board. Jumpers are available for setting the VMEbus base address of the board, and selecting various board operational modes. Jumpers identified as Dy 4 Systems reserved must be left in the default setting and only changed under direction of Dy 4 Systems customer support.

Table 2.1 – Jumper Block Definition

Jumper	On	Off	Default
JM1	A31 of VME address is zero	A31 of VME address is one. Therefore adds 0x80000000 to VME address of board.	Off
JM2	A30 of VME address is zero	A30 of VME address is one. Therefore adds 0x40000000 to VME address of board.	On
JM3	A29 of VME address is zero	A29 of VME address is one. Therefore adds 0x20000000 to VME address of board.	On
JM4	A28 of VME address is zero	A28 of VME address is one. Therefore adds 0x10000000 to VME address of board.	On
JM5	Standard Startup/Runtime code will execute	Minimal boot recovery code will execute, allowing flash to be re-burned or reconfigured (Note 1)	on
JM6	VME address set according to flash parameters.	VME address set by JM1-JM4, no matter how flash parameters are set. Load VxWorks Recovery BootROM when JM5 is removed (i.e. when the board is in recovery mode).	on
JM7	Load and run POST in SPEs, then load user program into SPEs from flash upon reset	Do not load SPE POST, or Common Boot Code, or user programs (including SPE Boot ROM) into SPEs from flash upon reset	on
JM8	Enable IOPlus VxWorks boot rom load (Note 2)	Do not load or execute IOPlus VxWorks boot rom	on
JM9	hard reset of the board	normal operation	off
JM10	board reset drives VME reset	board reset does not drive VME reset	off
JM11	Reserved		off
JM12	JT1 (IOPlus) JTAG enable	JT1 JTAG disabled	off
JM13 & JM14	These two jumpers are used to select which SPE the JTAG header corresponds to. They have the following meaning: JM13 / JM14 / SPE On / On / A On / Off / B Off / On / C Off / Off / D		

Note 1: Removing JM5 boots the board into a minimal recovery state, so that firmware can be re-burned, or flash parameters can be re-configured. This is most useful when flash firmware or configuration parameters have become corrupted.

Note 2: JM8 has no effect unless the VxWorks boot rom is installed in flash and enabled through the IOPlus flash configuration parameters. See *IOPlus Configuration Parameters* in Chapter 8.

There are no jumpers necessary for configuring the IXA7 as a VMEbus slot one system controller. The Universe II VME controller will automatically sense the board is in slot one and configure itself accordingly. If you are using the board as a slot one controller you may want to put a jumper on JM10 so that board resets will reset the VMEbus. Note, however, that the Bus Grant and IACK signals on the VME backplane must be jumpered correctly for automatic slot one identification to occur. Refer to the documentation accompanying your VME chassis for more information.

2.4 VMEbus Address Configuration

JM6 is used to select whether the VMEbus base address of the board is determined by the jumpers JM1-4, or the configuration parameters in the flash memory. When JM6 is removed, the VME base address of the board is set using JM1-4, regardless of the configuration parameters stored in flash. When JM6 is installed, the VME base address is set based on flash configuration parameter 0. To learn more about the programing of the flash configuration parameters see Chapter 8.

When the IXA7 board is prepared for shipment, the flash parameters are configured for a board base address of 0x80000000, and a 64 MB address window.

2.5 Installing PMCs

PCI Mezzanine Cards (PMCs) should be installed according to the directions provided by the PMC vendor. Before installation, however, you must remove the dummy bezel that is attached to the IXA7 by Dy 4 Systems before shipment. The purpose of the dummy bezel is to provide support for the front panel during insertion into a chassis, and it should never be removed unless a PMC is being installed. To remove the bezel, unscrew the two bezel screws located on the bottom of the board using a #1 Phillips screwdriver. The two screws that were removed, along with two of the four screws shipped with the IXA7 board, shall be used to secure the PMC. Do not use the screws supplied by the PMC vendor.

2.6 Installing theTransition Module Option

If you have purchased the Transition Module, TM, you must configure the module's jumpers prior to use. Figure 2.2 depicts the location of jumpers and connectors on the TM. The connectors are designed to plug into the backside of the VME backplane, and must be plugged into the same slot as the IXA7 board. The transition module provides an ethernet connection, a serial port connection and two JTAG/COP connections. One of the JTAG/COP connections is for the IOPlus (JT1) while the other (JT2) is a multiplexed connection to one of the four SPE processors (processor is selectable via JM13 and JM14 – see Table 2.1). Connector blocks J1 and J11 are reserved for Dy 4 Systems use.

> ! WARNING: Do not short any of the connector block pins (JT1, JT2, JM13, and JM14) together.

The TM has jumper blocks that are used for configuring the serial port. Table 2.2 provides the jumper definitions. Connecting a console to the serial port and interacting with the IOPlus VxWorks boot loader software on the IXA7 configures the Ethernet port. Ethernet port configuration and the serial port pin out are documented later in this chapter.

If you use the transition module in conjunction with a PMC-ENET1 PMC, you must install jumper JM1 on the transition module, and use the serial port on the PMC module.

Table 2.2 - Transition Module Jumper Definition

Jumper	Settings	Definition
JM1	off	Off = enable I2C, on = inhibit I2c
JM2	on	RS422/RS485 - Requires factory assembled option D
	off	RS232 (default setting)
JM3, JM4	off	Reserved
JM5, JM6	on , on	19.2K baud
	on , off	9600 baud (default setting)
	off , on	2400 baud
	off , off	1200 baud
JM7 – JM19	off	Reserved

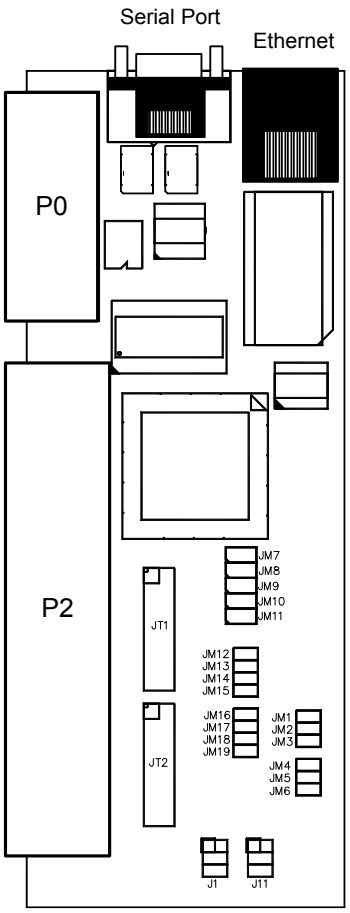


Figure 2.3 - Transition Module Layout

2.7 Installing the PMC-ENET1 Option

If you have purchased the PMC-ENET1 option, you must configure the module's jumpers prior to use. Figure 2.3 depicts the location of jumpers and connectors on the PMC-ENET1. The PMC module may plug on either PMC slot 1 or 2. Four screws shipped with the PMC-ENET1 board should be used to secure the PMC module to the IXA7. The PMC-ENET1 provides Ethernet and serial port connections.

The PMC-ENET1 has jumper blocks that are used for configuring the serial port. Table 2.3 provides the jumper definitions. Connecting a console to the serial port and interacting with the IOPlus VxWorks boot loader software on the IXA7 configures the Ethernet port. Ethernet port configuration and the serial port pin out are documented later in this chapter.

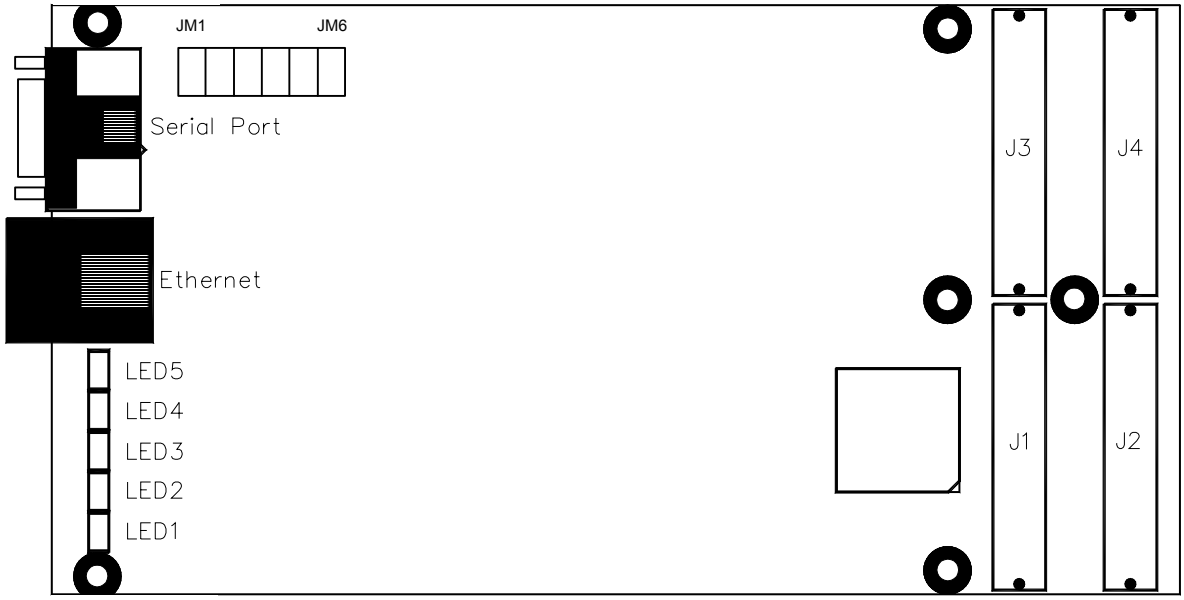


Figure 2.4 - PMC-ENET1 Layout (Bottom View)

Table 2.3 - PMC-ENET1 Jumper Definitions

Jumper	Settings	Definition
JM1	Off	Dy 4 Systems reserved
JM2	on	RS422/RS485 - Requires factory assembled option D
	off	RS232 (default setting)
JM3, JM4	off	Dy 4 Systems reserved
JM5, JM6	on , on	19.2K baud
	on , off	9600 baud (default setting)
	off , on	2400 baud
	off , off	1200 baud

2.8 Verifying VME Backplane Power

The IXA7 uses +5 V, 3.3 V, +12 V, -12 V power sources from the VMEbus backplane. The voltage tolerances and power requirements for each supply are shown in Table 2.4. Missing or below level voltages will cause the power detection circuitry on the board to hold the board in a powered down state.

Table 2.4 - IXA7 Power Supply Tolerances

Power Supply	Required Tolerance	Notes
+5V	4.875 V to 5.25 V	Less than 50 mV Peak-to-Peak Ripple Noise
+3.3V	3.25 V to 3.45 V	Includes regulation variation, noise and ripple.
+12V	11.64 V to 12.6 V	Less than 50 mV Peak-to-Peak Ripple Noise
-12V	-11.64 V to -12.6 V	Less than 50 mV Peak-to-Peak Ripple Noise

2.9 Board Boot Operation

Boot Sequence

After power is applied or the front panel reset switch is depressed, the board goes through its boot sequence. The specifics of the boot sequence are determined by jumper settings and flash parameters. The general case is described here:

1. All processors are reset. The 8240 starts execution from flash memory, and the 74xx processors stall, waiting for their bridges to be enabled. The red FAIL LED illuminates.

2. Depending on jumper settings, the 8240 either proceeds with initialization, or enters flash recovery mode. The remainder of this section describes the completion of the initialization steps.

3. The 8240 initializes the on-board FPGA. It initializes all bridges. It initializes, but does not enable, the SPE bridges.

4. The 8240 places a small executable program into SPE memory, and enables the SPE bridges. The SPEs run this program, completing bridge initialization from the processor side, and then loop, waiting for a more significant download.

5. If enabled in the flash parameters, the IOP coordinates the execution of POST on all processors. Green LEDs flash at this point.

6. If enabled in flash parameters and jumpers, the IOP loads common boot code (CBC) on all SPE processors.
    a. The SPEs begin executing common boot code.
    b. The SPEs initialize their respective memory management units.
    c. The green LEDs display the rotating, faded blinking pattern while waiting for a more significant download.

   d. If enabled, the IOP loads SPE applications programs into SPE memory, normally starting at location 0x20000. The SPE does not start the applications. The lights continue to flash the pattern.

7. The red LED goes out.

8. If enabled in flash, the IOP runs its application. Normally this is a VxWorks boot ROM capable of loading executables over the network connection.

9. The boot ROM uncompresses itself and relocates itself to high IOP memory. The boot ROM counts down, waiting up to 7 seconds for terminal input. You may enter any character to stop the auto-boot sequence. This allows you to change edit the flash boot parameters.

10. After reaching the count down, the boot ROM then examines flash boot parameters, initializes the network device driver, and downloads the specified application.

11. The downloaded application is normally a VxWorks kernel linked with applications and/or debug support tools. This program starts the SPE processors, loads its symbol table, and initializes the on-board shared memory network.

12. Having been started by the IOP, the SPE processors begin execution of their applications. This is normally a VxWorks boot ROM image (etc/spboot.s), executed from memory. This image does the following:

    a. It uncompresses itself and relocates itself to high memory
    b. It derives its boot parameters, including the Ethernet address, host name, and login parameters, from the IOPs boot parameters. See below for how these parameters are altered by the SPEs.
    c. It attaches to the shared memory network.
    d. Using the IOP as a network gateway, each SPE loads its executable image over the network into low memory.
    e. It detaches from the shared memory network, and jumps to the newly loaded image.

13. The loaded image is normally a VxWorks kernel linked with debug support tools and applications. The kernel attaches to the shared memory network, and initializes its applications.

The time from reset until SPE execution of application software is a function of executable size, whether or not the applications boot directly from flash or over the network, and network loading. For most applications, the above sequence completes in well under one minute.

SPE Boot Parameter Alteration

The IOP boot parameters govern the boot process for both the IOP and all SPEs. The SPEs use the same Ethernet host address, login, password, and flags as are used by the IOP. The SPEs ignore the start-up script parameter. The SPEs determine their network address by using the shared memory backplane address specified for the IOP, incrementing the least significant byte for each processor.

The executable file name for the SPEs is derived from the "other" field within the boot parameters. The SPE boot ROM uses the "other" field as the full path name of the executable image to be loaded.

It is often desirable to have each SPE boot using a different executable image. In order to accommodate this, the SPEs scan the "other" field and substitute the following parameters:

%b is replaced by the two-digit VME slot ID (00 through 20), as determined by the VME slot ID read from the backplane. (VME boards only).

%p is replaced by the SPE processor ID: A, B, C, or D.

Example:

If *other* is "/export/home/project/vxWorks5_2", all SPEs boot the file "/export/home/project/vxWorks5_2".

If *other* is "/export/home/project/vxWorks5_2%p", then SPEs boot:
"/export/home/project/vxWorks5_2A" for SPE A,
: "/export/home/project/vxWorks5_2B", for SPE B, etc.

If *other* is "/export/home/project/board%b/vxWorks5_2%p", then SPEs on the board in card slot two boot:
"/export/home/project/board02/vxWorks5_2A" for SPE A,
"/export/home/project/board02/vxWorks5_2B" for SPE B, etc.

This approach permits the use of the same boot ROM and boot parameters for all cards in the system. Of course, you need not specify %b or %p if you want all processors to boot the same executable image.

2.10 Power-up Diagnostics

The front panel's LEDs are used to indicate the health and status of the IXA7 board. The board status LED is red and the four SPE status LEDs are green. See Figure 2.1 for the location of these LEDs. When the board is first powered up, or after a reset, the following behavior should be observed:

1. The board status LED and the SPE status LEDs should all illuminate. The SPE LEDs will quickly turn off, while the board status LED will remain on during board initialization and self-test.

2. When the board self-test passes, the red board status LED will turn off -- this indicates that the board is healthy

3. This sequence should take approximately two seconds.

4. The SPE LED's should blink in a rotating pattern, indicating that the Common Boot Code has run and the processors are waiting for a load.

If a self-test failure occurs, the board status LED will blink four SOS sequences (three short blinks, followed by three long blinks, followed by three short blinks) after the boot sequence has completed. The SPE status LEDs can be used to diagnose the failure as defined in Table 2.5. After the four SOSes, the board status LED will begin to flash rapidly – this flashing indicates that the board has entered "recovery" mode, which allows the board firmware to be re-burned into the flash memory.

When the reset button is pressed and quickly released (much shorter than two seconds), the status LED will illuminate after the reset button is released (events 1 and 2 above). When the reset button is pressed and held for longer than two seconds, the LED will illuminate after approximately two seconds and remain illuminated until after the reset switch is released. If the board status LED does not illuminate, the board has been damaged or one of the required power supply voltages is missing. If this occurs be sure to check the +3.3, +5, +12, and –12 volt supplies.

Table 2.5 - Front Panel LED Diagnostic Codes

Board Status LED	SPE status LEDs				Meaning
	A	B	C	D	
Off	off	off	off	off	board booted successfully and is ready for use
Off	on or off	on or off	on or off	on or off	if one or more green LEDs remains and the red LED is off, this indicates that a problem has occurred programming the Xilinx
On	off	off	off	off	Xilinx programming problem
On	on	off	off	off	A/B SPE POST failure
On	off	on	off	off	C/D SPE POST failure
On	on	on	off	off	Universe II programming problem
On	off	off	on	off	PCI bridge #1 problem
On	off	off	off	on	PCI bridge #2 problem
On	off	off	on	on	PCI bridge #3 problem
On	on	off	on	off	A/B cluster local memory failed test
On	off	on	off	on	C/D cluster local memory failed test
On	on	off	off	on	global memory failed test
On	off	on	on	off	flash memory failed test
On	on	on	on	on	Board held in reset

2.11 Flash Recovery Procedure

The board may have difficulty booting or may behave erratically if flash memory becomes corrupted. Flash corruption can result from loss of power during a flash burn operation, or from burning an inoperative test program into flash memory. In many cases, the board can still boot even with a corrupted flash. In these cases, the procedures listed in section 8.12 describe how to determine the extent of the corruption and repair it. However, in some situations, the board is unable to boot if the flash is severely corrupted. Procedures for recovering the board in these situations are described in this section.

The following situations require flash memory recovery steps in order to restore the board to a working state:

- IOPlus firmware stored in flash has become corrupted.
- Xilinx firmware stored in flash has become corrupted.
- Flash configuration parameters have become corrupted.
- Common Boot Code (CBC) has been improperly configured.
- A user SPE program that is automatically being loaded from flash on power-up is crashing the board.

Flash recovery involves booting the IOP with one of the delivered vxWorks images (either vxWorks or vxWorks5_2), and then re-burning flash memory. The delivered vxWorks images include a flash burn task that services network connections. Once this task is operational, flash memory can be re-burned from a PC using the Board Configuration Utility described in Chapter 8.

If you are not sure which recovery procedure to try, then use the recovery for CBC and SPE errors first. This procedure, if applicable, only takes a few minutes to perform.

Recovering from CBC and SPE Errors

An improperly configured CBC or a faulty SPE image both have the ability to lock up the board shortly after board reset, potentially eliminating the possibility of reburning flash memory. Recovery from these conditions is straightforward. The SPE processors must be disabled, and the IOP must be booted with a vxWorks image that includes an Ethernet flash burn task. (The delivered vxWorks images for the IOP incorporate this task.) This task accepts a new CBC or SPE program over the network and reburns flash:

1. Power down the board.

2. Configure the board to inhibit SPE downloads: on the IXA-7, remove jumper JM7. (On the IXA-4, turn SW-7 off.)

3. Apply power, and boot the IOP using vxWorks or vxWorks5_2 from the delivered software.

4. Using the Board Configuration utility, replace the CBC or SPE software.

5. Power down.

6. Install JM7 (or restore SW-7).

7. Reapply power to the board.

Recovering from VxWorks Boot ROM or Startup Firmware Errors

A faultyVxWorks Boot ROM image or corrupted startup firmware both have the ability to lock up the board shortly after board reset, potentially eliminating the possibility of reburning flash memory. Recovery from these conditions is straightforward. The IOPlus must be configured to boot the Recovery VxWorks Boot ROM rather than the standard VxWorks Boot ROM, and the Boot ROM is then used to boot a vxWorks image that includes an Ethernet flash burn task. (The delivered vxWorks images for the IOP incorporate this task.) This task allows a new Boot ROM or startup firmware to be reburned using the Ethernet Burn Utility:

1. Power down the board.

2. Configure the board to load the recovery Boot ROM: on the IXA-7, remove jumpers JM5 and JM6. (On the IXA-4, turn SW-5 and SW-6 off.)

3. Apply power, and boot the IOP using vxWorks or vxWorks5_2 from the delivered software.

4. Using the Ethernet Burn utility, replace the VxWorks Boot Rom or startup firmware.

5. Power down.

6. Install JM5 and JM6 (or restore SW-5 and SW-6).

7. Reapply power to the board.

Recovering from Other Errors – Serial Port Download

Recovery from other errors is slightly more involved, and may take up to 15 minutes. This recovery procedure requires that the IOPlus be booted with a vxWorks image having an Ethernet flash burn task (as above). First, a new boot loader must be downloaded using the serial port. Serial port download during recovery is provided in revisions 2.3 or later of the recovery code.

1. Power down the board.

2. Place the board in "recovery" mode: on the IXA-7, remove jumper JM5, and install an EPMC in the upper PMC site. On the IXA-4, set SW-5 off, and install the card in a slot fitted with a rear panel module.

3. Connect the serial port from the EPMC or rear panel module to a PC executing terminal emulation software with file download capability. Attach the Ethernet cable. Configure the PC serial port for 9600 baud, 8 data bits, no parity, one stop bit.

4. Power-up the board -- it will boot into a "recovery" configuration. The red LED will flash rapidly to indicate that the board is in the "recovery" mode.

5. Download the S-record file "etc/dwnldrom.hex" through the serial port. This is a special vxWorks boot loader capable of loading IOP software over the Ethernet interface. The red LED will flash slowly to indicate the download is in progress. The flash rate is tied to the download data rate; pausing the download will pause the flashing. The download process takes 10 - 15 minutes.

6. If a download error is detected, the red LED will begin flashing an "SOS" pattern, and will ignore subsequent serial data. Stop the download from the PC, reset the board, examine the serial parameters, and go back to step 5.

7. If the download was successful, the red LED will turn off and execution control will be given to the downloaded code. DO NOT reset or power off the board at this point.

8. Within a few seconds, the vxWorks system boot prompt will appear on the PC screen. Enter appropriate vxWorks boot parameters, configuring the IOP to boot vxWorks. You MUST boot a vxWorks image having a flash burn task. Use vxWorks or vxWorks5_2 in the IOP directory of the delivered software.

9. Boot vxWorks without resetting the board (enter an '@' at the prompt).

10. Once booted, the IOP can accept flash burn commands using the Ethernet interface. Run the Board Configuration utility, and re-burn the IOP software. If you believe the CBC and SPE loads were corrupted, you may re-burn those now as well.

11. Power off the board.

12. Install jumper JM5 (or set SW-5 "on").

13. Re-apply power to the board -- it will now boot into a full configuration.

Recovering from Other Errors – Flash Burn via the Backplane

The recovery code also accepts flash burn parameters over the backplane. If your installation includes a PC with a PCI to backplane adaptor card with drivers supported by Dy 4 Systems, you can reburn flash over the backplane using the IXA flash burn utility. This tool. Described in section 8.11, reads files on the PC and deposits flash burn commands in the IXA board's memory. The recovery code interprets these commands, erasing and reburning flash as needed.

1. Power down the board. Move the board to a chassis containing the backplane adaptor.

2. Place the board in "recovery" mode. On the IXA-7, remove jumper JM5. On the IXA-4, set SW-5 off. Configure backplane address jumpers for the default backplane address (See table 2.1).

3. Power-up the board -- it will boot into a "recovery" configuration. Initially, the red LED will flash rapidly to indicate that the board is in the "recovery" mode.

4. Run the "ldflash" program, specifying the appropriate input file containing the list of flash commands to be performed. The red LED will flash as each file is downloaded to the board.

5. Power off the board.

6. Install jumper JM5 (or set SW-5 "on").

7. Re-apply power to the board -- it will now boot into a full configuration.

2.12 JTAG/COP Connections

There are no JTAG/COP connectors on the IXA7 board. JTAG/COP connections are made via the TM.

> ! WARNING: Improper connection of the emulator can damage the IXA7 board and/or the emulator. Emulator connector should be keyed.

2.13 Configuring an Emulator for an IXA7

An emulator may be used to interact with each processor and the board hardware on the IXA7 card. Two JTAG/COP connections are provided on the backplane transition module (see Section 2.6) for this purpose. Connector JT1 is used to connect an 8240 emulator to the IOPlus processor while connector JT2 is used to connect a 7xx emulator to one of the SPE processors. When connecting an emulator to JT1 or JT2, make sure that pin one of the pod aligns with pin one on the connector. Figure 2.2 shows pin one for both of these connectors outlined by a box. Always turn off the power before connecting or disconnecting the emulator.

Dy 4 Systems has tested the emulation capabilities of the IXA7 with an EST emulator. Therefore, the information in this section is based on the use of the EST 6xx/7xx and EST 82xx emulators. Both Vision ICE and Vision Probe emulators have been used.

The JTAG/COP interface on PowerPC processors is a combined JTAG interface and Common Onchip Processor (COP) interface. The JTAG interface on PowerPC processors is used for accessing the JTAG scan chain. The COP interface is used for emulation. Unlike JTAG interfaces, the COP interface cannot be daisy chained together. Therefore, since there is only one connector (JT2) on the TM to support the SPE processors, this connector is multiplexed between the four SPE processors. The target processor is chosen via IXA baseboard jumpers JM13 and JM14 (see Table 2.1). The steps for using an EST emulator with the SPE processors are outlined below. For other emulators, follow a similar procedure.

1. Turn off the board (chassis) power.
2. Set JM13 and JM14 to the processor that you wish to target.
3. Install the transition module (TM).
4. Connect the emulator to the JT2 connector on the TM.
5. Power up the IXA7 board.
6. Open the Vision Click software and follow the procedures outlined for opening communication with the emulator.

7. Type "inn" to get into background mode (do not type "in" as this command will attempt to initialize the SPE-PCI bridge).
8. Type "go" to put the emulator and card into the proper state.
9. You should now be able to communicate with the board.

Please note that when the target processor is reset, the entire board resets. This is necessary to make sure that the local processor bus is synchronized correctly and that the SPE-PCI bridges are properly configured. In the case of the EST 7xx emulator, this means that each time the "inn" command is executed, the IXA7 board will reset. Once the board has been reset, it is important to run the emulator (step 8). This assures that the board and COP interface are in the proper mode for further communication. If this is not done, you will not be able to communicate with the target processor.

Also note that in order to load the SPEs from the emulator, the load SPE enable flag in the flash configuration parameters must be unset and flash re-burned (see Chapter 8).

When using an 8240 emulator, no procedures other than those described in the emulator instructions need to be followed.

> ! WARNING: Resetting the SPE target processor ("inn" in the case of the EST emulator) will reset the entire IXA7 board.

2.14 Configuring the VxWorks Boot Parameters

The serial port on the TM module or the PMC-ENET1 is used to configure the VxWorks boot parameters. When the IXA7 is ordered with the VxWorks boot loader, the IOPlus will run board initialization and test software on startup and then load a VxWorks kernel into the IOPlus from flash. This kernel considers the serial port to be a system console and on start-up displays a banner page and commences a seven-second-countdown. If a character is received from a console device connected to the serial port during this countdown, the countdown will stop and a prompt will be displayed on the console. Otherwise, the VxWorks boot loader will try to load from the host defined in the boot parameters.

The VxWorks boot parameters are set from the system console. To work properly, the system console serial port must be set to the same settings as the jumper settings on the TM or PMC-ENET1 (See Table 2.2. or Table 2.3). All baud rate settings use 8 bit, no parity, and 1 stop bit and the driver expects a software handshake protocol (XON/XOFF). Also the serial cable connection must be proper for the device. The micro-mini DB-9 connector is wired as a standard IBM DCE device. See Table 2.6 for the pin definition of the DB-9 for RS-232 and RS422/485 operation.

Table 2.6 - Serial Port Connector Definition

Pin	RS-232 Definition	RS422/485 Definition
1	DCD	No Connection
2	RD	RD-
3	TD	TD+
4	DTR	No Connection
5	GND	GND
6	DSR	No Connection
7	RTS	RD+
8	CTS	TD-
9	RI	No Connection

Once you have the prompt on your system console, type:

```
p<CR>
```

This will display the boot parameters. A list similar to the following will be displayed:

```
boot device         : dc
unit number         : 0
processor number    : 0
host name           : james
file name           :/export/home/tornado/target/config/iop/vxWorks5_2
inet on ethernet    : 207.96.24.235:ffffff00
inet on backplane   : 192.99.22.100:ffffff00
host inet           : 207.96.24.237
user                : wrsuser
ftp password        : wrsuser
flags               : 0x0
target name         : vxtarget
other               :/export/home/tornado/target/config/spe7x/vxWorks
```

To change the boot parameters, type:

```
c<CR>
```

The boot device should be set to "dc" to specify the DEC 21143 Ethernet device for the TM, or "fei" to specify the INTEL 82559 device for the PMC-ENET1. The updated boot parameters are stored in flash on the IXA7 and preserved there when the card is powered off. Therefore, subsequent boots of the card can auto-boot without intervention.

Each SPE processor parses the IOP boot parameters for its own use. The SPEs determine their network address from these parameters. The SPEs also use the same host name, host network address, and host login parameters. Using these parameters, the SPEs download the file specified in the "other" field for their own use.

Other commands are also available at the system console. They can be displayed by typing h<CR> or ?<CR>. Refer to the Tornado User's Guide for further information on the shell interface commands.

Adding Tornado Routes

Before you can successfully communicate with the SPE processors, you must inform your Tornado host system of the shared memory IP addresses that each IXA7 processor uses. Each of the shared memory IP addresses must be associated with the IP address of the IOPlus processor (inet on ethernet) on the user network. To inform your host of this information, add the shared memory addresses to your host's route table. To do this, enter the route add command at the system prompt. The syntax for this command differs between the NT and UNIX operating system, so you should consult your host documentation for the correct format. The routes that should be added are as follows (UNIX format).

```
route add  192.99.22.100   inet on ethernet for IOP
route add  192.99.22.101   inet on ethernet for IOP
route add  192.99.22.102   inet on ethernet for IOP
route add  192.99.22.103   inet on ethernet for IOP
route add  192.99.22.104   inet on ethernet for IOP
```

So, for example, if the IOPlus boot parameters are as shown in the previous list, the "inet on ethernet for IOP" that should be used above is 207.96.24.235. Therefore, the route add command that would be used to add the IOPlus to the route table would be:

```
route add 192.99.22.100 207.96.24.235
```

As you can see, the ffffff00 should not be appended to this address when performing the route add command.

When using more than one IXA7, be sure to assign different IP addresses for each board. Table 2.7 shows an example of a two-board configuration. Note that both the IP addresses for each boards IOPlus are set in each board's boot parameters.

Table 2.7 - Multi-Board IP Mapping Example

Board	Processor	Shared Memory IP Address	Ethernet IP Address
1	IOPlus	192.99.22.100	207.96.24.235
1	SPE A	192.99.22.101	None
1	SPE B	192.99.22.102	None
1	SPE C	192.99.22.103	None
1	SPE D	192.99.22.104	None
2	IOPlus	192.99.22.200	207.96.24.236
2	SPE A	192.99.22.201	None
2	SPE B	192.99.22.202	None
2	SPE C	192.99.22.203	None
2	SPE D	192.99.22.204	None

2.15 Installing IXAtools

The installation procedure for IXAtools appears on the back of the IXAtools CD case. To install the software, you will be required to execute an installation program on the CD that will lead you through the installation. However, installing IXAtools may involve several different software destination selections. This will depend on your development environment and target application configuration.

IXAtools contains software for the following development system components, they are:

PowerPC Development Environment - This is the system where the PowerPC development tools reside. On this system compiling, linking, simulating, and debugging the SPE software, and optionally the IOPlus software, occurs. This system may use a JTAG emulator connection or/and Ethernet connection to permit the load, execution and debug on the IXA7. IXAtools contains support libraries for code development in this environment. If this environment supports a Tornado host, BSPs for both the IOPlus and the SPEs are provided.

Flash Configuration Workstation - IXAtools provides a Board Configuration Utility for writing software and board configuration parameters into the IXA7's flash memory. This utility runs under Windows 95/98/NT and requires an Ethernet connection between the IXA7 and the workstation. Chapter 8 describes the use of this utility. If the PowerPC Development Environment is compatible with this utility, select the installation of this component when installing the rest of the development environment software. Otherwise, you will need to install this utility separately on a compatible system. Installing this utility will create a IXAtools group in your desktop's Programs sub-menu.

Target Host Computer - Typically this will be a single board computer that is located in the VME chassis with the IXA7. Software libraries are provided with IXAtools to support applications running on this computer to load, start, and exchange information with the IXA7 over the VMEbus. This software is provided as verified linkable libraries for a number of host computers. For customers having an unsupported host requirement, the software is provided in source format also with porting procedures that are documented in this manual. See Chapter 9 for more details.

Your requirements will determine how many of the components you need to install and what systems to install them on. Everyone will need to install the development tools, but it is not necessary to program the IOPlus to reap the full benefit of the device since it comes from Dy 4 Systems preprogrammed for run-time board services. If you intend to develop the Host software on a PC, and develop PowerPC software using a different computer, you may need to install the relevant components in the two computers in two separate installations. The installation program allows you to dissect the installation into multiple installs.

Components installed in a Windows environment can be uninstalled by selecting the Add/Remove Icon in the Control Panel and selecting the removal of IXAtools. This uninstall relies on the install.log file and the file unwise.exe to be present at the top-level of the install directory. You can also uninstall the software by running the unwise.exe program directly.

Reprogramming the Flash Memory

The flash memory is programmed with the most recent board software at Dy 4 Systems prior to shipment, so if you've received the IXA7 with IXAtools you will not need to update the software in the flash. However, if you are receiving IXAtools as an update, you may need to upgrade the flash. The release notes included with IXAtools will direct you as to what files need to be updated. IXAtools always includes a complete set of the most recent flash programs that are installed with the Board Configuration Utility in the `/etc` subdirectory.

Reprogramming the flash memory will typically be done with the Board Configuration Utility. This utility is described in Chapter 8. Capability to program the flash is also provided in HostAPI, IOPlusAPI, and IXAbsp libraries. The use of these libraries for burning flash requires an application program to be developed that calls the functions with the appropriate file parameters.

Installing VxWorks BSPs

The installation process will place the BSPs for the IOPlus and the SPEs under the directory you specify as your Tornado base directory path ($WIND_BASE) under the paths:

```
$WIND_BASE/target/config/iop          IOPlus BSP
$WIND_BASE/target/config/spe7x        SPE BSP
```

You will also specify the directory for the IXAtools libraries: libioplus_api.a, libixaio.a, and libixabsp.a, to be installed. To build VxWorks with the BSPs and these libraries you will have to be sure the environmental variables $IOPLUS and $WIND_BASE are defined as the IXAtools and the Tornado base directories.

On-line manual pages are provided with the BSPs. In order to conveniently access these from the Tornado Help menu add the following code to the list of BSPs referenced in the $WIND_BASE/docs/BSP_Reference.html file:

```
<p><i><a href="./vxworks/bsp/iop.html">iop<a></i></p>
<p><i><a href="./vxworks/bsp/spe7x.html">spe7x<a></i></p>
```

Chapter 6 describes the IOPlus BSP and its use, and Chapter 8 describes the SPE BSP and its use.

Chapter 3: Hardware Architecture

3.1 Introduction

The IXA7 is composed of a number of major board components, which are depicted in Figure 3.1. This chapter will discuss each one of these components: their operation and their inter-operability with other board components.

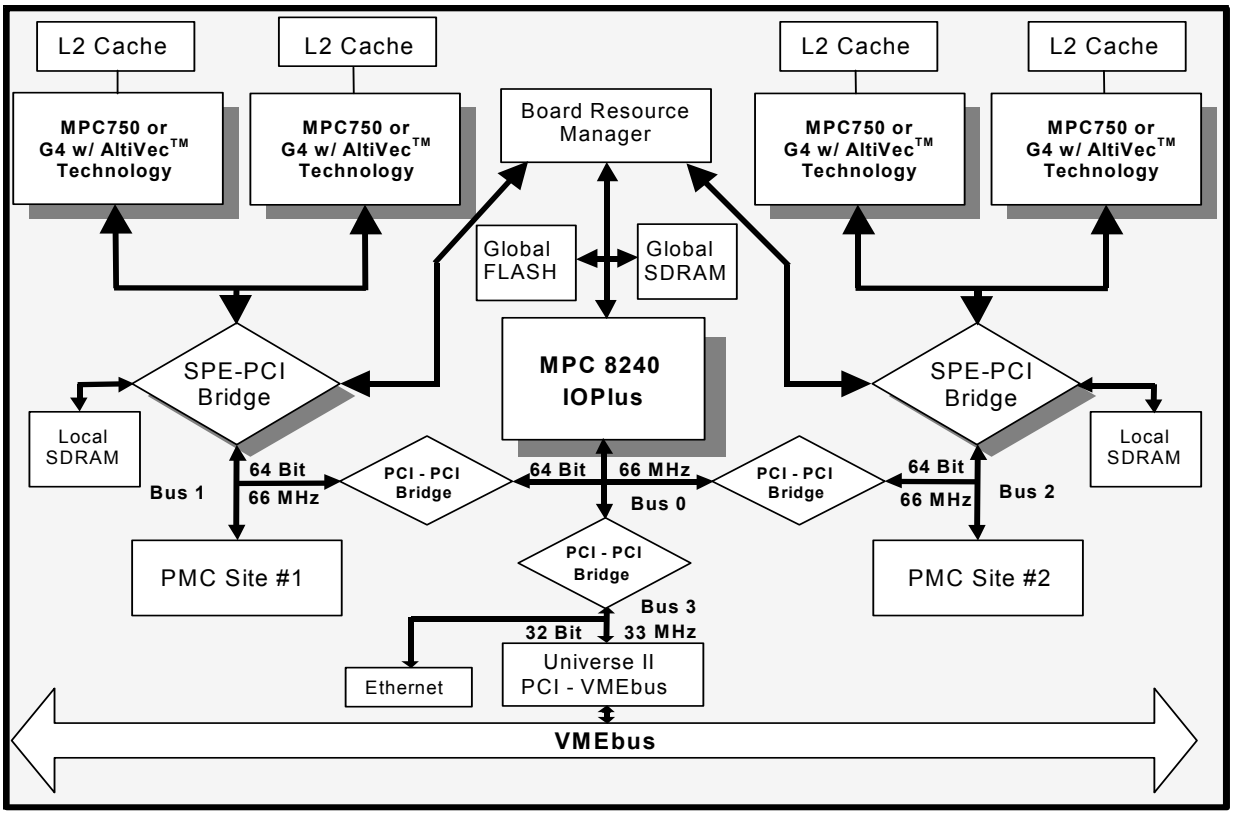


Figure 3.1 - IXA7 Board Architecture

3.2 IOPlus

The IOPlus consists of a Motorola MPC 8240 integrated processor. This is a PowerPC 603e core with the following integrated features:

Integrated 66 MHz 32-bit PCI bus interface, including PCI arbitration
Integrated 100 MHz 64-bit SDRAM controller, supporting up to 1 GByte of external memory
Integrated 2-channel DMA controller

Enhanced Programmable Interrupt Controller (EPIC)
Flash memory controller (8-bit)
Port X Interface – Generic programmable interface for register or peripheral support
Generation of Type 0 and Type 1 PCI configuration cycles
Supports Big and Little Endian bus modes

The IOPlus is code compatible with existing flavors of the 603e processor. The IOPlus is used for the following tasks:

Initialization of memory
Initialization of VMEbus interface
Initialization of all PCI bridges
Initialization of PMC modules
Loading and configuration of firmware devices
Command and status for VMEbus interface
Initialization of the SPEs
Communication with the SPEs

The power up initialization and run-time support programs for the IOPlus are written into flash memory prior to shipment. User's can also add application software to the IOPlus run-time programs. To use the resources of the IXA7 via the IOPlus software, it is important to not overwrite or disable the IOPlus run-time software. Chapter 5 details how the IOPlus can be commanded by the SPEs to perform a number of useful services.

3.3 SPEs

The IXA7 can be ordered with 750 or 7400 Signal Processing Elements (SPEs). They are configured as paired clusters with SPE A and SPE B sharing a PCI/memory bridge, and SPE C and SPE D sharing a PCI/memory bridge.

Each SPE is loaded with common boot code by the IOPlus upon board reset and started. This code permits the paired SPE's to operate out of a shared local memory space. Chapter 8 provides more information on how the common boot code facilitates this and how to write applications for it.

3.4 SPE-PCI Bridge

There are two SPE-PCI bridges on an IXA7, one for the SPE A and SPE B cluster and one for the SPE C and SPE D cluster. The primary function of the bridge is to interface two SPE's to the PCI bus, the local SDRAM, and to the board resources available on the Port X.

The SPE-PCI bridge has the following features:

32-bit PCI V2.1 compliant interface with a frequency of 66 MHz
Full PCI Target / Initiator functionality

Two 32-byte (one cache line) PCI to memory write buffers
Two 16-byte processor to PCI write buffers
One 32-byte (one cache line) processor read from PCI buffer
Interrupt controller logic
Two DMA channels
Four high resolution timers
100 MHz SDRAM interface
100 MHz 60x bus interface for connection to the SPE processors
Port X interface for connection to the Board Resource Manager
Support for memory coherency

3.5 Universe II

The IXA7 uses a Universe II VMEbus – PCI bus bridge. This bridge has the following features:

Support for VME 64 transfers (up to ~60 Mbytes/sec)
33 MHz 64-bit PCI interface
Address translation registers (VME PCI)
DMA controller function, including linked list buffers
Interrupt and semaphore support
Big / little endian conversion support

The IOPlus configures the Universe II at power up. Chapter 5 describes how the IOPlus can be commanded to change the Universe II configuration and perform VME data transfers. Chapter 4 shows the memory mapping of the Universe II control registers so that developers can modify the interface either locally or across the VMEbus. It is important to make sure the Universe II performs endian conversion on data so that the IXA7 PCI bus is operating in little endian and the VMEbus is operating in big endian.

3.6 PCI Local Bus

The IXA7 contains four local PCI buses each isolated by PCI bridges. The Primary PCI bus, PCI bus #0 in Figure 3.1, is a 64 bit, 66Mhz bus that hosts the IOPlus and the three bridges (connected to the primary PCI interface on the bridges). On the other side of the three bridges are secondary buses. PCI bus #3 is dedicated to servicing the Universe II. PCI bus #1 and PCI bus #2 serve the two SPE clusters and the two PMC sites. The PMC sites are addressed later in this chapter. Since all the key components of the IXA7 live in PCI space, the PCI memory map of the board plays an important role in the programing and data movement of the IXA7. This map is presented in Chapter 4.

Major features of the PCI-PCI bridges:

Three 66 MHz, 64-bit PCI-PCI bridges providing bus isolation and transaction forwarding

Primary side allows 64-bit, 66 MHz transactions
Secondary side allows 66 or 33 MHz PCI busses
Conversion between 32/64 bit data and 33/66 MHz clock domains
Accepts Type 0 and Type 1 configuration accesses
Supports a uniform linear system address map
Performs clocking, reset, and arbitration for the secondary bus

3.7 Interrupt Controller

The Interrupt Controller is an FPGA that provides interrupt, LED, and VME Geographical Address access for the IXA7. The device has three isolated Port X interfaces: one connected to the IOPlus and one connected to each of the SPE-PCI bridges. This provides a means of inter-processor interrupts, and board resource utilization that is independent of the PCI bus.

The Interrupt Controller provides the following capabilities:

Interrupt support functions such as routing, masking, and clearing,
LED control for each SPE's status LED,
VME Geographical Addressing support.

The Interrupt Controller is controlled by a group of addressable registers that are defined in Chapter 4.

3.8 Board Semaphores

Board semaphores are provided via registers in the Universe II. These registers are read/write registers. On a hardware reset all bits are cleared. Semaphores can only be set when they are zero. Writing zero to a semaphore register clears the semaphore. Writing a processor-unique nonzero value requests the semaphore. If the semaphore was zero at the time of the write operation, the semaphore retains the write value. If the semaphore was nonzero, then it is currently unavailable, and the write request has no effect. The software must read back the semaphore and compare the read value to the write value. Semaphores do not generate interrupts.

Semaphores are used to access critical board resources. As such many requestors may want a particular semaphore. The first requestor to actually write its value into the semaphore gains control of the resource. Once a requestor has written its value it must read the semaphore back to see if it has control of the resource.

Of the eight Universe semaphores available, the first four are reserved for DY 4 SYSTEMS library functions, and should not be used by applications.

Dy 4 Systems provides test-and-set library functions that are built upon semaphores. These functions are available for applications programs.

3.9 Global Memory

The IXA7 can have up to 64 Mbytes of SDRAM that is used by the IOPlus for local memory and a global memory area for the board. This memory has a 100Mhz clock rate and is 64/32/16 and 8-bit accessible. The page size is 2048 bytes and multiple open pages are allowed. The global memory is available to the SPE's, VMEbus, IOPlus, and any other devices that can reach the PCI local bus. See Table 4.1 in Chapter 4 for the address range of the global and IOPlus local portions.

3.10 SPE Local Memory

The IXA7 can be populated with up to 128 Mbytes of SDRAM per SPE cluster. The SDRAM bus runs at 100MHz. IXA7 SDRAM is located in the SPE's local address space, which starts at 0x00000000. The page size is 2048 bytes and multiple open pages are allowed. The SDRAM is 64/32/16 and 8-bit accessible and acesses to it are managed by the SPE-PCI Bridge. The bridge will arbitrate accesses by the two SPE's in the cluster as well as PCI accesses from the other board devices.

Since the SDRAM is shared by a SPE cluster, applications must map into the shared space in a controlled manner so that each processor sees the space appropriately. The PowerPC expects its exception vector table to be mapped at address 0x00000000 and therefore presents a complication when two processors are trying to make use of the same space. IXAtools has solved this problem with Common Boot Code (CBC) that is loaded to the SPEs by the IOPlus at startup. Chapter 7 covers the operation and use of the CBC.

When writing applications that run on each processor which share the local SDRAM, the user must be careful in the method used to access the memory in order to achieve maximum data transfer performance. This is particularly true if both processors access memory while at the same time, data is being read or written to the local memory over the PCI bus. If both processors are polling on local SDRAM locations using a tight loop (three or four assembly instructions), PCI transations to local memory may be continually retried because the PCI transation cannot break in. This is especially true if both processors are polling on locations in separate pages of the SDRAM. If polling cannot be avoided, PCI bus snooping may be turned on to elevate the priority of the PCI transation. However, with snooping on, an extra address only cycle occurs on the SPE local 60x bus for every PCI transaction that affects the 60x bus performance.

If both processors run instructions out of local SDRAM from separate pages, and snooping is not turned on, PCI retries will also occur. In this case, it is suggested to run out of instruction cache as much as possible.

> ! Caution: PCI accesses to local SDRAM may be continually retried (fail) if user code on the SPE processors associated with the SDRAM accessed by the PCI bus "hogs" the memory interface.

3.11 Flash Memory

The IXA7 provides up to 16 Mbytes of 90 ns flash memory that is connected to the IOPlus. It is used to store initialization code, IXA7 configuration data, and user application code. Flash memory can be accessed from the IOPlus, the SPEs, and from a VME host computer using routines contained in IXAtools. Chapters 5 - 9 present a number of different approaches to command the IOPlus to set flash values. IXAtools contains a utility that provides a user-friendly interface for programming flash memory. See Chapter 8 for more information on this utility.

3.12 PMC Sites

The IXA7 provides two PCI Mezzanine Card (PMC) sites: PMC Site 1 and PMC site 2. The PMC Site 1 and the A/B SPE-PCI Bridge are both connected to the Secondary PCI Bus 1. The PMC Site 2 and the C/D SPE-PCI Bridge are both connected to the Secondary PCI Bus 2. See Figure 3.1.

The PMC sites are designed to conform to the following specifications with the exceptions noted below:

PCI Local Bus Specification Revision 2.2, 18 December 1998.
Draft Processor PMC Standard (VITA 32-199x) Draft 0.3, October 30, 1999.
Draft Standard Physical and Environmental Layers for PCI Mezzanine Cards: PMC; IEEE P1386.1 Draft 2.1; October 11, 1999.
Draft Standard for a Common Mezzanine Card Family: CMC; Draft 2.0, April 4,1995.
PMC-P4 Pin Out Mapping to VME-P0 and VME64x-P2 Draft Standard (VITA 35-199x), DRAFT 0.3, 30 June 1999.
For Conduction Cooled IXA7 Only: Conduction Cooled PCI Mezzanine Card CCPMC; (VITA 20-199x) Draft Standard; Draft 1.11, November, 1999

A PMC can be a 3.3 V board (uses 3.3 V signaling), a 5 V board (uses 5 V signaling), or a universal board (autoselects and configures for the signaling level used by the PMC Host). Each PMC card is required to have a keying hole to indicate the type of PCI signaling it utilizes. If the PMC's PCI bus utilizes 3.3 V PCI signaling, it shall provide a 3.3 V keying hole. If the PMC's PCI bus utilizes 5 V PCI signaling, it shall provide a 5 V keying hole. If both voltages can be supported, then both keying holes shall be provided. Unfortunately, not all PMCs conform to the PMC standard.

Each PMC site on the IXA7 board, PMC Site 1 or PMC Site 2, can be factory configured to support either 3.3 V or 5 V signaling on their PCI bus. Both PMC sites can be configured to support the same signaling levels or differant signaling levels. Each PMC site will have a keying pin installed to indicate the signaling level for which that PMC sight is configured. When the PMC site is configured for 5 V PCI signaling, the 5 V keying pin will be installed between the four PMC connectors, Jn1, Jn2, Jn3, and Jn4 (where n is the respective PMC site number). When the PMC site is configured for 3.3 V PCI signaling, the 3.3 V keying pin will be installed below the Jn1 and Jn3 connectors.

When a PMC manufacturer does not provide the required keying hole, the keying pin on the IXA7 PMC site must be removed in order to install the PMC. If this is done, extreme care must be exercised to prevent the wrong type of PMC from being installed on the PMC site. Installing a PMC with incompatible signaling levels can cause permanent damage to the IXA7 baseboard as well as the PMC.

> ! WARNING: Each PMC site is factory configured for either 3.3 V or 5 V signaling bus levels (both PMC sites on the IXA board can be configured the same or differently). Do not install a 5 V PMC on a 3.3 V PMC site. Do not install a 3.3 V PMC on a 5 V site. Installing incompatible PMCs can permanently damage the IXA baseboard and/or the PMC.

The PMC Sites support PMCs implementing 64-bit or 32-bit PCI bus widths. The PMC sites are capable of operating at 66 MHz or 33 MHz. Installing a 66 MHz PMC will cause the respective secondary PCI bus to operate at 66 MHz. Installing a 33 MHz PMC will cause the respective secondary PCI bus to operate at 33 MHz. The primary PCI bus will always operate at 66 MHz no matter what PMCs are installed.

The commercial IXA7 boards support front panel user I/O and backplane user I/O through the P2 and P0 VMEbus connectors. The Rugged (Conduction Cooled) IXA7 boards only support backplane user I/O. The backplane I/O for PMC Site 1 is routed out the IXA7 VMEbus P0 connector. The backplane I/O for the PMC Site 2 is routed out the IXA7 VMEbus P2 connector. Table 3.1 shows the backplane I/O connections for PMC sites 1 and 2.

Table 3.1 - PMC to VME Connector Mapping

PMC Jn4 Connector	PMC Site 1 Connections (VMEbus Connector Pin)	PMC Site 2 Connections (VMEbus Connector Pin)
1	P0E-4	P2C-1
2	P0D-4	P2A-1
3	P0C-4	P2C-2
4	P0B-4	P2A-2
5	P0A-4	P2C-3
6	P0E-5	P2A-3
7	P0D-5	P2C-4
8	P0C-5	P2A-4
9	P0B-5	P2C-5
10	P0A-5	P2A-5
11	P0E-6	P2C-6
12	P0D-6	P2A-6
13	P0C-6	P2C-7
14	P0B-6	P2A-7
15	P0A-6	P2C-8
16	P0E-7	P2A-8
17	P0D-7	P2C-9
18	P0C-7	P2A-9
19	P0B-7	P2C-10
20	P0A-7	P2A-10
21	P0E-8	P2C-11
22	P0D-8	P2A-11
23	P0C-8	P2C-12
24	P0B-8	P2A-12
25	P0A-8	P2C-13
26	P0E-12	P2A-13
27	P0D-12	P2C-14
28	P0C-12	P2A-14
29	P0B-12	P2C-15
30	P0A-12	P2A-15
31	P0E-13	P2C-16
32	P0D-13	P2A-16
33	P0C-13	P2C-17
34	P0B-13	P2A-17
35	P0A-13	P2C-18
36	P0E-14	P2A-18
37	P0D-14	P2C-19
38	P0C-14	P2A-19
39	P0B-14	P2C-20
40	P0A-14	P2A-20

Table 3.1 - PMC to VME Connector Mapping (cont.)

PMC Jn4 Connector	PMC Site 1 Connections (VMEbus Connector Pin)	PMC Site 2 Connections (VMEbus Connector Pin)
41	P0E-15	P2C-21
42	P0D-15	P2A-21
43	P0C-15	P2C-22
44	P0B-15	P2A-22
45	P0A-15	P2C-23
46	P0E-16	P2A-23
47	P0D-16	P2C-24
48	P0C-16	P2A-24
49	P0B-16	P2C-25
50	P0A-16	P2A-25
51	P0E-17	P2C-26
52	P0D-17	P2A-26
53	P0C-17	P2C-27
54	P0B-17	P2A-27
55	P0A-17	P2C-28
56	P0E-18	P2A-28
57	P0D-18	P2C-29
58	P0C-18	P2A-29
59	P0B-18	P2C-30
60	P0A-18	P2A-30
61	P0E-19	P2C-31
62	P0D-19	P2A-31
63	P0C-19	P2C-32
64	P0B-19	P2A-32

The PMC BUSMODE[4:2] signals are hardwired on the IXA7 and constantly drive the following values: BUSMODE[4:2] = 0b001. This signals the PMC card that it is connected to a PMC site. If the card is a PMC card, it will begin driving its interface signals and will drive a logic 0 on the BUSMODE1 signal. Note that the IXA7 receives the BUSMODE1 signals from each PMC but does not check its value.

The IXA7 deviates from the PMC specification by utilizing two PMC Reserved pins for an alternate function. The I^2C interface of the IOPlus is connected to the following pins:

Pn2 pin 55 is connected to the I^2C signal SDA

Pn2 pin 57 is connected to the I^2C signal SCL.

Do not install a PMC that connects a signal to these pins as it could damage the IXA7 board or the PMC.

! WARNING: All PMCs installed on the IXA7 board must not utilize the reserved PMC Jn2 pins 55 and 57. Shorting these signals or driving these signals can cause damage to the IXA7 board and/or the PMC.

The IXA7 provides support for the PPMC standard. The PPMC support includes: 66 MHz signaling capability (M66EN signal implemented), Optional Second PCI Agent capability (IDSELB, REQB#, and GNTB# signals implemented), support for the RESET_OUT# and EREADY signals. The following signals are supported on the IXA7 board:

The MONARCH# signal is a new signal on Jn2 pin 64. The IXA7 baseboard is the monarch. To signal this to connected PMCs, the MONARCH# signals on each PMC site are not connected, allowed to float high.

M66EN signal is on Jn2 pin 47. When grounded by the PMC, it indicates 33 MHz PCI bus operation. When the PMC allows this pin to float high, it indicates 66 MHz operation.

The IDSELB signal is a new signal on Jn2 pin 34. It is used to select an optional second PCI agent.

The REQB# signal is a new signal on Jn2 pin 52. This signal is a request issued by the optional second PCI agent requesting the owership of the PCI bus.

The GNTB# signal is a new signal on Jn2 pin 54. This signal is a grant issued to the optional second PCI agent requesting the ownership of the PCI bus via the corresponding REQB# signal.

The RESET_OUT# signal is a new signal on Jn2 pin 60. This signal is an open drain output from the PMC. When asserted by the PMC, the IXA7 will perform a board reset (same as pushing the reset switch on the IXA7 board).

The EREADY signal is a new signal on Jn2 pin 58. This signal is an open drain output on non-monarch PPMCs that indicates the PMC has completed its on-board initialization and can respond to PCI bus enumeration. The IXA7 configuration software will not perform enumeration on the respective PMC until the PMC releases this signal.

The IXA7 provides support for an Optional Second PCI Agent. Non-monarch PPMCs may include an optional second PCI agent. The IXA7 only supports one load per PMC PCI signal when operating at 66 MHz. If a PPMC places more than one load on any of the PCI signals, then it must ground its M66EN pin to cause the respective secondary PCI bus to operate at 33 MHz.

! WARNING: The IXA7 only allows a PMC to place one load on each PCI signal when operating at 66 MHz. Failure to adhere to this requirement can cause timing violations on the affected signals, resulting in data loss and or corruption.

Power Requirements

The IXA7 supplies +5 V, 3.3 V, +12 V, -12 V power to the PMC sites. The maximum current for each supply is shown in Table 3.2. The VIO voltage can be factory configured for 5 V or 3.3 V signaling. The maximum power dissipation allowed for each PMC site is 10 Watts.

Table 3.2 - PMC Maximum Current Loads

Supply	Max. Supply Current
5 V	2 A
3.3 V	3 A
+ 12 V	500 mA
- 12 V	100 mA

3.13 VMEbus Interface

The IXA7 is fully compliant with the VITA standard VME64x specification. The board can function as a slot one controller or a slave. Tables 3.3, 3.4, and 3.5 give the signal mappings for the P0, P1, and P2 connectors respectively.

Table 3.3 - P0 Connector Definitions

Pin #	Rows				
	A	B	C	D	E
1	GTDO	GTDI	GTRST	GTCK	GTMS
2	3AD1	PTRST	PSRST	PTCK	PTDO
3	3AD0	PCSTPO	PHRST	PTMS	PTDI
4	1PMC5	1PMC4	1PMC3	1PMC2	1PMC1
5	1PMC10	1PMC9	1PMC8	1PMC7	1PMC6
6	1PMC15	1PMC14	1PMC13	1PMC12	1PMC11
7	1PMC20	1PMC19	1PMC18	1PMC17	1PMC16
8	1PMC25	1PMC24	1PMC23	1PMC22	1PMC21
9	GTDO2	JTRST	JSRST	KTCK	KTDO
10	SCL	SDA	JHRST	KTMS	KTDI
11	P0_INT1	XTCK	XTMS	XTDO	XTDI
12	1PMC30	1PMC29	1PMC28	1PMC27	1PMC26
13	1PMC35	1PMC34	1PMC33	1PMC32	1PMC31
14	1PMC40	1PMC39	1PMC38	1PMC37	1PMC36
15	1PMC45	1PMC44	1PMC43	1PMC42	1PMC41
16	1PMC50	1PMC49	1PMC48	1PMC47	1PMC46
17	1PMC55	1PMC54	1PMC53	1PMC52	1PMC51
18	1PMC60	1PMC59	1PMC58	1PMC57	1PMC56
19	P0_INT0	1PMC64	1PMC63	1PMC62	1PMC61

1. 1PMC* identifies signals routed from PMC Site 1, * is the corresponding PMC J14 pin number.
2. G[xxx] are the Global JTAG signals with GTDO and GTDO2 being the TDO signals before and after the PMCs, which are the last in the chain.
3. P[xxx] are the Processor JTAG signals.
4. J[xxx] and K[xxx] are the additional JTAG signals required for the IOPlus.
5. 3[xxx] are signals from segment 3 of the PCI bus.
6. SCL and SDA are the two I^2C serial signals
7. P0_INT0 and P0_INT1 are inputs to the board interrupt multiplexer. P0_INT1 is not available if a transition module is installed.

Table 3.4 - P1 Connector Definitions

Pin #	Rows				
	Z	A	B	C	D
1	NC	D00	BBSY*	D08	+5 Volts
2	GND	D01	BCLR*	D09	GND
3	NC	D02	ACFAIL*	D10	NC
4	GND	D03	BG0IN*	D11	NC
5	NC	D04	BG0OUT*	D12	NC
6	GND	D05	BG1IN*	D13	NC
7	NC	D06	BG1OUT*	D14	NC
8	GND	D07	BG2IN*	D15	NC
9	NC	GND	BG2OUT*	GND	GAP
10	GND	SYSCLK	BG3IN*	SYSFAIL*	GA0
11	NC	GND	BG3OUT*	BERR*	GA1
12	GND	DS1*	BR0*	SYSRESET*	+3.3 Volts
13	NC	DS0*	BR1*	LWORD*	GA2
14	GND	WRITE*	BR2*	AM5	+3.3 Volts
15	NC	GND	BR3*	A23	GA3
16	GND	DTACK*	AM0	A22	+3.3 Volts
17	NC	GND	AM1	A21	GA4
18	GND	AS*	AM2	A20	+3.3 Volts
19	NC	GND	AM3	A19	NC
20	GND	IACK*	GND	A18	+3.3 Volts
21	NC	IACKIN*	NC	A17	NC
22	GND	IACKOUT*	NC	A16	+3.3 Volts
23	NC	AM4	GND	A15	NC
24	GND	A07	IRQ7*	A14	+3.3 Volts
25	NC	A06	IRQ6*	A13	NC
26	GND	A05	IRQ5*	A12	+3.3 Volts
27	NC	A04	IRQ4*	A11	NC
28	GND	A03	IRQ3*	A10	+3.3 Volts
29	NC	A02	IRQ2*	A09	NC
30	GND	A01	IRQ1*	A08	+3.3 Volts
31	NC	-12 Volts	NC	+12 Volts	GND
32	GND	+5 Volts	+5 Volts	+5 Volts	+5 Volts

Note: GAP – GA4 are for geographical addressing support

Table 3.5 - P2 Connector Definition

Pin #	Rows				
	Z	A	B	C	D
1	3GNT1	2PMC2	+5 Volts	2PMC1 / 3AD0*	3RST
2	GND	2PMC4	GND	2PMC3 / 3AD1*	3AD30
3	3CLK	2PMC6	NC	2PMC5 / INT1*	3AD29
4	GND	2PMC8	A24	2PMC7 / SDA*	3AD31
5	3REQ1	2PMC10	A25	2PMC9 / SCL*	3AD27
6	GND	2PMC12	A26	2PMC11 / PWRON*	3AD26
7	3AD28	2PMC14	A27	2PMC13 / PQREQ*	3AD24
8	GND	2PMC16	A28	2PMC15	3AD25
9	3AD23	2PMC18	A29	2PMC17	3CBE3
10	GND	2PMC20	A30	2PMC19	3AD20
11	3AD22	2PMC22	A31	2PMC21	3AD21
12	GND	2PMC24	GND	2PMC23	3AD18
13	3AD19	2PMC26	+5 Volts	2PMC25	3AD17
14	GND	2PMC28	D16	2PMC27	3FRAME
15	3AD16	2PMC30	D17	2PMC29	3IRDY
16	GND	2PMC32	D18	2PMC31	3TRDY
17	3CBE2	2PMC34	D19	2PMC33	3DEVSEL
18	GND	2PMC36	D20	2PMC35	3STOP
19	3SERR	2PMC38	D21	2PMC37	3PERR
20	GND	2PMC40	D22	2PMC39	3CBE1
21	3PAR	2PMC42	D23	2PMC41	3AD13
22	GND	2PMC44	GND	2PMC43	3AD15
23	3AD14	2PMC46	D24	2PMC45	3AD12
24	GND	2PMC48	D25	2PMC47	3AD11
25	3AD10	2PMC50	D26	2PMC49	3AD8
26	GND	2PMC52	D27	2PMC51	3AD9
27	3CBE0	2PMC54	D28	2PMC53	3AD7
28	GND	2PMC56	D29	2PMC55	3AD2
29	3AD6	2PMC58	D30	2PMC57	3AD5
30	GND	2PMC60	D31	2PMC59	3AD3
31	3AD4	2PMC62	GND	2PMC61	GND
32	GND	2PMC64	+5 Volts	2PMC63	+5 Volts

* These seven pins on P2-C may be factory-configured for either set of signals. This allows PCI and serial usage without a P0 connector, as well as providing signals for future expansion.

1. 2PMC* identifies signals routed from PMC Site 2, * is the corresponding PMC J24 pin number.
2. G[xxx] are the Global JTAG signals with GTDO and GTDO2 being the TDO signals before and after the PMCs, which are the last in the chain.
3. P[xxx] are the Processor JTAG signals.
4. J[xxx] and K[xxx] are the additional JTAG signals required for the IOPlus.
5. 3[xxx] are signals from segment 3 of the PCI bus.
6. SCL and SDA are the two I^2C serial signals
7. INT1 is not available if the transition module is installed.

Chapter 4: Memory Maps

4.1 Introduction

This chapter presents memory maps for the IOPlus, the SPEs, and the global memory. (Note that since the IOPlus can access the SPE memory, the SPE memory maps are of interest to the IOPlus.) Table 4.1 presents the board memory map as viewed by the IOPlus and the PCI bus.

The board's PCI bus is represented as 4 segments in Tables 4.1 - 4.3. These segments correspond to the PCI buses that exist between the PCI bridges. Bus 0 corresponds to the bus that connects the IOPlus to the three PCI-PCI bridges. Bus 1 is for PMC site #1 and the SPE A/B cluster. Bus 2 is for PMC site #2 and the SPE C/D cluster. Bus 3 is for the Universe II PCI interface. The actual bus numbers that get assigned during the board's initialization may differ if a PMC site is populated with a PMC device that contains PCI bridges that further segment the PCI bus. In that case, the device buses will pick up the extra sequences in the numbers at that location. For example a PMC device that has three local buses and that is located in the PMC site #1, will contain bus 2, 3, & 4 with bus 5 being the new number for PMC site #2.

Since all the components of the IXA7 are connected together by the PCI buses, effectively, all the memory on the board is globally accessible. When this manual uses the term "global memory", it is refering to the SDRAM local to the IOPlus. This is done because part of this memory area is used as a global resource for IOPlus / SPE communication.

In viewing Table 4.1 you'll notice many "holes" in the PCI and IOPlus address maps. Some of these will be filled with future upgrades of the memory, and others are unreachable regions by design.

The memory map of the IXA7 may look a little intimidating at first, but the capability to address many of the boards features, and the ability to make the maps work for you, make the IXA7 a unique solution.

4.2 IOPlus Memory Map

Table 4.1 presents the memory map for the IOPlus.

Table 4.1 - IOPlus-Local and PCI Memory Map

IOPlus Address Range	Size	Description	IOPlus Description	Bus	PCI Address Generated
00000000 – 07FFFFFF	128 MB	Global Memory	System Memory	N/A	None
08000000 – 3FFFFFFF	896 MB	Reserved		N/A	None
40000000 – 7FFFFFFF	1 GB	IOPlus Reserved		N/A	None
80000000 – 9FFFFFFF	512 MB	PMC Site 1	PCI Memory Space	1	80000000 – 9FFFFFFF
A0000000 – A7FFFFFF	128 MB	SPE AB SDRAM			A0000000 – A7FFFFFF
A8000000 – B5EFFFFF	223 MB	Reserved			A8000000 – B5EFFFFF
B5F00000 – B5FFFFFF	1 MB	SPE AB Embedded Utilities Memory Block			B5F00000 – B5FFFFFF
B6000000 – B7FFFFFF	32 MB	Bus 1 Miscellaneous Devices			B6000000 – B7FFFFFF
B8000000 – BFFFFFFF	128 MB	Dy 4 Systems Reserved		0	00000000 – 07FFFFFF
C0000000 – DFFFFFFF	512 MB	PMC Site 2		2	C0000000 – DFFFFFFF
E0000000 – E7FFFFFF	128 MB	SPE CD SDRAM			E0000000 – E7FFFFFF
E8000000 – F5EFFFFF	223 MB	Reserved			E8000000 – F5EFFFFF
F5F00000 – F5FFFFFF	1 MB	SPE CD Embedded Utilities Memory Block			F5F00000 – F5FFFFFF
F6000000 – F6FFFFFF	16 MB	Bus 2 Miscellaneous Devices			F6000000 – F6FFFFFF
F7000000 – F7FFFFFF	16 MB	Ethernet Paddleboard and Bus 3 Miscellaneous Devices		3	F7000000 – F7FFFFFF
F8000000 – FBFFFFFF	64 MB	VME Address Space			F8000000 – FBFFFFFF
FC000000 – FCFFFFFF	16 MB	Universe II Registers			FC000000 – FCFFFFFF
FD000000 – FD0FFFFF	1 MB	IOPlus Embedded Utilities Memory Block	Available ONLY to the IOPlus	N/A	
FD100000 – FDFFFFFF	15 MB	PCI/ISA Memory Space	PCI/ISA Memory Space		FD100000 – FDFFFFFF
FE000000 – FEBFFFFF	12 MB	PCI I/O Space	PCI I/O Space	N/A	None
FEC00000 – FEDFFFFF	2 MB	Configuration Address Register	IOPlus Registers		
FEE00000 – FEEFFFFF	1 MB	Configuration Data Register			
FEF00000 – FEFFFFFF	1 MB	PCI Interrupt Acknowledge			
FF000000 – FF7FFFFF	8 MB	EPLD Registers, FPGA Interrupt Mux (Port X)	RCS1- FLASH/ROM	N/A	None
FF800000 – FFDFFFFF	6 MB	Reserved	Not Used		None
FFE00000 – FFFFFFFF	2 MB	Boot FLASH (Port X)	RCS0- FLASH/ROM		None

4.3 SPE Memory

Tables 4.2 and 4.3 present the memory maps for the SPEs while Table 4.4 presents a summary of the PCI memory map.

Table 4.2 - SPE A/B Cluster Memory Map

SPE Local Address	Size	Description	Local Description	Bus	PCI Address Generated
00000000 – 07FFFFFF	128 MB	SPE AB SDRAM	Local System Memory	N/A	None
08000000 – 3FFFFFFF	896 MB	Not Used			None
40000000 – 77FFFFFF	896 MB	SPE-PCI Bridge Reserved			None
78000000 – 7BFFFFFF	64 MB	64-bit Extended Flash/ROM	Not Used (RCS3)		None
7C000000 – 7FFFFFFF	64 MB	64-bit Extended Flash/ROM	Not Used (RCS2)		None
80000000 – 9FFFFFFF	512 MB	PMC Site 1	PCI Memory Space	1	80000000 – 9FFFFFFF
A0000000 – A7FFFFFF	128 MB	Illegal Block		N/A	None
A8000000 – B5EFFFFF	223 MB	Reserved		1	A8000000 – B5EFFFFF
B5F00000 – B5FFFFFF	1 MB	Illegal Block		N/A	None
B6000000 – B7FFFFFF	32 MB	Bus 1 Miscellaneous Devices		1	B6000000 – B7FFFFFF
B8000000 – BFFFFFFF	128 MB	Global SDRAM (must use SPE_PCI Bridge Outbound Translation Register to translate address to global memory)		0	00000000 – 07FFFFFF (After Boot and Initialization)
C0000000 – DFFFFFFF	512 MB	PMC Site 2		2	C0000000 – DFFFFFFF
E0000000 – E7FFFFFF	128 MB	SPE CD SDRAM			E0000000 – E7FFFFFF
E8000000 – F5EFFFFF	223 MB	Reserved			E8000000 – F5EFFFFF
F5F00000 – F5FFFFFF	1 MB	SPE CD Embedded Utilities Memory Block			F5F00000 – F5FFFFFF
F6000000 – F6FFFFFF	16 MB	Bus 2 Miscellaneous Devices			F6000000 – F6FFFFFF
F7000000 – F7FFFFFF	16 MB	Ethernet Paddleboard and Bus 3 Miscellaneous Devices.		3	F7000000 – F7FFFFFF
F8000000 – FBFFFFFF	64 MB	VME Address Space			F8000000 – FBFFFFFF
FC000000 – FCFFFFFF	16 MB	Universe II Registers			FC000000 – FCFFFFFF
FD000000 – FD0FFFFF	1 MB	Local Address of SPE AB Embedded Utilities Memory Block	Available ONLY to SPE AB at this Address		None
FD100000 – FDFFFFFF	15 MB	PCI/ISA Memory Space	PCI/ISA Memory Space	N/A	FD100000 – FDFFFFFF
FE000000 – FEBFFFFF	12 MB	PCI I/O Space	PCI I/O Space	N/A	None
FEC00000 – FEDFFFFF	2 MB	Configuration Address Register	SPE-PCI Bridge Registers		
FEE00000 – FEEFFFFF	1 MB	Configuration Data Register			
FEF00000 – FEFFFFFF	1 MB	PCI Interrupt Acknowledge			
FF000000 – FF7FFFFF	8 MB	FPGA Interrupt Mux. (Port X)	RCS1 – FPGA Access	N/A	None
FF800000 – FFEFFFFF	7 MB	Not Used	Not Used		None
FFF00000 – FFF01FFF	8 KB	Boot Area (must use SPE-PCI Bridge Outbound Translation Register to translate address to global memory)	Global SDRAM	0	40000 – 41FFF (At Boot Time Only; This range is Programmable in the SPE-PCI Bridge)
FFF02000 – FFFFFFFF	1016 KB	Not Used		N/A	FFF02000 – FFFFFFFF

Table 4.3 - SPE C/D Cluster Memory Map

SPE Local Address	Size	Description	Local Description	Bus	PCI Address Generated
00000000 – 07FFFFFF	128 MB	SPE CD SDRAM	Local System Memory	N/A	None
08000000 – 3FFFFFFF	895 MB	Not Used			None
40000000 – 77FFFFFF	896 MB	SPE_PCI Bridge Reserved			None
78000000 – 7BFFFFFF	64 MB	64-bit Extended Flash/ROM	Not Used (RCS3)		None
7C000000 – 7FFFFFFF	64 MB	64-bit Extended Flash/ROM	Not Used (RCS2)		None
80000000 – 9FFFFFFF	512 MB	PMC Site 1	PCI Memory Space	1	80000000 – 9FFFFFFF
A0000000 – A7FFFFFF	128 MB	SPE AB SDRAM			A0000000 – A7FFFFFF
A8000000 – B5EFFFFF	223 MB	Reserved			A8000000 – B5EFFFFF
B5F00000 – B5FFFFFF	1 MB	SPE AB Embedded Utilities Memory Block			B5F00000 – B5FFFFFF
B6000000 – B7FFFFFF	32 MB	Bus 1 Miscellaneous Devices			B6000000 – B7FFFFFF
B8000000 – BFFFFFFF	128 MB	Global SDRAM (must use SPE-PCI Bridge Outbound Translation Register to translate address to global memory)		0	00000000 – 07FFFFFF (After Boot and Initialization)
C0000000 – DFFFFFFF	512 MB	PMC Site 2		2	C0000000 – DFFFFFFF
E0000000 – E7FFFFFF	128 MB	Illegal Block		N/A	
E8000000 – F5EFFFFF	223 MB	Reserved		2	E8000000 – F5EFFFFF
F5F00000 – F5FFFFFF	1 MB	Illegal Block		N/A	
F6000000 – F6FFFFFF	16 MB	Bus 2 Miscellaneous Devices		2	F6000000 – F6FFFFFF
F7000000 – F7FFFFFF	16 MB	Ethernet Paddleboard and Bus 3 Miscellaneous Devices		3	F7000000 – F7FFFFFF
F8000000 – FBFFFFFF	64 MB	VME Address Space			F8000000 – FBFFFFFF
FC000000 – FCFFFFFF	16 MB	Universe II Registers			FC000000 – FCFFFFFF
FD000000 – FD0FFFFF	1 MB	Local Address of SPE CD Embedded Utilities Memory Block	Available ONLY to SPE CD at this Address		None
FD100000 – FDFFFFFF	15 MB	PCI/ISA Memory Space	PCI/ISA Memory Space	N/A	FD100000 – FDFFFFFF
FE000000 – FEBFFFFF	12 MB	PCI I/O Space	PCI I/O Space	N/A	Reserved
FEC00000 – FEDFFFFF	2 MB	Configuration Address Register	SPE-PCI Bridge Registers		
FEE00000 – FEEFFFFF	1 MB	Configuration Data Register			
FEF00000 – FEFFFFFF	1 MB	PCI Interrupt Acknowledge			
FF000000 – FF7FFFFF	8 MB	FPGA Interrupt Mux. (Port X)	RCS1 – FPGA Access	N/A	None
FF800000 – FFEFFFFF	7 MB	Not Used	Not Used		None
FFF00000 – FFF01FFF	8 KB	Boot Area (must use SPE-PCI Bridge Outbound Translation Register to translate address to global memory)	Global SDRAM	0	40000 – 41FFF (At Boot Time Only; This Range is Programmable in the SPE-PCI Bridge)
FFF02000 – FFFFFFFF	1016 KB	Not Used		N/A	FFF02000 – FFFFFFFF

Table 4.4 - PCI Memory Map

PCI Address Range	Size	Description	Bus
00000000 - 07FFFFFF	128 MB	Global Memory	
08000000 - 7FFFFFFF	1920 MB	Not Used	0
80000000 – 9FFFFFFF	512 MB	SPE AB PMC	
A0000000 – A7FFFFF	128 MB	SPE AB SDRAM	
A8000000 – B5EFFFFF	223 MB	Reserved	
B5F00000 – B5FFFFFF	1 MB	SPE AB Embedded Utilities Memory Block	1
B6000000 – B7FFFFFF	32 MB	Bus 1 Miscellaneous Devices	
B8000000 – BFFFFFFF	128 MB	Not Used	
C0000000 – DFFFFFFF	512 MB	SPE CD PMC	
E0000000 – E7FFFFF	128 MB	SPE CD SDRAM	
E8000000 – F5EFFFFF	223 MB	Reserved	
F5F00000 – F5FFFFFF	1 MB	SPE CD Embedded Utilities Memory Block	2
F6000000 – F6FFFFFF	16 MB	Bus 2 Miscellaneous Devices	
F7000000 – F7FFFFFF	16 MB	Ethernet Paddleboard and Bus 3 Miscellaneous Devices	3
F8000000 – FBFFFFFF	64 MB	VME Address Space	
FC000000 – FCFFFFFF	16 MB	Universe II Registers	

4.4 Global Memory

The global memory block is the SDRAM memory that is local to the IOPlus. This memory is partitioned into several sections, some which are reserved for the IOPlus to use and others, which are available to the SPEs and the VMEbus. The IOPlus does not control the allocation of the global memory area available for the SPE and VMEbus usage. It is up to the developer to prevent one application from overwriting another application's space. The partitioning of the global memory area is shown in Table 4.5. There are two sections that are reserved for the IOPlus, one for user applications, and one section containing the Board Information Structure (BIS) that is created by the IOPlus but globally available.

Board Information Structure

The board information structure contains a collection of board configuration and status information. This structure is used by IXAtools. Memory mapping for the board information structure is provided for those customers developing their own interfaces. Two values in the structure are of significant value to the developer. They are the address of the host list table, and the address of the master list address table.

The board information structure is located at PCI address 0x30000. Both the IOPlus and the SPEs can access this structure. The structure is stored in "big endian" mode.

The host list table's address is located at an offset of 0x138 and the master list address table's address is located at an offset of 0x13C from the board information structure's address.

Table 4.5 - Global Memory Map

PCI/IOP Address Range	Size	Description	Notes
00000000 – 0002FFFF	192 KB	IOPlus Reserved	
00030000 – 00030FFF	4 KB	Board Information Structure	Fixed address
00031000 – 0007FFFF	316 KB	IOPlus Reseved	
00080000 – 00FFFFFF	15.5 MB	Used by VxWorks, if installed	VxWorks treat this memory space as reserved if VxWorks is running on the IOPlus. Otherwise, this area is available to the developer.
01000000 – 03FFFFFF	48 MB	User area	Currently the global memory is only populated with 64 M. Space exists in the map for 128 MB

4.5 VME Memory

The IXA7 is configured to provide VMEbus memory windows into the IXA7 memory space. Table 4.6 shows the three provided VME windows. Using A32 accesses the board information structure and the global memory area can be reached. Using A32 accesses the configuration registers of the Universe II can also be reached.

There are 8 slave mapping registers in the Universe II of which the IOPlus uses the first two for the board information structure and the global memory area. A VME host processor can use these mapping registers to move VME address windows around the IXA7 PCI memory space (the HostAPI function *host_map_resource* performs this function). To use these mapping registers directly, consult the Universe II manual. There are also 8 master mapping registers in the Universe II. IXAtools provides functions for the configuration and use of these registers when the IXA7 is performing master transfers.

Each space is located at an offset from the VME base address of the board (for instance, if the VME base address is 0x80000000, then the board information structure is located at 0x800A0000).

Table 4.6 - VME Memory Map

Offset from VME base address	Address size	Privileged / Non-Privileged	Program / Data	Description
0x00000000	A32	Both	Both	Universe II Configuration Registers
0x000A0000 – 0x000AFFFF	A32	Both	Both	Board Information Structure (corresponds to 0x30000 in PCI memory space)
0x01000000 – 0x02FFFFFF	A32	Both	Both	User global memory (corresponds to 0x40000 in PCI memory space)

4.6 Interrupt Controller Register Map

The Interrupt Controller occupies 8 MB of the IOPlus and SPE-PCI bridge memory maps in the range 0xff00_0000 and 0xff7f_ffff. Tables 4.7 and 4.8 present the registers available in that address range from the AB cluster's and CD cluster's respective window into that space. Table 4.9 presents the registers available to the IOPlus in that space.

Table 4.7 – Interrupt Controller Memory Map for Cluster AB

Processor A/B Address	Register Type	Description
FF000000	Read/Write	Processor A INT Mask 1 Register
FF000008	Read/Write	Processor A INT Mask 2 Register
FF000010	Read/Write	Processor B INT Mask 1 Register
FF000018	Read/Write	Processor B INT Mask 2 Register
FF000020	Read Only	AB General Interrupt Status Register
FF000028	R (Bits 0-6) W (Bit 7)	AB Interrupt Status 1/LED A Control Register
FF000030	R (Bits 0-6) W (Bit 7)	AB Interrupt Status 2/LED B Control Register
FF000038	Read Only	AB Universe Interrupt Status Register
FF000040	Read/Write	Interrupt Status and Clear A Register
FF000048	Read/Write	Interrupt Status and Clear B Register
FF000050	Write Only	Processor Interrupt Generation Register A
FF000058	Write Only	Processor Interrupt Generation Register B
FF000060	Read/Write	AB Test Register
FF000068	Read/Write	Processor A INT Mask 3 Register
FF000070	Read/Write	Processor B INT Mask 3 Register
FF0000C0	Read/Write	User Interrupt Detection Select Register (USER_SEL_AB)

Table 4.8 – Interrupt Controller Memory Map for Cluster CD

Processor C/D Address	Register Type	Description
FF000000	Read/Write	Processor C INT Mask 1 Register
FF000008	Read/Write	Processor C INT Mask 2 Register
FF000010	Read/Write	Processor D INT Mask 1 Register
FF000018	Read/Write	Processor D INT Mask 2 Register
FF000020	Read Only	CD General Interrupt Status Register
FF000028	R (Bits 0-6) W (Bit 7)	CD Interrupt Status 1/LED C Control Register
FF000030	R (Bits 0-6) W (Bit 7)	CD Interrupt Status 2/LED D Control Register
FF000038	Read Only	CD Universe Interrupt Status Register
FF000040	R/W	Interrupt Status and Clear C Register
FF000048	R/W	Interrupt Status and Clear D Register
FF000050	Write Only	Processor Interrupt Generation Register C
FF000058	Write Only	Processor Interrupt Generation Register D
FF000060	Read/Write	CD Test Register
FF000068	Read/Write	Processor C INT Mask 3 Register
FF000070	Read/Write	Processor D INT Mask 3 Register
FF0000C0	Read/Write	User Interrupt Detection Select Register (USER_SEL_AB)

Table 4.9 – Interrupt Controller Memory Map for the IOPlus

IOPlus Address	Register Type	Description
FF400000	Read Only	IOPlus General Interrupt Status Register
FF400008	Read Only	IOPlus Universe Interrupt Status Register
FF400010	Read Only	IOPlus Status 1 Register
FF400018	Read Only	IOPlus Status 2 Register
FF400020	Read/Write	Processor Interrupt Status and Clear Register K
FF400028	Write Only	Processor Interrupt Generation Register K
FF400030	Read/Write	IOPlus Interrupt Mask Register (High Order)
FF400038	Read/Write	IOPlus Interrupt Mask Register (Low Order)
FF400040	Read/Write	Reserved
FF400048	Read/Write	Reserved
FF400050	Read/Write	Reserved
FF400058	Read/Write	Reserved
FF400060	Read/Write	User Interrupt Detection Select Register (USER_SEL_AB)
FF400068	Read/Write	Reserved
FF400070	Read/Write	Reserved
FF400078	Read/Write	Reserved
FF400080	Read Only	VME Geographical Address Register
FF400088	Read/Write	IOPlus Test Register
FF400090	Read	Board Resource Manager Revision ID Register

Mask Registers

The Interrupt Mask Registers are used to route certain interrupt sources to interrupt outputs. The Interrupt Controller provides five maskable interrupt outputs. Four of the masked outputs go to the SPEs (one per processor) while one of the outputs in an input to the IOPlus. Each SPE processor is provided with three INT interrupt mask registers that are accessible locally (i.e. no pci bus transactions are necessary) while the IOPlus processor is provided with two locally accessable mask registers.

The interrupt mask registers are shown in Figures 4.1 through 4.4. Figure 4.1 shows the IOPlus controlled mask registers (which are accessible only to the IOPlus) while Figure 4.2, Figure 4.3 and Figure 4.4 show the INT interrupt mask registers that are only accessible by the SPE processors. In Figure 4.2 note that bit 7 of the Processor X INT Mask is a global mask for processor interrupts targeted for processor X. Setting this bit will allow any of the other processors on the IXA card to interrupt processor X.

Figure 4.1 - Interrupt Mask Register Format for CPE (IOPlus) processor

High Order Register

Bit	15	14	13	12	11	10	09	08
	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	CCD_ INT_EN	CCD_PCI_ INT_EN
Bit	07	06	05	04	03	02	01	00
	CAB_ INT_EN	CAB_PCI_ INT_EN	USER_ INT1_EN	USER_ INT0_EN	PMC2_B/D _EN	PMC2_A/C _EN	PMC1_B/D _EN	PMC1_A/C_ EN

Low Order Register

Bit	15	14	13	12	11	10	09	08
Low Register	INT5_EN	INT4_EN	INT3_EN	INT2_EN	INT1_EN	INT0_EN	Reserved	Reserved
Bit	07	06	05	04	03	02	01	00
	LINT7_EN	LINT6_EN	LINT5_EN	LINT4_EN	LINT3_EN	LINT2_EN	LINT1_EN	LINT0_EN

(Note that this register is split into a high and low order register in order to accommodate all the bits needed)

Figure 4.2 - INT Interrupt Mask 1 Register Format for SPE processors

Processor X INT Mask 1 Register (where X is A,B,C, or D)

Bit	07	06	05	04	03	02	01	00
	Global PIGR Mask	Reserved	USER_ INT1_EN	PMC2_B/D_ EN	PMC2_A/C_ EN	USER_ INT0_EN	PMC1_B/D_ EN	PMC1_A/C_ EN

Figure 4.3 - INT Interrupt Mask 2 Register Format for SPE processors

Processor X INT Interrupt Mask 2 Register (where X is A or B)

Bit	07	06	05	04	03	02	01	00
	INT5_EN	INT4_EN	INT3_EN	INT2_EN	INT1_EN	INT0_EN	CAB_ PCI_EN	CAB_ INT_EN

Processor X INT Interrupt Mask 2 Register (where X is C or D)

Bit	07	06	05	04	03	02	01	00
	INT5_EN	INT4_EN	INT3_EN	INT2_EN	INT1_EN	INT0_EN	CCD_ PCI_EN	CCD_ INT_EN

Figure 4.4 - INT Interrupt Mask 3 Register Format for SPE processors

Processor X INT Interrupt Mask 3 Register (where X is A, B, C, or D)

Bit	07	06	05	04	03	02	01	00
	LINT7_EN	LINT6_EN	LINT5_EN	LINT4_EN	LINT3_EN	LINT2_EN	LINT1_EN	LINT0_EN

Table 4.10 - Bit Field Definitions for the Interrupt Mask Registers

Bit Field	Description: Enable Signal for:
PMC1_A/C_EN	Cluster AB PMC interrupts A & C
PMC1_B/D_EN	Cluster AB PMC interrupts B & D
PMC2_A/C_EN	Cluster CD PMC interrupts A & C
PMC2_B/D_EN	Cluster CD PMC interrupts B & D
INT(5-0)_EN	Processor Interrupt Identification bits
CAB_PCI_EN	PCI Interrupt from AB SPE-PCI bridge
CCD_PCI_EN	PCI Interrupt from CD SPE PCI bridge
CAB_INT_EN	Primary Interrupt from AB SPE-PCI bridge
CCD_INT_EN	Primary Interrupt from CD SPE-PCI bridge
USER_INT0_EN	User Interrupt 0
USER_INT1_EN	User Interrupt 1 (from Ethernet Board)
LINT[2:0]_EN	Universe II Interrupts LINT[7:0]

The definitions of Mask Register bit fields are provided in Table 4.10. All mask registers are read/write registers. On hardware reset all bits are cleared. A one in any bit will enable that interrupt source. Multiple interrupt sources can be forwarded to a single output. A single source can also be used to generate multiple interrupts although, in this case, the application software running on the processors interrupted must properly coordinate the clearing of the interrupt if it is latched only at the source. Please note that

the PMC interrupts (PMC1_A/C_EN, PMC1_B/D_EN, PMC2_A/C_EN, and PMC2_B/D_EN) are wired or'ed in pairs. For example, PMC1_A/C_EN will allow interrupt A or C from PMC site 1 to activate an interrupt output.

The IOPlus Interrupt Status Registers reflect the status of the masked interrupts. The interrupt source is masked prior to being captured in the IOPlus Interrupt Status Registers. In contrast, the SPE Interrupt Status registers reflect the current status of the interrupt input. In other words, INT Interrupt Mask 1,2 and 3 Registers do not affect the values latched in the SPE Interrupt Status Registers. This means that when a SPE processor is interrupted, it must read the interrupt status registers with the foreknowledge of the interrupt mask configuration that it set.

NOTE: By enabling the appropriate mask bit and writing to the PIGR register, a processor may interrupt any other processor on the board, including itself.

WARNING: The SPE interrupt Status Registers reflect the current status of the interrupt inputs, not the masked version of these interrupts.

Processor Interrupt Generation Register

The external interrupts are not latched in the Board Resource Manager since they are latched at the source. However, the interrupts generated from Processor Interrupt Generation registers (PIGR) are latched. The PIGR has two primary fields to consider. A three-bit Processor ID field defines the processor to be interrupted and a three-bit field which defines the Processor Interrupt Identification value. To use the register, a six-bit value is written to the PIGR. This causes the Interrupt Status and Clear Register to be updated for the target processor. The Interrupt Status and Clear Register for the target processor reflects the decoded Processor Identification value. For instance, SPE A writing a binary 0001_0011 to its PIGR register causes Processor Identification bit 3 to be set in SPE B Interrupt Status and Clear register. Note that the processor which origininated the interrupt is not evident by the value in the Interrupt Status and Clear Register. See Figure 4.5 for further details.

Since the Processor interrupts are latched, the user application must clear these interrupts by writing to the appropriate Processor Interrupt Status and Clear register. For the example described above, processor B receives an interrupt from processor A, with a status value reflecting INT3 asserted in the Interrupt Status and Clear Register. The user must write a one to this bit to clear the interrupt. The Processor Interrupt Generation Registers are shown in Figure 4.5.

Processor Interrupt Generation Register A (PIGR_A; available to SPE AB only)

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	P2	P1	P0	INT2	INT1	INT0

Processor Interrupt Generation Register B (PIGR_B; available to SPE AB only)

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	P2	P1	P0	INT2	INT1	INT0

Processor Interrupt Generation Register C (PIGR_C; available to SPE CD only)

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	P2	P1	P0	INT2	INT1	INT0

Processor Interrupt Generation Register D (PIGR_D; available to SPE CD only)

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	P2	P1	P0	INT2	INT1	INT0

Processor Interrupt Generation Register K (PIGR_K; available to IOPlus only)

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	P2	P1	P0	INT2	INT1	INT0

The bit definitions are:

P2-0 Processor Identification Bits
000 Interrupt IOPlus
001 Interrupt SPE A
010 Interrupt SPE B
011 Interrupt SPE C
100 Interrupt SPE D
101 Reserved
110 Interrupt SPE A,B,C,D
111 Interrupt SPE A,B,C,D,IOPlus

INT2-0 Interrupt Identification Bits decoded to bits 5:0 in the Target Processor Interrupt Status and Clear Register

Figure 4.5 - Processor Interrupt Generation Registers

Interrupt Status

There are three different register types for interrupt status: Interrupt Status Registers, Interrupt Status and Clear Registers and Interrupt Status/LED Control Registers.

1) Interrupt Status Registers per Port X interface:
   - SPE AB:
     - AB General Interrupt Status Register
     - AB Universe Interrupt Status Register
   - SPE CD:
     - CD General Interrupt Status Register
     - CD Universe Interrupt Status Register
   - IOPlus:
     - IOPlus General Interrupt Status Register
     - IOPlus Universe Interrupt Status Register
     - IOPlus Status 1 Register
     - IOPlus Status 2 Register

2) Interrupt Status and Clear Registers per Port X interface:
   - SPE AB:
     - Processor A Interrupt Status and Clear
     - Processor B Interrupt Status and Clear
   - SPE CD:
     - Processor C Interrupt Status and Clear
     - Processor D Interrupt Status and Clear
   - IOPlus: IOPlus Interrupt Status and Clear Register

3) Interrupt Status/LED Control Registers:
   - SPE AB:
     - AB Status 1/LED A Control Register
     - AB Status 2/LED B Control Register
   - SPE CD:
     - CD Status 1/LED C Control Register
     - CD Status 2/LED D Control Register

There are a total of eight Interrupt Status Registers, five Interrupt Status and Clear Registers, and four Interrupt Status/LED Control Registers. The Interrupt status registers for each SPE convey the same SPE interrupt information. These registers are used to determine the Processor Interrupt Identification. The General Interrupt Status Registers are read by each processor to determine the general interrupt source. Once this is determined, the processor reads the appropriate Interrupt Status Register (as specified by the active bit in the General Interrupt Status Register) to determine the specific interrupt source. The relationship between the bits in the General Interrupt Status Register and the other interrupt status registers is provided in Table 4.11. The IOPLUS GISR is also mapped to the external interrupt pins of the IOPlus (See Figure 4.8).

Table 4.11 - Relationship of General Interrupt Status Register Fields to Other Status Registers

General Interrupt Status Register Bit Field	Corresponding Status Register to Check
Universe	Universe Interrupt Status Register
Group 1	Status 1/LED Control Register
Group 2	Status 2/LED Control Register
A_INT	Processor A Interrupt Status & Clear Register
B_INT	Processor B Interrupt Status & Clear Register
C_INT	Processor C Interrupt Status & Clear Register
D_INT	Processor D Interrupt Status & Clear Register
K_INT	Processor K Interrupt Status & Clear Register

Each processor has an associated Interrupt Status and Clear Register and must read this register to determine Processor Interrupt Identification bits. This register is also used to clear processor interrupts by writing a binary one to clear a specific active interrupt. Bit 7 of the Interrupt Status and Clear Register indicates the processor cluster ID. Bit 7 is set to zero for processors A and B and one for processors C and D. Bit 7 is not writable.

The registers defined to convey interrupt status are illustrated in the Figures 4.6, 4.7, and 4.8. The figures are grouped according to processor access privileges.

The Status Registers are read-only registers while the Status and Clear Registers are read/write registers. On hardware reset all bits are cleared to zero.

AB General Interrupt Status Register

Bit	07	06	05	04	03	02	01	00
	Revision	Reserved	Reserved	B_INT	A_INT	Group 2	Group 1	Universe

where:

Universe: Source of Interrupt was Universe.
Group 1: Source of Interrupt was from SPE AB or User Interrupt 0
Group 2: Source of Interrupt was from SPE CD or User Interrupt 1.
A_INT: A Processor Interrupt occurred and is intended for processor A
B_INT: A Processor Interrupt occurred and is intended for processor B
Revision: 1 indicates support of multiple inter-processor interrupts

AB Universe Interrupt Status Register

Bit	07	06	05	04	03	02	01	00
	LINT7	LINT6	LINT5	LINT4	LINT3	LINT2	LINT1	LINT0

AB Status 1/LED A Control Register (Accessible to Processors A and B only)

Bit	07	06	05	04	03	02	01	00
	LED_A	Reserved	Reserved	USER INT0	PMC1 _B/D	PMC1 _A/C	CAB_PCI	CAB_INT

AB Status 2/LED B Control Register

Bit	07	06	05	04	03	02	01	00
	LED_B	Reserved	Reserved	USER INT1	PMC2 _B/D	PMC2 _A/C	CCD_PCI	CCD_INT

Processor A Interrupt Status and Clear Register

Bit	07	06	05	04	03	02	01	00
	Cluster ID 0	Reserved	INT5	INT4	INT3	INT2	INT1	INT0

Processor B Interrupt Status and Clear Register

Bit	07	06	05	04	03	02	01	00
	Cluster ID 0	Reserved	INT5	INT4	INT3	INT2	INT1	INT0

where:

INT5-0 Interrupt Identification Bits
One bit is set when a Processor Interrupt Occurs
Multiple bits set indicate that more than one processor Interrupt has occurred.

Figure 4.6 - SPE AB Status Registers

CD General Interrupt Status Register

Bit	07	06	05	04	03	02	01	00
	Revision	Reserved	Reserved	D_INT	C_INT	Group 2	Group 1	Universe

where:

Universe: Source of Interrupt was Universe.
Group 1: Source of Interrupt was from SPE AB or User Interrupt 0
Group 2: Source of Interrupt was from SPE CD or User Interrupt 1.
C_INT: A Processor Interrupt occurred and is intended for processor C
D_INT: A Processor Interrupt occurred and is intended for processor D
Revision: 1 indicates support of multiple inter-processor interrupts

CD Universe Interrupt Status Register

Bit	07	06	05	04	03	02	01	00
	LINT7	LINT6	LINT5	LINT4	LINT3	LINT2	LINT1	LINT0

CD Status 1/LED C Control Register

Bit	07	06	05	04	03	02	01	00
	LED_C	Reserved	Reserved	USER INT0	PMC1 _B/D	PMC1 _A/C	CAB_PCI	CAB_INT

CD Status 2/LED D Control Register

Bit	07	06	05	04	03	02	01	00
	LED_D	Reserved	Reserved	USER INT1	PMC2 _B/D	PMC2 _A/C	CCD_PCI	CCD_INT

Processor C Interrupt Status and Clear Register

Bit	07	06	05	04	03	02	01	00
	Cluster ID 1	Reserved	INT5	INT4	INT3	INT2	INT1	INT0

Processor D Interrupt Status and Clear Register

Bit	07	06	05	04	03	02	01	00
	Cluster ID 1	Reserved	INT5	INT4	INT3	INT2	INT1	INT0

where:

INT5-0 Interrupt Identification Bits

Figure 4.7 - SPE CD Status Registers

IOPlus General Interrupt Status Register

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	Reserved	Reserved	K_INT	Group 2	Group 1	Universe

where:

Universe: Source of Interrupt was Universe.
Group 1: Source of Interrupt was from SPE AB or User Interrupt 0
Group 2: Source of Interrupt was from SPE CD or User Interrupt 1.
K_INT: A Processor Interrupt occurred and is intended for IOPlus

IOPlus Universe Interrupt Status Register

Bit	07	06	05	04	03	02	01	00
	LINT7	LINT6	LINT5	LINT4	LINT3	LINT2	LINT1	LINT0

IOPlus Status 1 Register

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	Reserved	USER INT0	PMC1B/D	PMC1A/C	CAB_PCI	CAB_INT

IOPlus Status 2 Register

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	Reserved	USER INT1	PMC2B/D	PMC2A/C	CCD_PCI	CCD_INT

IOPlus Interrupt Status and Clear Register

Bit	07	06	05	04	03	02	01	00
	Reserved	Reserved	INT5	INT4	INT3	INT2	INT1	INT0

where:

INT5-0 Interrupt Identification Bits

The GISR register is mapped to the external interrupt pins of the IOPLUS.
When GISR[3] is set to one, external interrupt 0 asserts
When GISR[2] is set to one, external interrupt 1 asserts
When GISR[1] is set to one, external interrupt 2 asserts
When GISR[0] is set to one, external interrupt 3 asserts

Figure 4.8 - IOPlus Interrupt Status Registers

Processor Status LED Control

The Board Resource Manager contains bits for controlling the SPE Processor Status LEDs. The bits for LED control are contained in the SPE AB and SPE CD Interrupt Status/LED Control Registers defined in Figure 4.6 and Figure 4.7. The LEDs are active after configuration of the Board Resource Manager by the boot code.

TEST Registers

There is one Read/Write test register for each Port X interface. These registers are provided to facilitate Port X validation. They are intended for test use only. The Test Register for each Cluster Port X interface is eight bits wide while the Test Register for the IOPlus Port X interface is 16 bits wide.

VME Geographical Address Register

This 16 bit read only register allows the software to read the VME Geographical Addressing Pins.

The format of the VME Geographical Address Register (VGAR) is as shown in Figure 4.9. This register may only be accessed by the IOPlus.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Not Used										GAP	GA(4:0)				

Figure 4.9 - VME Geographical Address Register Format

GA(4:0) VME Geographical Address pins.
GAP VME Geographical Address Parity pin.

The Geographical Address (GA) pin signals are pulled up on the IXA7 board. When the GA pins are read through the VGAR, the value read is the inverted value of the slot number into which the board is plugged. Table 4.12 illustrates the inversion of the slot number. The GAP bit is used to provide odd parity. If the board was powered without being plugged into a backplane, the slot number will be zero with a parity error (see No Slot in Table 4.12).

Table 4.12 - VME Geographical Address Assignments

Slot No.	GAP	GA4	GA3	GA2	GA1	GA0
No Slot	1	1	1	1	1	1
1	1	1	1	1	1	0
2	1	1	1	1	0	1
3	0	1	1	1	0	0
4	1	1	1	0	1	1
5	0	1	1	0	1	0
6	0	1	1	0	0	1
7	1	1	1	0	0	0
8	1	1	0	1	1	1
9	0	1	0	1	1	0
10	0	1	0	1	0	1
11	1	1	0	1	0	0
12	0	1	0	0	1	1
13	1	1	0	0	1	0
14	1	1	0	0	0	1
15	0	1	0	0	0	0
16	1	0	1	1	1	1
17	0	0	1	1	1	0
18	0	0	1	1	0	1
19	1	0	1	1	0	0
20	0	0	1	0	1	1
21	1	0	1	0	1	0

User Interrupts (Applies to Rev. C-12 and Rev. E-4 boards)

User interrupts 0 and 1 (USER_INT0 and USER_INT1) from the rear panel can now be programmed to allow edge detection of a pulse provided that the pulse is 60 nS or longer. While the default mode for these interrupts is level detect, they can be changed to be edge sensitive by programming one of three USER_SEL_XX Registers. Three read/write USER_SEL_XX registers have been added to the interrupt controller, one for Cluster AB, one for Cluster CD, and one for the IOPlus. The location of these registers in the memory map is shown in Table 1.3 thru 1.5. These registers may be programmed independently, meaning that a processor in cluster AB can set a USER_INT to be edge sensitive while a processor in the other cluster or from the IOPlus can set the same interrupt to be level sensitive. After making a change to the detection mode for a USER_INT, the user must make sure that the USER_INT is cleared in the corresponding Interrupt Status Register since changing the detection mode of a USER_INT can cause a spurious interrupt to appear in the status register. For example, if processor A sets USER_INT1 to be edge sensitive (in USER_SEL_AB), the user must make sure that bit 4

in the AB Interrupt Status 2 Register is clear. If it is not, the user should write a one to this bit to clear this interrupt.

!Warning: After making a change to the detection mode of a USER_INTX, the user must make sure that the USER_INTX is cleared in the corresponding Interrupt Status Register since changing the detection mode of a USER_INTX can cause a spurious interrupt to appear in the status register.

Table 4.13 – Format of USER_SEL_XX Registers

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Reserved	Reserved	USER1_SEL[1]	USER1_SEL[0]	USER0_SEL[1]	USER0_SEL[0]

Table 4.14 –USER_SEL Register Settings

Settings for User Select Bits		
UserN_Sel[1]	UserN_Sel[0]	Description
0	0	Default Condition – User Interrupt signal is detected when low level
0	1	User Interrupt signal is detected when high level
1	0	User Interrupt signal Is detected on a high to low transition (falling edge). This event is captured and held in the interrupt status register.
1	1	User Interrupt signal is detected on a low to high transition (rising edge). This event is captured and held in the interrupt status register.

Please note the following concerning USER_INT0 and USER_INT1:

1. These interrupts are independently configurable
2. The design changes to the interrupt controller are backward compatible
3. There are no changes to the way these interrupts are masked and mapped to the General Interrupt Status register.
4. When these interrupts are configured to edge detection mode, software must write a one to the active user interrupt bit location in the Interrupt Status 1 or Interrupt Status 2 registers to clear them. As before, Interrupt Status Register 1, bit 4 indicates User Interrupt 0 and Interrupt Status Register 2, bit 4 indicates User Interrupt 1. As in previous boards, level detected interrupts are cleared at the source of the interrupt.

Chapter 5: Using the IOPlus SmartDMA

5.1 Introduction

The IOPlus implements a feature called "SmartDMA". This feature allows the IOPlus to act as a smart DMA controller for programs running on the SPEs, asynchronously moving data between on- and off-board resources, generating interrupts, and handling interrupts. The SPEs control the IOPlus SmartDMA by creating linked lists (or chains) of commands, which the SmartDMA engine then processes. This chapter will first present an overview of how the IOPlus SmartDMA works, and then describe how to use this feature of the IOPlus.

5.2 SmartDMA Overview

An SPE program creates a command chain consisting of a linked list of one or more SmartDMA commands in either global or local memory. The command chain can be created outside of the program's main processing loop to minimize chain creation overhead. The SPE interrupts the IOPlus to indicate that the IOPlus should start processing the command chain. As SmartDMA processes each command in the chain, it creates a chain of responses that indicate the completion status of each command. This response chain can be parsed by the SPE to determine the completion status of each command.

SmartDMA implements many advanced capabilities. These include the ability to handle multiple chains simultaneously, either round robin or atomically (atomic processing can be specified individually for each link in the chain). Circular chains can be created to cause SmartDMA to continuously perform a sequence of operations. A single command chain can be halted upon SPE command. Command chains can be configured to halt processing when an error is encountered, or to continue processing with the next command. Response chains can be configured to contain a response for each command, or only for commands that had errors. Command and response chains can be located either in global memory or in SPE local memory.

SmartDMA works whether or not VxWorks is running on the IOPlus. If the IOPlus is running VxWorks, SmartDMA is implemented as a task named "tIopTask". This task is idle until an SPE creates and starts a command chain. If the IOPlus is running the "Dy 4 Systems Runtime Code", then the IOPlus is dedicated to running SmartDMA. Note that SmartDMA is a service that the IOPlus provides to SPE programs only. If you need to use SmartDMA on the IOPlus, use the "IOPlus_api" library, which is described in the next chapter of this manual.

The IXAbsp library (libixa.a) contains functions for managing SmartDMA command chains. Each command chain is attached to an IOPlus command channel, which contains pointers to a command chain, response chain, and channel management data. The IOPlus

defines a fixed number of command channels that can be attached to command chains by either SPEs or by host programs (using HostAPI). The IXAbsp library provides functions to open and close command chains (which attach the chains to channels), to build command chains, to start and stop command chain processing, and to check the status of command chains.

A command chain consists of one or more command packets and can be located either in global memory or SPE local memory. As SmartDMA processes each command packet, it adds a response packet to the response chain. Like the command chain, the response chain consists of one or more response packets and can be located either in global memory or SPE local memory. The locations of the command and response chains are specified by the IXAbsp function call that opens a command channel. The command and response chains are located independently of each other, i.e. both can reside in global memory, both in SPE local memory, or one in global memory and the other in SPE local memory.

By default, the control structure for command channels is located in global memory. When opening a command channel, you have the option of relocating this command structure into SPE memory. The advantage of relocating the command channel is that the SPE processor has complete control of the command channel without having to go out onto the PCI bus to access the channel control structure. The disadvantage is a very slight decrease in command processing performance, since the IOPlus must access the channel control structure through the PCI-PCI bridges.

Command channels are designed so that a program can open any number of channels provided they are available. Most likely, you will want to configure all the command channels needed in the initialization sequence, start the processing loop, and kick off command channels in response to events as they occur. Thus the overhead of setting up the command channels will not be incurred during the time-critical processing loop.

All commands and responses are formatted into standard data packets. These packets consist of a packet header and packet data. The packet header consists of five 32-bit words. The packet data is variable-length. The format of a packet is shown in Figure 5.1.

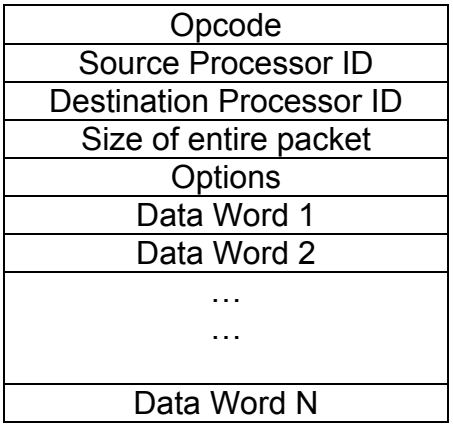


Figure 5.1 - Command Packet

Command packets are constructed using functions described below.

Additional information on the implementation of SmartDMA is contained in Appendix A of this manual.

5.3 When to Use SmartDMA

CHAMP boards provide a rich set of options for moving data between on-board resources, and between on- and off-board resources. For instance, there are four different functions that an SPE can use to move data from local memory to the PCI bus. These functions are `ixa_vme_write()`, `ixa_VME_dma_out()`, `ixa_cmd_VME_write()`, and a SmartDMA command chain. The first, `ixa_vme_write()`, causes the SPE to directly access the Universe II VME bridge, and transfer data across the VME bus. The SPE does this using a "memcpy"-like loop. The second, `ixa_VME_dma_out()`, configures the Universe II DMA controller to move the data using DMA. The third, `ixa_cmd_VME_write()`, constructs a command chain with a single command that instructs SmartDMA to move the data from local to VME memory. The fourth option is similar to the third, except that multiple commands can be connected into an elaborate chain, for processing by SmartDMA. Each function has advantages and disadvantages, depending on the circumstances. When should each function be used?

Having the SPE directly perform data movement (using `ixa_vme_write()`, for instance) is useful when latency is of primary importance. Since the SPE directly performs the transfer, there is no overhead of building or starting a command chain. The desired transfer begins immediately. The disadvantage of this approach is that it uses SPE processor cycles rather than happening in the background.

Having the SPE directly configure a DMA controller (using `ixa_VME_dma_out()`, for example), allows data movement to operate in the background while the SPE is performing other tasks. This is probably the best option for most situations.

Using SmartDMA to perform data movement (using `ixa_cmd_VME_write()`, for example), frees the SPE to perform other tasks while the data movement progresses. Thus, SPE processor cycles can be used to perform useful work. Another advantage is that this function is easy to use, compared to creating a SmartDMA command chain using the techniques described in this chapter. The disadvantage of using SmartDMA to perform the transfer is that there is a latency from the time that the SPE calls `ixa_cmd_start()` to start the SmartDMA transfer, until the transfer actually starts. This latency is caused by communications and command processing overhead in the SmartDMA engine. The result of this latency is that transfers do not start immediately, which can limit the effective data transfer rate. A second disadvantage of using functions such as `ixa_cmd_VME_write()` is that the rich options provided by SmartDMA are hidden in order to increase usability.

Using SmartDMA command chains to perform data movement has the same advantages and disadvantages as using functions such as `ixa_cmd_VME_write()` (which actually use a simple command chain to control SmartDMA). However, SmartDMA command chains can be used to create complicated sequences of commands, which are performed automatically by the SmartDMA engine. In effect, an IOPlus program is being created without actually writing any code that runs on the IOPlus.

The following chart summarizes when to use the various types of data movement:

Latency is more important than burning SPE processor cycles – the data transfer must start immediately.	Use SPE functions which directly perform the data movement, such as `ixa_vme_write()`
Saving SPE processor cycles is more important than latency. Parallelism is important.	Use SPE functions which directly manipulate a hardware DMA engine, or use SmartDMA.
Do not want to write code for the IOPlus, but want to use it to perform data movement while the SPE performs other tasks	Use SmartDMA
Want to build complicated command sequences that are processed asynchronously and automatically once started without writing IOPlus code.	Use SmartDMA

5.4 Using SmartDMA

An SPE program allocates a command channel using the *ixa_cmd_open* function. The parameters of this function include memory addresses where the command and response chains for the channel are located, and an options flag that indicates whether the chains and channel are located in global or SPE local memory.

The following basic example illustrates a simple data movement using SmartDMA. It is designed to illustrate several features of SmartDMA, but is not necessarily the best way to solve any specific problem. The following code segment opens a command channel, and places both the command and response chains in SPE local memory allocated using *calloc* (which fills the allocated memory with zeros):

```
CMD_CHANNEL   cmd_chan;
unsigned int *cmd1, *cmd2;
unsigned int *response;
IXABSP_STATUS stat;
unsigned int  cmd_chan_option;

cmd1 = (unsigned int *) calloc( CMD_MOVE_DATA_SZ,
```

```
                                    sizeof(unsigned int) );

cmd2 = (unsigned int *) calloc( CMD_GENERATE_INT_SZ,
                                    sizeof(unsigned int) );

response = (unsigned int ) calloc( 2*CMD_ACK_SZ,
                                       sizeof(unsigned int) );

cmd_chan_option = CMD_CHANNEL_OPTION_CMD_LOCAL |
                  CMD_CHANNEL_OPTION_RESPONSE_LOCAL |
                  CMD_CHANNEL_OPTION_RELOCATE;

stat = ixa_cmd_open( &cmd_chan,
                     cmd1,
                     response,
                     cmd_chan_option );
```

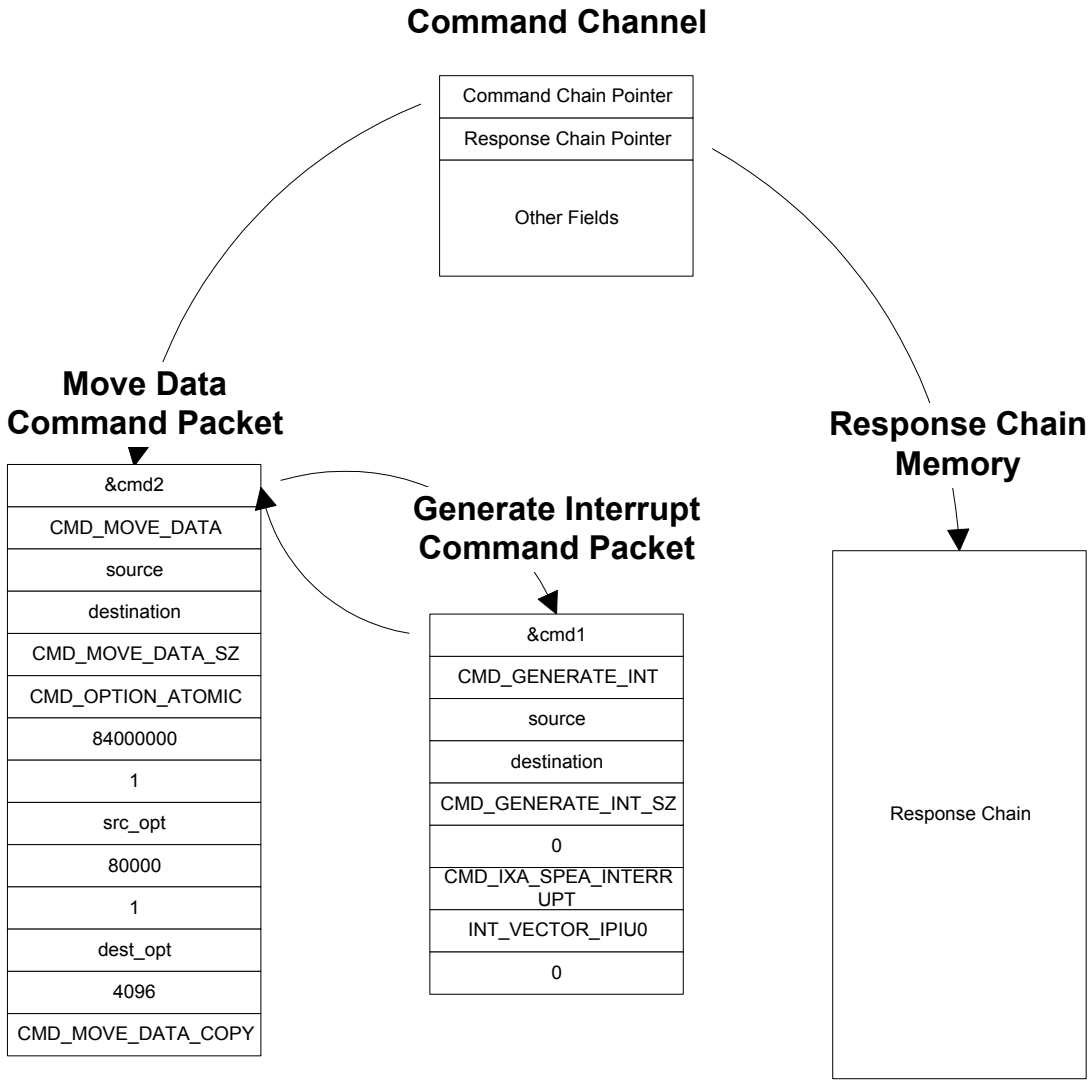
It is important to note that the *ixa_cmd_open* function does not allocate memory for the command or response chains. You must ensure that the addresses selected for these chains are already allocated and have sufficient memory available to hold command/response packets. In the above examples, sufficient memory was allocated for the response chain to hold two CMD_ACK response packets.

Once a command channel is opened, the command chain can be built using the *ixa_cmd_set* functions. Most *ixa_cmd_set* functions have 4 arguments - a pointer to the command channel structure, a command packet index, a command packet field id, and the value to be placed in that field. The *ixa_cmd_set_param* function has an additional parameter index argument. When building a command chain, care must be taken to create the chain in the proper order, starting with the first command packet in the chain and ending with the last. The next pointer field in a command packet is used to determine where the subsequent command in the chain is located, so it is particularly important to set the next pointer of a command packet before attempting to set fields in subsequent command packets. Failure to do so is likely to cause errors in processing the command chain and could result in corrupted memory. The next field of the last command packet in the chain must be set to zero if the command chain should terminate after processing the last command. In the example shown below, the “next” field is set to point back to “cmd1”, so that a loop is established. Note also that the next field requires a physical address rather than a logical address for command chains that are located in SPE local memory. The “ixa_mmu_map_addr()” function can be used to convert from logical to physical addresses.

It is also a good practice to set all of the parameter fields for a command packet, including those to be set to zero as there may be residual non-zero values in those memory locations that may cause unexpected results when the command is processed.

The following code segment builds the command chain shown in the following figure, which consiss of a “move data” command packet which transfers data from the VME bus to global memory followed atomically (i.e. SmartDMA will not process any commands from other command chains between the “Move Data” and “Generate Interrupt” commands) by a “generate interrupt” command packet which will trigger an interrupt:

Command Chain Created by Example Program



Here is the code segment which creates the command chain shown above:

```
src_opt = CMD_MOVE_DATA_VME_ADDR |
          CMD_MOVE_DATA_SUPER_DATA_AM |
          CMD_MOVE_DATA_A32 |
          CMD_MOVE_DATA_D32BLT;

dest_opt = CMD_MOVE_DATA_PCI_ADDR;
```

```
/* Move Data command packet */
ixa_cmd_set_opcode(&cmd_chan, 0, CMD_MOVE_DATA);
ixa_cmd_set_option(&cmd_chan, 0, CMD_OPTION_ATOMIC);
ixa_cmd_set_param( &cmd_chan, 0, 0, 0x84000000);    /* source address */
ixa_cmd_set_param( &cmd_chan, 0, 1, 1);             /* source stride  */
ixa_cmd_set_param( &cmd_chan, 0, 2, src_opt);       /* source options */
ixa_cmd_set_param( &cmd_chan, 0, 3, 0x80000);         /* dest address */
ixa_cmd_set_param( &cmd_chan, 0, 4, 1);               /* dest stride  */
ixa_cmd_set_param( &cmd_chan, 0, 5, dest_opt);        /* dest options */
ixa_cmd_set_param( &cmd_chan, 0, 6, 4096);              /* num words */
ixa_cmd_set_param( &cmd_chan, 0, 7, CMD_MOVE_DATA_COPY); /* operation */

/* Note that the IOPlus requires a physical address for the "next"
pointer, so convert the logical address to a physical address */
ixa_mmu_map_addr( (void *)cmd2, &phys_addr );
ixa_cmd_set_next(  &cmd_chan, 0, phys_addr);

/* Generate Interrupt command packet */
ixa_cmd_set_opcode(&cmd_chan, 1, CMD_GENERATE_INT);
ixa_cmd_set_option(&cmd_chan, 1, 0 );                 /* No options */
ixa_cmd_set_param(&cmd_chan, 1, 0, CMD_IXA_SPEA_INTERRUPT );/* SPE A*/
ixa_cmd_set_param(&cmd_chan, 1, 1, INT_VECTOR_IPIU0 ); /* User int */
ixa_cmd_set_param(&cmd_chan, 1, 2, 0); /* not used */

/* Create a loop by linking the "next" pointer for the second command
back to the first - this will cause SmartDMA to continuously execute
this sequence. */
ixa_mmu_map_addr( (void *)cmd1, &phys_addr );
ixa_cmd_set_next(  &cmd_chan, 1, phys_addr);
```

You may notice that the source, destination, and size fields of the command headers do not appear to be explicitly set. This is because *the ixa_cmd_set_opcode* function does this in addition to setting the opcode field.

Once the command chain has been created, the processing of the commands by SmartDMA can be initiated by calling the *ixa_cmd_start* function.

```
stat = ixa_cmd_start(&cmd_chan, SYNC);
```

In synchronous mode, the function will not return until the IOPlus has completed processing the commands. A status flag is polled to determine when SmartDMA has finished processing the chain. When called in asynchronous mode, the function returns as soon as the command processing has been started. Upon completion of the command chain processing or upon an error condition, the IOPlus updates the status of the command channel. In the above case, the command channel structure was relocated to local memory when the channel was opened. The status of a command channel can be determined by calling the *ixa_cmd_status* function. Should an error condition exist, information about the error can be obtained from the response chain packets by calling the *ixa_cmd_error* function.

```
do
  {
    ixa_cmd_status(&cmd_chan, &status);
  } while (status == CMD_STATUS_ACTIVE);
```

```
if (status == CMD_STATUS_ERROR)
  ixa_cmd_error(&cmd_chan, &cmd_err);
```

Note that since the command channel structure was relocated to local memory when it was opened, the SPE does not have to go out on the PCI bus to monitor the channel status. Likewise, since the response chain was setup in local memory, the SPE does not have to access the PCI bus to get error information.

Command processing for a given command channel can by stopped by calling the *ixa_cmd_stop* function. If a command channel is no longer needed, you can close the channel and make it available to other processes using the *ixa_cmd_close* function:

```
stat = ixa_cmd_close(&cmd_chan);
```

A more detailed command chain example is included on the IXATools distribution CD.

5.5 Programming SmartDMA

The functions listed in this section are used to create and manage SmartDMA command chains. These functions are prototyped in "ixa.h" and are included in "libixa.a".

ixa_cmd_close

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_close (CMD_CHANNEL *cmd_chan);
```

DESCRIPTION:

ixa_cmd_close frees a SmartDMA command channel opened with *ixa_cmd_open.*

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPEN: command channel has not been opened.

ixa_cmd_error

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_error ( CMD_CHANNEL *cmd_chan,
                              CMD_ERROR   *cmd_err);
```

DESCRIPTION:

The *ixa_cmd_error* function returns the error status of the command channel specified by *cmd_chan*. The error status is stored in *cmd_err*, which is a structure defined as follows:

```
typedef struct _cmd_error_struct
{
  unsigned int cnt;      /* number of errors in response chain */
  unsigned int op;       /* opcode of command causing error */
  unsigned int err_code; /* param 1 of response packet */
  unsigned int err_val1; /* param 2 of response packet */
  unsigned int err_val2; /* param 3 of response packet */
  unsigned int err_val3; /* param 4 of response packet */
} CMD_ERROR;
```

The *cnt* element indicates how many error responses exist in the command channel response chain. The *op* element identifies the command operation that generated the first error in the response chain. The *err_xxxx* values are the parameters of the first error CMD_ACK response packet in the response chain. Possible values for err_code are as follows:

CMD_ERR_NONE – no error or no response packet
CMD_ERR_NOT_SUPPORTED – command not supported
CMD_ERR_INVALID_PARAM – invalid parameter
CMD_ERR_BUS_ERROR – bus error
CMD_ERR_OPERATION_FAILED – operation failed
CMD_ERR_NOT_OPENED – failed to open
CMD_ERR_FULL – resource full
CMD_ERR_DATA_MISMATCH – data mismatch

For more detail on CMD_ACK packets, see the following section.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_CMD_CHANNEL_BUSY: command channel already active.
IXABSP_DEVICE_NOT_OPEN: command channel has not been opened.

ixa_cmd_open

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_open (  CMD_CHANNEL  *cmd_chan,
                              unsigned int *cmd_addr,
                              unsigned int *response_addr,
                              unsigned int  options);
```

DESCRIPTION:

ixa_ cmd_open opens a SmartDMA command channel. The *cmd_chan* parameter is a pointer to a CMD_CHANNEL structure. Once successfully opened, this CMD_CHANNEL structure pointer can be used to manage IOPlus command processing using the various *ixa_cmd* functions. The *cmd_addr* parameter points to the start of the command chain. *response_addr* points to the response chain. By default, the *cmd_addr* and *response_addr* parameters are interpreted as PCI addresses. The *options* parameter can be used to identify them as local memory addresses and/or to relocate the chains to local memory. The following options can be OR'd together:

CMD_CHANNEL_OPTION_CMD_LOCAL – the *cmd_addr* is a local memory address
CMD_CHANNEL_OPTION_RESP_LOCAL – the *resp_addr* is a local memory address
CMD_CHANNEL_OPTION_RELOCATE –the command channel is located in SPE local memory rather than in global memory.

SmartDMA can only handle a fixed number of command channels simultaneously. Once created by calling *ixa_cmd_open*, a command channel can be freed by calling *ixa_cmd_close*, thereby freeing the command channel for use by another processor.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_RESOURCE_UNAVAILABLE: all command channels already in used.

ixa_cmd_set_next, ixa_cmd_set_opcode, ixa_cmd_set_option, ixa_cmd_set_param

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_set_next (    CMD_CHANNEL *cmd_chan,
                                    unsigned int cmd_idx,
                                    unsigned int next_val);

IXABSP_STATUS ixa_cmd_set_opcode (  CMD_CHANNEL *cmd_chan,
                                    unsigned int cmd_idx,
                                    unsigned int op_val);

IXABSP_STATUS ixa_cmd_set_option (  CMD_CHANNEL *cmd_chan,
                                    unsigned int cmd_idx,
                                    unsigned int opt_val);

IXABSP_STATUS ixa_cmd_set_param (   CMD_CHANNEL *cmd_chan,
                                    unsigned int cmd_idx,
                                    unsigned int param_idx,
                                    unsigned int param_val);
```

DESCRIPTION:

The *ixa_cmd_set* functions provide a convenient method for building a command chain. The *cmd_chan* parameter specifies the command channel opened by *ixa_cmd_open*. *cmd_idx* specifies which command in the chain is being modified, with the first command in the chain having the index of 0. The *val* parameter specifies the value to which the command field is to be set. The *ixa_cmd_set_param* function has an additional parameter, *param_idx*, which specifies a parameter index, starting with the index of 0. The specific command fields set by the various *ixa_cmd_set* functions are as follows:

ixa_cmd_set_opcode – sets the opcode field of the command. In addition, calling this function will also set the source, destination and size fields of the specified command.

ixa_cmd_set_option – sets the option field of the command. Valid values, which can be OR'd together, are as follows:

CMD_OPTION_ACK – generate a CMD_ACK response packet
CMD_OPTION_ATOMIC – immediately process next command in the chain

ixa_cmd_set_next – sets the next field of the command. The last command in a command chain should have its next field set to 0. Before setting fields for a given command, you must first set the next field of the previous command in the chain.

ixa_cmd_set_param – sets the specified parameter field of the command. Various macros are defined in *ixa.h* to assist in setting parameter fields for specific commands. For example, CMD_MOVE_DATA_PCI_ADDR can be used to set the source and destination address option parameters of a CMD_MOVE_DATA packet, to identify that the address is a PCI address. Please refer to the *ixa.h* header file for the full list of command parameter macros.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPEN: command channel has not been opened.

ixa_cmd_start

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_start ( CMD_CHANNEL *cmd_chan,
                              unsigned int mode);
```

DESCRIPTION:

ixa_cmd_start initiates IOPlus processing of the command channel indicated by *cmd_chan*. The *cmd_chan* must have been previously opened by the *ixa_cmd_open* function. The *mode* parameter specifies whether the function waits for the IOPlus to complete command processing for the channel before returning (SYNC), or if it returns immediately after initiating the IOPlus request (ASYNC). Prior to calling this function, a command chain must have been built using the *ixa_cmd_set* functions. The status of a command channel can be determined by calling *ixa_cmd_status*.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPEN: command channel has not been opened.

ixa_cmd_status

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_status (CMD_CHANNEL  *cmd_chan,
                              unsigned int *status);
```

DESCRIPTION:

The *ixa_cmd_status* function returns the status of the command channel specified by *cmd_chan*. Upon return, the *status* pointer will be one of the following:

CMD_STATUS_ACTIVE – if the request is still being processed by the IOPlus
CMD_STATUS_DONE – if command processing is complete
CMD_STATUS_ERROR – if an error occurred during command processing

If an error status is indicated, more information regarding the error can be obtained by calling the *ixa_cmd_error* function.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPEN: command channel has not been opened.

ixa_cmd_stop

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_stop (CMD_CHANNEL *cmd_chan);
```

DESCRIPTION:

ixa_cmd_stop halts IOPlus processing of the command channel indicated by *cmd_chan*. The *cmd_chan* must have been previously opened by the *ixa_cmd_open* function.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPEN: command channel has not been opened.

5.6 SmartDMA Commands

This section describes all commands processed by SmartDMA. The format of the command, as well as a description of the command, is provided. All commands consist of packet header, parameters, and data, as described in Appendix A.

These commands are used to build command chains, using the SPE functions described in the previous section.

Several commands use a memory section ID as a parameter. Table 5.1 lists the valid IDs for that parameter. Tables 5.2, 5.3 and 5.4 give the offsets for the ID locations that contain tables of information. The offsets are used as an additional parameter to commands that require a memory section ID.

Table 5.1 – Command Table

Command	Op-code	Description
CMD_ACK	ACEF0001	Acknowledgement packet – generated by SmartDMA
CMD_GENERATE_INT	ACEF0007	Generate an interrupt
CMD_MOVE_DATA	ACEF0006	Move data between on-board resources, or between on- and off-board resources
CMD_READ_DATA	ACEF000C	Read data from a specified resource
CMD_RESET	ACEF0003	Reset a processor, cluster, or board
CMD_SUPPORT_QUERY	ACEF0002	Determine whether command is supported
CMD_TOGGLE_LED	ACEF0004	Toggle red LED
CMD_USER	ACEF000A	User-defined command

Table 5.2 - Memory Section IDs

ID	Memory Type	Supported by IOPlus	Supported by SPE
0	Local data memory	X	X
1	Local program memory	X	X
2	Global memory (base address)	X	
3	Global memory (user space)	X	X
4	PMC1 memory	X	X
5	PMC2 memory	X	X
6	Local SDRAM memory		X
7	Reserved		
8	SPE A/B SDRAM memory	X	
9	Reserved		
10	Reserved		

11	Reserved		
12	Reserved		
13	Reserved		
14	Reserved		
15	Reserved		
16	SPE C/D SDRAM memory	X	
17	Reserved		
18	Reserved		
19	Reserved		
20	Reserved		
21	Reserved		
22	Reserved		
23	Reserved		
24	Version Information table	X	
25	Status Information table	X	
26	Board Information table	X	
27	Flash recovery code	X	
28	Flash startup code	X	
29	Flash runtime code	X	
30	Production parameters	X	
31	Configuration parameters	X	
32	Xilinx program	X	
33	Reserved	X	
34	User program	X	
35	User data	X	
36	This table	X	

Table 5.3 - Format of Version Information Table (Table ID 24)

Field offset	Value
0	Version ID of the version information (default is 0)
1	IOPlus recovery code version
2	IOPlus initialization code version
3	IOPlus run-time code version
4	Reserved
5	Reserved
6	Board Resource Manager version
7	Reserved
8	Reserved
9	Reserved
10	Serial Number
11	Initial Build Configuration
12	Current Build Configuration
13	Initial Production Release Date
14	PCO / RMA 1 ID
15	PCO / RMA 1 Date
16	PCO / RMA 2 ID
17	PCO / RMA 2 Date
18	PCO / RMA 3 ID
19	PCO / RMA 3 Date

20	PCO / RMA 4 ID
21	PCO / RMA 4 Date
22	PCO / RMA 5 ID
23	PCO / RMA 5 Date
24-31	Reserved

Table 5.4 - Format of Status Information Table (Table ID 25)

Field offset	Processor	Valid Processor Status Values
0	IOPlus status	0x00000000: Unknown 0x00000001: Reset 0x00000002: Waiting for load of user code 0x00000003: Loading, waiting to start 0x00000004: Running 0x00000005: Stopped at breakpoint 0x00000006: Failed self test 0x00000007: Not Installed
1	SPE A status	
2	SPE B status	
. . . .	. . . .	
N	SPE N status	

Table 5.5 - Format of Board Information Table (Table ID 26)

Field offset	Value
0	Board Type
1	Trace status
2	Board health
3	Product version
4	Initalization code version
5	Initialization code date
6	Runtime code version
7	Runtime code date
8	Software status (0 = no errors)
9	Health counter
10	IOPlus type
11	DSP/SPE type
12	Number of SPEs
13	Number of Hosts
14	Number of unallocated LAT entries
15	Reserved
16	Global Memory bank 0 size
17	Global Memory bank 0 type
18	Global Memory bank 1 size
19	Global Memory bank 1 type
20	Global Memory bank 2 size
21	Global Memory bank 2 type
22	Global Memory bank 3 size
23	Global Memory bank 3 type
24	Local Memory bank 0 size
25	Local Memory bank 0 type
26	Local Memory bank 1 size
27	Local Memory bank 1 type
28	Local Memory bank 2 size
29	Local Memory bank 2 type
30	Local Memory bank 3 size
31	Local Memory bank 3 type
32 – 35	Flash memory (0 – 3) types
36	VME bridge ID
37 – 40	PMC (0 – 3) Ids
41 – 59	Reserved
60	VME base address
61	VME address set method
62	VME System controller status
63	VME to PCI address translation
64	Board Information Structure address
65	Host list table address
66	Reserved
67	Address of user area of global memory
68	Upper 24 bits of MAC address
69	Lower 24 bits of MAC address
70	Reserved
71	Host command area address

Table 5.5 - Format of Board Information Table (Table ID 26) cont.

Field offset	Value
72	Host command area size
73	Host response area address
74	Host response area size
75	Hardware variant id
76	New command in LAT flag
77 - 78	Reserved
79	Address of PMC configuration space cotents
80 - 91	Reserved
92	System Processor ID
93	Board revision ID
94 - 95	Reserved
96	Local processor clock rate
97	Local memory bus speed
98	L2 cache size
99	L2 cache ratio
100	Flash size

CMD_ACK

Header:

Opcode: CMD_ACK
Source: source ID
Destination: destination ID
Size: CMD_ACK_SZ

Parameters:

Parm1:	status	0:	Yes or Pass
		<> 0:	Error code
Parm2:	Info1		
Parm3:	Info2		
Parm4:	Command Op-code		

Description:

CMD_ACK is a generic command acknowledgment message. Parameter 1 specifies a result code, where zero indicates the command has completed successfully, and a non-zero value indicates that an error has occurred. The other parameter fields provide more detailed information about the error condition.

The IOPlus will return a CMD_ACK packet in response to all commands. The following error messages may be returned in the CMD_ACK packet:

ERR_NONE:

Param 1	0
Param 2	0
Param 3	0
Param 4	Op-code

ERR_NOT_SUPPORTED:

Param 1	-1
Param 2	Op-code that is not supported
Param 3	0
Param 4	Op-code

ERR_INVALID_PARAM:

Param 1	-2
Param 2	Parameter number that is invalid
Param 3	Value of invalid parameter
Param 4	Op-code

ERR_BUS_ERROR:

Param 1	-3
Param 2	Address at which bus error occurred
Param 3	0
Param 4	Op-code

ERR_OPERATION_FAILED:

Param 1	-4
Param 2	0
Param 3	0
Param 4	Op-code

ERR_NOT_OPENED:

Param 1	-5
Param 2	0
Param 3	0
Param 4	Op-code

ERR_FULL:

Param 1	-6
Param 2	0
Param 3	0
Param 4	Op-code

ERR_DATA_MISMATCH:

Param 1	-7
Param 2	Expected Value
Param 3	Actual Value
Param 4	Op-code

CMD_GENERATE_INT

Header:

Opcode: CMD_GENERATE_INT
Source: source ID
Destination: destination ID
Size: CMD_GENERATE_INT_SZ

Parameters:

Parm1: Interrupt type (VME, or mailbox)
- CMD_VME_INTERRUPT
- IXC_MBOX1_INTERRUPT (IXC board only)
- IXC_MBOX2_INTERRUPT (IXC board only)
- CMD_IXA_SPEA_INTERRUPT (IXA board only)
- CMD_IXA_SPEB_INTERRUPT (IXA board only)
- CMD_IXA_SPEC_INTERRUPT (IXA board only)
- CMD_IXA_SPED_INTERRUPT (IXA board only)

Parm2: Info1
- When generating VME interrupts:
  - Interrupt level
- When generating Mailbox interrupts (on IXC boards):
  - Processor ID
- When generating SPE interrupts (on IXA boards):
  - INT_VECTOR_IPIU0
  - INT_VECTOR_IPIU1
  - INT_VECTOR_IPIU2
  - INT_VECTOR_IPIU3

Parm3: Info2
- When generating VME interrupts:
  - Interrupt vector
- Otherwise:
  - Reserved (must be 0)

Description:

This command generates interrupts on the VME bus. It also will generate interrupts to specific SPEs. The meaning of the Info1 and Info2 fields changes depending on the type of interrupt being generated. Validity checks are performed on the interrupt type field, interrupt level, and processor ID.

Response:

ERR_NONE after the interrupt has been generated and acknowledged successfully.

INVALID_PARAM if any of the parameter fields is invalid.

BUS_ERROR if an error occurs while generating the interrupt.

CMD_MOVE_DATA

Header:

Opcode: CMD_MOVE_DATA
Source: source ID
Destination: destination ID
Size: CMD_MOVE_DATA_SZ

Parameters:

Parm1: Source address
Parm2: Stride in words
Parm3: Option flag

The following options can be OR-ed together

Memory address type:
- CMD_MOVE_DATA_PCI_ADDR
- CMD_MOVE_DATA_VME_ADDR
- CMD_MOVE_DATA_IOP_ADDR
- CMD_MOVE_DATA_DSPA_ADDR
- CMD_MOVE_DATA_DSPB_ADDR
- CMD_MOVE_DATA_DSPC_ADDR
- CMD_MOVE_DATA_DSPD_ADDR

VME transfer mode:
- CMD_MOVE_DATA_USER_DATA_AM
- CMD_MOVE_DATA_USER_PRG_AM
- CMD_MOVE_DATA_SUPER_DATA_AM
- CMD_MOVE_DATA_SUPER_PRG_AM

VME address size:
- CMD_MOVE_DATA_A32
- CMD_MOVE_DATA_A24
- CMD_MOVE_DATA_A16

VME Data size:
- CMD_MOVE_DATA_D32
- CMD_MOVE_DATA_D16
- CMD_MOVE_DATA_D8
- CMD_MOVE_DATA_D32BLT
- CMD_MOVE_DATA_D64BLT

Parm4: Destination address
Parm5: Stride in words
Parm6: Option flag (see definition of option flag above)
Parm7: Number of words
Parm8: Operation code
- CMD_MOVE_DATA_COPY
- CMD_MOVE_DATA_BYTE_SWAP

Parm9: Info1 (reserved – set to 0)
Parm10: Info2 (reserved – set to 0)

Description:

This command moves data between memory located on the VME bus, PCI bus, and local SPE bus. Data is copied from the source address (with the specified stride), to the destination address (with the specified stride), while performing the operation on the data specified in the operation code.

Validity checks are performed on all parameters except Stride. The option flag indicates options for the address, which indicate whether the address is on the PCI, VME or local bus. If the transaction is for a VME address, then the VME address modifier parameters should be specified. The operation code specifies an operation to be performed on the data while it is copied from the source to the destination memory. The Info parameters provide additional information required by certain operation codes.

When data is copied from an on-board resource to or from the VME bus with a stride of 1, the Universe II DMA engine is used to move the data efficiently. When data is copied between on-board resources with a stride of 1, the MPC8240 DMA engine is used to move data efficiently. Other types of data movement are performed directly by the IOPlus.

Response:

ERR_NONE after the data has been successfully written.

INVALID_PARAM if any of the parameter fields is invalid.

BUS_ERROR if an error occurs while performing the move.

CMD_READ_DATA

Header:

Opcode: CMD_READ_DATA
Source: source ID
Destination: destination ID
Size: CMD_READ_DATA_SZ

Parameters:

Parm1: Memory Section ID (See Table 5.7)
Parm2: Offset
Parm3: Stride
Parm4: Option flag
- 0: bus type = PCI
- 1: bus type = reserved

Description:

This command is used to read a block of data from on-board memory. Memory Section ID selection is described in CMD_WRITE_DATA. Validity checks for memory ID and option flag are performed.

Response:

ERR_NONE after the data has been successfully read. The read data is appended to the CMD_ACK message after the fourth parameter.

INVALID_PARAM if any of the parameter fields is invalid.

BUS_ERROR if an error occurs while performing the read.

CMD_RESET

Header:

Opcode: CMD_RESET
Source: source ID
Destination: destination ID
Size: CMD_RESET_SZ

Parameters:

Parm1: Reset type Reset or release | SPE processor or board

0x00000001	Reserved
0x00000002	Reserved
0x00000004	Reset Board
0x80000001	Reserved
0x80000002	Reserved

Description:

This command is used to reset the board. Validity checks for Reset type and Processor ID are performed.

Response:

ERR_NONE indicates that the reset (or release from reset) has been performed successfully.

INVALID_PARAM if any of the parameter fields is invalid.

CMD_SUPPORT_QUERY

Header:

Opcode: CMD_SUPPORT_QUERY
Source: source ID
Destination: destination ID
Size: CMD_SUPPORT_QUERY_SZ

Parameters:

Parm1: Command query op-code
Parm2: Command version (reserved for future use; must be 0x00000000)

Description:

This command is used by either the host, or by an SPE, to determine whether the IOPlus supports a specified command. The IOPlus responds to this command with either a CMD_ACK / ERR_NONE or a CMD_ACK / ERR_NOT_SUPPORTED.

CMD_TOGGLE_LED

Header:

Opcode:	CMD_TOGGLE_LED
Source:	source ID
Destination:	destination ID
Size:	CMD_TOGGLE_LED_SZ

Parameters:

Parm1:	Count
Parm2:	On duration (in milliseconds)
Parm3:	Off duration (in milliseconds)

Description:

This command is used to toggle LEDs. Count specifies the number of toggle cycles. No validity checks are performed.

If count is zero, then the state of the LED is toggled (i.e., if the LED is currently on, it is turned off; if the LED is currently off, it is turned on). When count is zero, the second and third parameters are ignored.

Response:

ERR_NONE indicates that the toggle has been performed successfully.

CMD_USER

Header:

Opcode:	CMD_USER
Source:	source ID
Destination:	destination ID
Size:	CMD_USER_SZ

Parameters:

Parm1:	Info1
Parm2:	Info2
Parm3:	Info3
Parm4:	Info4

Description:

User software can use this command to setup user-defined commands.

Response:

The response to this command is user-defined.

CMD_VME_RMW

Header:

Opcode: CMD_VME_RMW
Source: source ID
Destination: destination ID
Size: CMD_VME_RMW_SZ

Parameters:

Parm1: VME source address
Parm2: VME option
Parm3: Operation
- CMD_VME_RW_SET
- CMD_VME_RW_CLEAR
- CMD_VME_RW_INCREMENT
- CMD_VME_RW_DECREMENT
- CMD_VME_RW_COMPARE_SWAP

Parm4: Condition
- CMD_VME_RW_ALWAYS
- CMD_VME_RW_IF_ZERO
- CMD_VME_RW_IF_NOT_ZERO

Parm5: Compare Value
Parm6: Swap Value

Description:

This command generates the read-modify-write cycle onto VME bus. It is sent from SPE processors to the IOPlus.

Response:

ERR_NONE after the read/modify/write has been performed successfully.

INVALID_PARAM if any of the parameter fields is invalid.

BUS_ERROR if an error occurs while performing the read/modify/write.

CMD_WAIT_INT

Header:

Opcode: CMD_WAIT_INT
Source: source ID
Destination: destination ID
Size: CMD_WAIT_INT_SZ

Parameters:

Parm1: Interrupt type

CMD_WAIT_INT_VME
CMD_WAIT_INT_PCI
CMD_WAIT_INT_MBOX
CMD_TRAP_INT_VME
CMD_TRAP_INT_PCI
CMD_TRAPT_INT_MBOX

Parm2: Info1

When waiting for VME interrupt:
Specifies the interrupt level to wait for
When waiting for mailbox interrupt:
Specifies the value in the mailbox to wait for

Parm3: Info2

When waiting for VME interrupt:
If 0xFFFFFFFF, then wait for any vector
Otherwise, specifies the interrupt vector to wait for.

Parm4: Info3

Action to take when desired interrupt occurs:

Bit Fields								
31	----------	26	25	----------	3	2	1	0
Processor ID			Reserved (must be 0)			Global	Write	Interrupt

Interrupt bit: Setting this bit to one causes the SPE with ID "processor ID" to be interrupted

Write bit: Setting this bit to one causes the value at the address specified in parameter 5 to be filled with the value described below

Global bit: Setting this bit indicates that the address specified in parameter 5 is located in global memory rather than in the local memory of the SPE specified by the relative processor ID.

Note: Setting both the "interrupt bit" and the "write bit" to zero causes the IOPlus to

continue chain processing without other action.

Parm5: Address

Address (either local SPE address or global memory address) to modify when the desired interrupt occurs and the "write bit" of parameter 4 is set. This field is ignored by the IOPlus if the "write bit" of parameter 4 is cleared. The value written to this address will have the following format:

Bit Fields											
31	----	24	23	----------	16	15	----------	8	7	-------	0
Level			0xFF			0xFF			Vector		

Description:

SPE processors use this command to signal a wait for interrupt to the I/O controller. Validity checks are performed on the interrupt type field. When the Interrupt Type is set to "wait", then the IOPlus will wait until the specified interrupt occurs before continuing to process the command chain (note that command chains for other software components will continue to be processed). When the Interrupt Type is set to "trap", then the IOPlus will continue chain processing, and perform the specified action when an interrupt occurs.

Response:

ERR_NONE after the interrupt has been received.

INVALID_PARAM if any of the parameter fields is invalid.

CMD_WRITE_DATA

Header:

Opcode: CMD_WRITE_DATA
Source: source ID
Destination: destination ID
Size: CMD_WRITE_DATA_SZ + size of data

Parameters:

Parm1: Memory Section ID (See Table 5.7)
Parm2: Offset within section
Parm3: Stride
Parm4: Option flag
- 0: bus type = PCI
- 1: bus type = reserved
- 2: initial write to flash data type, ignore data
- 4: final write to flash data, type, flush data

Description:

This command can be used to write a block of data to on-board memory. Memory Section ID indicates what memory section is targeted. Offset specifies the location within the memory section. Stride specifies the incremental step used to increment the memory pointer. Option Flag specifies the type of bus used to transfer the data. Fields to be checked for validity in this command are memory ID and option flag.

Response:

ERR_NONE after the data has been successfully written.

ERR_INVALID_PARAM if any of the parameter fields is invalid.

ERR_BUS_ERROR if an error occurs while performing the write.

Note: to write data to VME address space, use CMD_MOVE_DATA

Writing to flash memory:

Note: This command may be used to write to the flash memory. Because dealing with flash is different than dealing with memory, a slightly different protocol must be used when writing to flash. To write a specific flash data type to flash, the following sequence of commands should be used:

CMD_WRITE_DATA
- Size set to size of the entire data object that will be written
- Memory section ID set to the proper flash data type
- Options flag set to 2, which indicates that this is the first write to flash for the new flash data type. No write is actually performed; rather, the directory structure is configured properly for the write that is about to occur
- No data should be appended to this command

CMD_WRITE_DATA
- Size set to size of block of data attached to this command
- Memory section ID set to the proper flash data type
- Options flag set to 0, which indicates a normal flash memory write
- Data to be written to flash

CMD_WRITE_DATA
- Size set to size of block of data attached to this command
- Memory section ID set to the proper flash data type
- Options flag set to 0, which indicates a normal flash memory write
- Data to be written to flash

CMD_WRITE_DATA
- Size set to size of block of data attached to this command
- Memory section ID set to the proper flash data type
- Options flag set to 0, which indicates a normal flash memory write
- Data to be written to flash

CMD_WRITE_DATA
- Size set to size of the entire data object that has been written
- Memory section ID set to the proper flash data type
- Options flag set to 4, which indicates that the write of this flash type has completed, and any buffered data should be flushed to the flash
- No data should be appended to this command

Chapter 6: Programming the IOPlus

6.1 Introduction

Chapter 5 shows how the IOPlus firmware can be used by SPE and host applications to perform various background data movement and board resource manipulation activities. This chapter addresses the subject of developing and running an application on the IOPlus. There are two approaches to accomplishing this, using the IOPlus JTAG for code load and debug, or using the Ethernet port and a boot loader to remotely load and debug from a networked workstation. The boot loader supports VxWorks development from a Tornado based workstation.

When using the IOPlus for an application you can still have the IOPlus service commands from the SPEs in the same manner described in Chapter 5. This is accomplished by linking your application to a library that has background services it provides in addition to services your application can directly access. This chapter defines the IOPlus functional interface for IOPlus application development (IOPlusAPI).

6.2 VxWorks and the IOPlus

The development of an application for the IOPlus, that uses VxWorks and supports the Tornado environment, requires the installation and configuration of the TM as described in Chapter 2. It also requires the VxWorks BSP for the IOPlus to be installed as part of the IXAtools installation process. This BSP is a library of C and assembly, source and object files that enables VxWorks to use the IOPlus resources.

VxWorks BSP for the IOPlus

The VxWorks BSP implementation for the IOPlus is a subset of the full BSP definition given in the Wind River documentation. This is primarily due to the IXA7's design having less single board computer attributes, and the off-loading of a number of capabilities to the IOPlusAPI library. Table 6.1 presents a summary of BSP features and Table 6.2 provides a list of the BSP functions implemented. Many of the sysBus functions have ixa_VME counterparts. These functions can be called by their sysBus function names, or by their ixa_VME counterpart names. For those functions having both VxWorks and IXATools names, we recommend you use the sysBus names for applications developed under VxWorks. You may consult section 7 of this document for detailed information on the equivalent IXA functions.

Table 6.1 - IOPlus VxWorks BSP Features

Feature	Description
Boot ROM Images	bootrom
Boot Devices	Ethernet:dc
VxWorks images	vxWorks, vxWorks.st, vxWorks5_2
MMU	- basic bundled MMU support - uses BAT registers and PTEs for address translation
Cache mode	Instruction, data, & L1 cache in copyback cache mode
Sysclock	using PPC Decrementer
AuxClock	using timer 1 of EPIC
Serial	using I2C interface on TM
Timestamp	using timer 0 of EPIC
Ethernet	END driver for DEC21143 chip on TM END driver for INTEL 82559 on PMC-ENT1

The default configuration of the BSP and the supplied VxWorks image, bootrom.hex, are meant to be a starting point to allow you to quickly run VxWorks with the IOPlus. Therefore, the configuration may not be initially suitable for the user application or target system, and reconfiguration of the BSP may be necessary. Refer to the VxWorks Programmer's Guide: Configuration: The Board Support Package for more information on the configuration of the BSP.

All BSP configuration modifications should be made to the file config.h (unless specified differently in this document) that resides in the build directory of the BSP. You should walk through the file and make changes to reflect your system configuration, then rebuild the image with the new configuration. Changes to RAM_LOW_ADRS or RAM_HIGH_ADRS require the bootrom.hex image to be rebuilt and the Flash memory re-programmed.

Since flash memory is managed by the IOPlus firmware, the VxWorks boot entry point in flash memory is fixed. Users should not attempt to modify ROM_BASE_ADRS or ROM_TEXT_ADRS.

Table 6.2 - IOPlus VxWorks BSP Functions

Function	Description
sysAuxClkConnect()	connect a routine to the auxiliary clock interrupt
sysAuxClkDisable()	turn auxiliary clock interrupts off
sysAuxClkEnable()	turn auxiliary clock interrupts on
sysAuxClkInt()	handle auxiliary clock interrupts
sysAuxClkRateGet()	get the auxiliary clock rate
sysAuxClkRateSet()	set the auxiliary clock rate
sysBspRev()	return the bsp version and the bsp revision number
sysBusIntAck()	acknowledge/clear interrupt
sysBusIntGen()	generate an inter-processor or VME interrupt
sysBusTas()	test and set an on-board location
sysBusToLocalAdrs()	convert bus address to local address
sysClkConnect()	connect a routine to the system clock interrupt
sysClkDisable()	turn off system clock interrupts
sysClkEnable()	turn on system clock interrupts
sysClkRateGet()	get the system clock rate
sysClkRateSet()	set the system clock rate
sysLocalToBusAdrs()	convert local address to bus address
sysIntDisable()	disable interrupts
sysIntEnable()	enable interrupts
sysMailboxConnect()	Install an interrupt service routine for a mailbox
sysMailboxDisable()	Disable a mailbox interrupt
sysMailboxEnable()	Enable a mailbox interrupt
sysMailboxPost()	Post to a local mailbox, generating an interrupt
sysMemTop()	get the address of the top of VxWorks memory
sysModel()	return the model name of the target card
sysPhysMemTop()	get the address of the top of physical memory
sysProcNumGet()	get the processor number
sysProcNumSet()	set the processor number
sysSerialChanGet()	get the SIO_CHAN device associated with a serial channel
sysTimestamp()	get the timestamp timer tick count
sysTimestampConnect()	connect a user routine to the timestamp interrupt
sysTimestampDisable()	disable the timestamp timer
sysTimestampEnable()	initialize and enable the timestamp timer
sysTimestampFreq()	get the timestamp timer clock frequency
sysTimestampLock()	get the timestamp tick counter
sysTimestampPeriod()	get the timestamp timer period
sysToggleLed()	turn the status LED on or off
sysToMonitor()	transfer control to the ROM monitor
sysBusOpen()	Open a VME master window
sysBusClose()	Close a VME master window
sysBusDmaIn()	Start VME dma input operation
sysBusDmaOut()	Start VME dma output operation
sysBusMapAddr()	Map a VME address within a window to a local address
sysBusReMap()	Reset the mapping parameters for a window
sysBusRead()	Read VME data into local memory
sysBusWrite()	Write VME data into local memory
sysBusPeek()	Read a single byte, word or long from VME bus

sysBusPoke()	Write a single byte, word or long to VME bus
sysBusProbe()	Determine if an address is accessible
pciIomapLibInit()	Initialize the configuration access method and address
pciFindDevice()	Find the nth device with a given device and vendor ID
pciFindClass()	Find the nth device with a given class
pciConfigInByte()	Read one byte from configuration space
pciConfigInWord()	Read one word from configuration space
pciConfigInLong()	Read one long from configuration space
pciConfigOutByte()	Write one byte to configuration space
pciConfigOutWord()	Write one word to configuration space
pciConfigOutLong()	Write one long to configuration space
pciSpecialCycle()	Generate a PCI special cycle
pciInt()	Interrupt handler for shared PC interrupts
pciIntConnect()	Connect the interrupt handler to the PCI interrupt
pciIntDisconnect()	Disconnect the interrupt handler from a PCI interrupt
pciDeviceShow()	Print information about PCI devices
pciHeaderShow()	Print a header of a specied PCI device
pciFindDeviceShow()	Find device by ID then print
pciFindClassShow()	Find device by class, then print

RAM Usage

The first 256K bytes (0x40000) of RAM is reserved for exception vector handlers and global IOPlus data structures. This area of memory is configured to be uncached. This is necessary since on-board SPEs must be able to access global IOP data structures residing in this region. RAM_LOW_ADRS, defined in config.h, is defined to start after this location.

Locations RAM_LOW_ADRS through (LOCAL_MEM_SIZE – USER_RESERVED_MEM) are available for VxWorks. In order to maximize VxWorks performance, this region is configured to have cache enabled. Other on-board processors must not use memory in this region, since coherency is not guaranteed. These constants are defined in config.h, and may be adjusted according to your application's needs.

The region from (LOCAL_MEM_SIZE – USER_RESERVED_MEM) through the end of memory is non-cached. This region is available for use by other processors and PMC devices. Because this region is uncached and guarded, write operations to this region from the IOPlus occur in order and are visible to other processors.

The VxWorks ROM boot loader uses an additional region of memory, RAM_HIGH_ADRS through LOCAL_MEM_SIZE, for its workspace. After reset, the ROM bootstrap loader moves a copy of VxWorks from flash memory to this region, uncompressing the load during the move. The boot loader then runs from this location, loading a VxWorks image using FTP and a specified host on the network. Therefore, SPE devices should not use this region of memory if the ROM boot loader is employed. Typically this loader is used only during development and debug. Users can replace the

bootstrap loader with a VxWorks application that relocates itself from flash memory to low memory.

Table 6.3 shows the VxWorks memory map and how it partitions the first block of global memory space. This memory is further partitioned into a VxWorks region and a shared memory region. This partitioning can be adjusted by modifying config.h. The use of separate regions for VxWorks and shared memory allows optimal caching attributes to be specified. The shared region should not be cached, since any other processor or device can access it. However the VxWorks area may be cached.

Table 6.3 - RAM Map for VxWorks

From	To	Description
0x00000000	0x00002FFF	Interrupt Vectors
0x00030000	0x00037FFF	Board information data structure
0x00038000	0x0003FFFF	VxWorks initial stack and work area
0x00040000	0x00FFFFFF	VxWorks text data bss and stack
0x01000000	0x01FFFFFF	Shared global memory

Memory Management Unit

The BSP uses both the Block Address Translation (BAT) and segment models for memory mapping. Refer to the VxWorks Programmer's Guide: Appendix F - PowerPC for details on the Memory Management Unit of the PowerPC, and the memory map models.

The BSP is configured to use the bundled basic memory management support (INCLUDE_MMU_BASIC) for both data and instructions.

The region of memory from RAM_LOW_ADRS through (LOCAL_MEM_SIZE – USER_RESERVED_MEM) is cache enabled. Applications must avoid accessing this region of memory from other processors. All other regions of memory are cache inhibited and guarded.

MMU and Cache Configuration in sysLib.c

The BAT registers are configured using the data structure sysBatDesc[] defined in sysLib.c. Translation using BAT registers is faster than translation using Page Table Entries (PTEs), and is better suited for mapping large regions of memory. The first four BAT registers translate instruction addresses. The last four translate data addresses. Table 6.4 shows the default configuration of the BAT registers.

Table 6.4 - Default BAT Configuration

BAT	Description	Address Range	Cache Attribute
IBAT0	Flash	ROM_BASE_ADRS .. 0xFFFFFFFF	Cache Inhibited

IBAT1	Unassigned	0x .. 0x	Disabled
IBAT2	Unassigned	0x .. 0x	Disabled
IBAT3	Unassigned	0x .. 0x	Disabled
DBAT0	PMC2 space	0xD0000000 .. 0xDFFFFFFF	Cache Inhibited, Guarded
DBAT1	PMC1 space	0x80000000 .. 0x8FFFFFFF	Cache Inhibited, Guarded
DBAT2	PMC2 space	0xC0000000 .. 0xCFFFFFFF	Cache Inhibited, Guarded
DBAT3	I/O, Flash	0xF8000000 .. 0xFFFFFFFF	Cache Inhibited, Guarded

PTEs govern address translation and memory access for other regions of memory. PTEs are built by VxWorks during initialization from information in the sysPhysMemDesc[] data structure. Page Table Entries are more costly than BAT registers since PTEs require a data structure for each 4K of mapped space. Furthermore, PTEs are fetched on demand from memory when not found in the processor's translate look-a-side buffer, resulting in some performance impact.

Because of this, PTEs are used to manage smaller regions of memory. The default PTE configuration is shown in Table 6.5. Developers may change this by changing constants in config.h, or by editing the sysPhysMemDesc as needed. The last two entries allow the processors to access control registers within the Ethernet and VME controllers, and must not be modified.

Table 6.5 - Default PTE Configuration

Address Range	Cache Attributes
LOCAL_MEM_LOCAL_ADRS .. (RAM_LOW_ADRS-1)	writable, cache inhibited
RAM_LOW_ADDRS .. (LOCAL_MEM_SIZE–USER_RESERVED_MEM-1)	writable, cacheable
(LOCAL_MEM_SIZE-USER_RESERVED_MEM)..(LOCAL_MEM_SIZE-1)	writeable, cache inhibited
ENET_DEVICE_ADDR .. (ENET_DEVICE_ADDR+4k)	writeable, cache inhibited
VME_DEVICE_ADDR .. (VME_DEVICE_ADDR+4k)	writeable, cache inhibited

Device Drivers

The BSP includes the device drivers listed in Table 6.6. This section provides additional information on these drivers.

Table 6.6 - VxWorks BSP Device Drivers

Device Driver Name	Description
ppcDecTimer.c	PPC decrementer driver
auxTimer.c	auxiliary clock
nvRamToFlash.c	flash storage driver
epicIntrCtl.c	interrupt controller for the EPIC
epicTimestamp.c	timestamp driver

dec21x4xEnd.o	Ethernet driver for the DEC21143
Fei82557End.o	Ethernet driver for the INTEL 82559
i2c.o	i2c interface
sio.c	serial driver for the I2C interface

System Clock

The system clock is implemented using the PowerPC decrementer register. The minimum and maximum clock rates are defined as follows:

```
SYS_CLK_RATE_MIN      10
SYS_CLK_RATE_MAX      5000
```

Auxiliary Clock

The auxiliary clock is implemented using timer 1 of the EPIC on the IOPlus. The minimum and maximum clock rates are defined as follows:

```
AUX_CLK_RATE_MIN      20
AUX_CLK_RATE_MAX      5000
```

Non-Volatile RAM

The IXA7 contains no NVRAM. Instead, functions are provided to simulate NVRAM using a region of flash memory. Instead of a flat model for accessing flash memory, the IOPlus superimposes a directory structure onto flash memory. A 1K region of flash memory, called vxworks_nvram, is used to simulate NVRAM. This memory location is used to store the boot strings for the processors.

Interrupt Controller

The IOPlus includes an on-chip Enhanced Programmable Interrupt Controller (EPIC). This device is initialized and configured from the BSP interrupt controller device driver. VxWorks functions that connect, enable, and disable interrupts call this device driver.

External input 4 into the EPIC is driven from an interrupt multiplexer, an FPGA that connects any combination of up to 32 interrupt sources to the EPIC. Interrupt vectors and priorities are described in epicIntrCtl.h and shown in Table 6.7 This includes vector numbers for all EPIC interrupts, and vector numbers for the interrupt multiplexer as well. The priorities can be modified to suit the needs of your application. The vector numbers should not be altered.

> The I2C interrupt is used by the serial device driver, and must not be altered by user applications.

Table 6.7 - Interrupt Vectors and Priorities

Name	Vector	Priority	Description
INT_VECTOR_TIMER0	16	08	Epic timer 0 (timestamp timer)
INT_VECTOR_TIMER1	17	08	Epic timer 1 (auxiliary timer)
INT_VECTOR_TIMER2	18	08	Epic timer 2 (unused)
INT_VECTOR_TIMER3	19	08	Epic timer 3 (unused)
INT_VECTOR_EXT0	20	06	interrupt (SPE A)
INT_VECTOR_EXT1	21	06	interrupt (SPE B)
INT_VECTOR_EXT2	22	06	interrupt (SPE C)
INT_VECTOR_EXT3	23	06	interrupt (SPE D)
INT_VECTOR_EXT4	24	06	Interrupt multiplexer
INT_VECTOR_DMA0	09	07	DMA 0 completion
INT_VECTOR_DMA1	0A	07	DMA 1 completion
INT_VECTOR_MSG	FD	05	I2O Messaging (unused)
INT_VECTOR_I2C	FE	05	I2C (used for serial I/O)
INT_VECTOR_SPURIOUS	FF	n/a	Spurious interrupt

In addition to the vectors shown in Table 6.7 vectors 0x80 through 0x9F represent interrupt sources into the interrupt multiplexer. The device driver translates these interrupt vectors into appropriate bit masks that are then applied to the interrupt multiplexer.

> In this implementation, the VxWorks functions intEnable and intDisable, take a specific interrupt vector as an input, and they enable or disable that specific interrupt. This differs from many implementations, where all interrupts below a certain level are enabled or disabled as a group.

Timestamp Driver

The timestamp driver is implemented using timer 0 of the EPIC.

Network Driver

The BSP includes Ethernet interfaces for the DEC21143 and the INTEL 82559. The driver interface is based on the 4.4 BSD network stack (END). To use the DEC21143 network interface, specify "dc" as the boot device in the boot parameters. To use the INTEL 82559 boot device, specify "fei" as the boot device.

The IOPlus BSP initializes the shared memory network, allowing Ethernet access to all processors on the board with the IOPlus as the gateway (master).

The MAC address for an IXA7 TM is programmed in flash memory at Dy 4 Systems and is not user alterable.

Serial Interface

The serial interface is provided by the TM card or the PMC_ENET1 card. The serial interface connects to the IOPlus through an I2C interface. This is a two-wire, multi-master interface. Its use as an intermediate bus is generally transparent to the user application. However, your application should be aware of the fact that the I2C bus is used, and therefore not alter the state of the I2C control registers or its interrupt vectors.

The I2C interface supports a simple protocol between the IOPlus and the serial controller chip. This interface does not support modem control status information or programmable baud rates. Therefore, the baud rate is fixed on the TM by jumper settings that are sensed by the serial device at power-up.

The VxWorks serial device driver is programmed to accept any baud rate and modem control commands, and return 9600 baud should an application try and read the baud rate. This information is fictitious, since the device driver cannot access or alter these parameters over the I2C interface.

The IOPlus BSP will set up receive/transmit buffers for each processor in a shared memory region dedicated to serial I/O.

PCI and VME

The IOPlus initialization software, which runs before VxWorks, performs all PCI and VME configuration cycles. Since these resources are sharable between the IOPlus and the SPEs it is advisable to use the IOPlusAPI functions.

IOPlus Task Configuration

By default the IOPlus task is started under VxWorks using the USER_APPL_INIT code block in config.h. This task provides the IOPlus command interface for the SPEs. It also manages the allocation of flash memory as explained in Chapter 5. The task parameters may be modified to the point of excluding the task totally by undefining INCLUDE_USER_APPL in config.h. However excluding the task will result in the IOPlus becoming unresponsive to SPE commands.

6.3 The IOPlus Application Programming Interface

The IOPlus Application Programming Interace, IOPlusAPI, contains routines that provide access to IOPlus functionality from within programs running on the IOPlus (either standalone or within VxWorks). These routines are contained in a library called *libioplus_api.a*. Prototypes for the IOPlusAPI functions are specified in an include file called *ioplus_api.h*.

There is one additional file that goes with libioplus_api.a called "iop_semaphore.o", which contains stub routines that "fake" VxWorks semaphore. For VxWorks development, you should NOT link with this file. For standalone development on the IOPlus, you SHOULD link with this file.

The functionality provided by the IOPlusAPI follows:

ioplus_calloc

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_calloc( unsigned long num_bytes,
                                  unsigned long **ptr );
```

DESCRIPTION:

ioplus_calloc allocates num_bytes of memory, then zeros the allocated memory.

RETURN STATUS:

IOPLUS_NO_ERROR: memory allocated successfully.
IOPLUS_OUT_OF_MemORY: not enough free memory in the heap

NOTES:

This routine should be called instead of *ioplus_malloc* when the allocated memory needs to be zeroed.

ioplus_check_pci_dma_done

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_check_pci_dma_done( void );
```

DESCRIPTION:

ioplus_check_pci_dma_done checks whether the PCI DMA initiated by *ioplus_move_data* has completed.

RETURN STATUS:

IOPLUS_NO_ERROR: transfer completed.
IOPLUS_BUS_ERROR: a bus error occurred while attempting the transfer.
IOPLUS_PCI_DMA_IN_PROGRESS: the DMA transfer has not yet completed.

NOTES:

Call this routine when *ioplus_move_data* returns a status of IOPLUS_PCI_DMA_IN_PROGRESS.

ioplus_check_vme_dma_done

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_check_vme_dma_done( void );
```

DESCRIPTION:

ioplus_check_vme_dma_done checks whether the VME DMA initiated by *ioplus_move_data* has completed.

RETURN STATUS:

IOPLUS_NO_ERROR: transfer completed.
IOPLUS_BUS_ERROR: a bus error occurred while attempting the transfer.
IOPLUS_VME_DMA_IN_PROGRESS: the DMA transfer has not yet completed.

NOTES:

Call this routine when *ioplus_move_data* returns a status of IOPLUS_VME_DMA_IN_PROGRESS.

ioplus_free

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_free(unsigned long *ptr );
```

DESCRIPTION:

ioplus_free releases memory that was allocated by *ioplus_malloc*.

RETURN STATUS:

IOPLUS_NO_ERROR: memory released successfully.

NOTES:

This routine should be called to deallocate memory that was previously allocated by *ioplus_malloc*.

ioplus_generate_interrupt

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_generate_interrupt( unsigned long int_type,
                                              unsigned long id,
                                              unsigned long vector );
```

DESCRIPTION:

ioplus_generate_interrupt generates an interrupt on the specified interrupt line.

Interrupt types are:	1	VME
	2	reserved
	3	Mailbox INT 4 (IXC6 only)
	4	Mailbox INT 5 (IXC6 only)
Interrupt ID	When generating VME interrupts, specifies interrupt level, otherwise, specfies processor ID.	
Interrupt Vector	When generating VME interrupts, specifies interrupt vector, otherwise, reserved.	

RETURN STATUS:

IOPLUS_NO_ERROR: transfer completed.

NOTES:

This routine may be used to notify another processor or board that an event has occurred.

ioplus_malloc

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_malloc( unsigned long num_bytes,
                                  unsigned long **ptr );
```

DESCRIPTION:

ioplus_malloc allocates num_bytes of memory from a heap managed by the IOPlus. Note that this heap is shared among the IOPlus and the SPEs. The SPEs can cause memory to be allocated from this heap by commanding the IOPlus.

RETURN STATUS:

IOPLUS_NO_ERROR: memory allocated successfully.
IOPLUS_OUT_OF_MEMORY: not enough free memory in the heap.

NOTES:

This routine implements a multi-processor memory allocator.

ioplus_map_vme

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_map_vme( unsigned int             vmestart,
                                   IOPLUS_VME_ADDRESS_MODIFIER addr_mod,
                                   IOPLUS_VME_DATA_WIDTH       width,
                                   unsigned int               *pcistart );
```

DESCRIPTION:

ioplus_map_vme maps 8 MB of VME bus address space onto the PCI bus. The starting address of the VME bus memory is *vmestart*. The specified address modifier *addr_mod* and data width *width* are when accessing the VME bus using this mapping. The starting PCI address of the mapping is returned in *pcistart*. Once *ioplus_map_vme* has been called, the VME bus may be accesed by dereferencing a pointer, starting at address *pcistart*.

RETURN STATUS:

IOPLUS_NO_ERROR: success.
IOPLUS_INVALID_PARAM: *vmestart* not on a 64 kB boundary
IOPLUS_NO_RESOURCES_AVAILABLE: All 7 VME windows already allocated

NOTES:

vmestart must be on a 64 kilobyte boundary.

To perform a write to VME address 0x5000001C, first call *ioplus_map_vme()* with the *vmestart* parameter set to 0x50000000. Add 0x1C bytes to the pointer returned in *pcistart*, then dereference the pointer to perform the write.

ioplus_move_data

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_move_data( unsigned long src_address,
                                     unsigned long src_address_type,
                                     unsigned long src_stride,
                                     unsigned long src_options,
                                     unsigned long dest_address,
                                     unsigned long dest_address_type,
                                     unsigned long dest_stride,
                                     unsigned long dest_options,
                                     unsigned long num_words );
```

DESCRIPTION:

ioplus_move_data moves data from a source memory resource to a destination memory resource. These memory resources include global memory, SPE local memory, and VMEbus memory (i.e. other boards on the VMEbus). The routine determines the correct bus to use to perform the transfer (including PCI, and VMEbus). Note that optimal performance is obtained when using a stride of 1. DMA transfers will be performed when transferring between two PCI memory regions or between PCI memory and VMEbus memory.

Address Type is:

0x00000000: PCI address
0x00000001: VME address
0x00000002: IOPlus local address
0x00000003: SPE A local address
0x00000004: SPE B local address
0x00000005: SPE C local address
0x00000006: SPE D local address
0x00000007: SPE A PCI SDRAM address
0x00000008: SPE B PCI SDRAM address
0x00000009: SPE C PCI SDRAM address
0x0000000A: SPE D PCI SDRAM address

Options word is:

Bit Fields															
31	----	18	17	-	14	13	12	11	10	9	-	6	5	----------	0
Reserved (must be 0)			VME Data Size			VME Address Size		VME transfer mode		Bus Type			Set to address type (see above)		

Bus type:
- 0x00000000: VME
- 0x00000040: PCI
- 0x000000C0: Flash

VME transfer mode:
- 0x00000000: User / data address mode
- 0x00000400: User / program address mode
- 0x00000800: Supervisor / data address mode
- 0x00000C00: Supervisor / program address mode

VME address size:
- 0x00000000: A32
- 0x00001000: A24
- 0x00002000: A16

VME Data size:
- 0x00000000: D32
- 0x00004000: D16
- 0x00008000: D8
- 0x0000C000: D32BLT
- 0x00010000: D64BLT

RETURN STATUS:

IOPLUS_NO_ERROR: transfer performed successfully.
IOPLUS_BUS_ERROR: a bus error occurred while attempting the transfer.
IOPLUS_VME_DMA_IN_PROGRESS: the routine initiated a DMA transfer using the Universe II.
IOPLUS_PCI_DMA_IN_PROGRESS: the routine initiated a DMA transfer using the IOPlus DMA engine.

NOTES:

Optimal performance is obtained when both the source and destination have a stride of 1, and when transferring either between two PCI addresses, or between a PCI and VME address.

When the return indicates that a DMA is in progress, use the routine *ioplus_check_vme_dma_done* or *ioplus_check_pci_dma_done* to determine when the DMA has completed.

ioplus_pci_find_device

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_pci_find_device( unsigned int vendorID,
                                           unsigned int deviceID,
                                           unsigned int index,
                                           unsigned int *bus,
                                           unsigned int *device,
                                           unsigned int *function,
                                           PCI_CONFIG_DEVICE_HEADER *config );
```

DESCRIPTION:

Retrieve information for the *index* instance of the PCI device having the specified *vendorID* and *deviceID*. The *bus* number, *device* number, *function* number, and PCI device header information for the specified function are returned.

RETURN STATUS:

IOPLUS_NO_ERROR: success.
IOPLUS_DEVICE_NOT_FOUND: Device with specified vendor and device ID not found.

NOTES:

index specifies which instance of a device should be located. For instance, to find the first Universe II device, set *index* to 0. To find the third PCI-PCI bridge, set *index* to 2.

To determine the base addresses at which a PCI device is mapped, call the *ioplus_pci_find_device()* function, then inspect the *base0* through *base5* fields.

ioplus_read_data

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_read_data( unsigned int section_id,
                                     unsigned int offset,
                                     unsigned int stride,
                                     unsigned int option,
                                     unsigned int num_words,
                                     unsigned int *ptr,
                                     char         *name );
```

DESCRIPTION:

ioplus_read_data reads the data buffer referenced by *ptr* from the *offset* within the memory section specified by *section_id.* The data may be strided if desired. Num_words is the number of 32-bit words to transfer. Name is used only to access a named portion of the flash memory.

RETURN STATUS:

IOPLUS_NO_ERROR: data written successfully.
IOPLUS_INVALID_PARAM: Unrecognized parameter.

NOTES:

This routine is used to access memory regions by name without specifying their physical address.

A list of section Ids is provided in Table 5.7

Options: 0 – PCI bus, 2 – read from flash. The “name” field is meaningful only for flash read operations.

Refer to the command “CMD_READ_DATA” for additional information on using this command.

ioplus_realloc

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_realloc( unsigned long num_bytes,
                                   unsigned long **ptr );
```

DESCRIPTION:

ioplus_realloc resizes memory which was previously allocated by either *ioplus_malloc* or *ioplus_calloc.*

RETURN STATUS:

IOPLUS_NO_ERROR: memory allocated successfully.
IOPLUS_OUT_OF_MemORY: not enough free memory in the heap.

NOTES:

This routine should be called whenever you need to resize a previously allocated memory block. Note that the pointer returned by *ioplus_realloc* will most likely be different from the pointer input to this function. Note also that the contents of the previous memory are preserved.

ioplus_reset

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_reset( unsigned long reset_type,
                                 unsigned long reset_item );
```

DESCRIPTION:

ioplus_reset resets the specified device.

Reset types are:	0x00000001	Reserved
	0x00000002	Reserved
	0x00000004	Reset board

Reset Items are not applicable to the IXA family, and should be set to zero.

RETURN STATUS:

IOPLUS_NO_ERROR: reset performed successfully.
IOPLUS_INVALID_PARAM: Unknown reset type or reset item.

NOTES:

This routine may be used to reset on-board devices.

ioplus_toggle_led

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_toggle_led( unsigned long num_blinks,
                                      unsigned long duration );
```

DESCRIPTION:

ioplus_toggle_led toggles the red board status LED.

RETURN STATUS:

IOPLUS_NO_ERROR: transfer completed.

NOTES:

This routine can be used to indicate board failure, or to facilitate code debugging.

ioplus_unmap_vme

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_unmap_vme( unsigned int pcistart );
```

DESCRIPTION:

ioplus_unmap_vme destroys a mapping created by *ioplus_map_vme*, releasing all allocated resources. Always call *ioplus_unmap_vme* when the VME-PCI mapping is no longer needed.

RETURN STATUS:

IOPLUS_NO_ERROR: success.
IOPLUS_INVALID_PARAM: Mapping not found

NOTES:

The *pcistart* parameter should be set to the pointer returned by *ioplus_map_vme*.

ioplus_vme_peek

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_vme_peek( unsigned int vme_address,
                                    unsigned int *value );
```

DESCRIPTION:

Perform a single A32/D32 read from the specified *vme_address*, returning the *value* read.

RETURN STATUS:

IOPLUS_NO_ERROR: success.
IOPLUS_NO_RESOURCES_AVAILABLE: All VME windows already allocated

NOTES:

ioplus_vme_peek only performs A32/D32 transfers. If other types of transfers are needed, using *ioplus_map_vme* to create a VME-PCI address mapping.

ioplus_vme_poke

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_vme_poke( unsigned int vme_address,
                                    unsigned int value );
```

DESCRIPTION:

Perform a single A32/D32 write of *value* to the specified *vme_address*.

RETURN STATUS:

IOPLUS_NO_ERROR: success.
IOPLUS_NO_RESOURCES_AVAILABLE: All VME windows already allocated

NOTES:

ioplus_vme_poke only performs A32/D32 transfers. If other types of transfers are needed, using *ioplus_map_vme* to create a VME-PCI address mapping.

ioplus_vme_read

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_vme_read( unsigned int vme_address,
                                    unsigned int local_address,
                                    int          num_words );
```

DESCRIPTION:

Read *num_words* from *vme_address* into *local_address* by performing A32/D32 reads on the VMEbus.

RETURN STATUS:

IOPLUS_NO_ERROR: success.
IOPLUS_NO_RESOURCES_AVAILABLE: All VME windows already allocated

NOTES:

ioplus_vme_read only performs A32/D32 transfers. If other types of transfers are needed, using *ioplus_map_vme* to create a VME-PCI address mapping.

Enough memory must be allocated in *local_address* for performing the read.

DMAs are not performed by this function. If you want to use DMAs to perform the data transfer, use the *ioplus_move_data* function instead.

ioplus_vme_write

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_vme_write( unsigned int vme_address,
                                     unsigned int local_address,
                                     int          num_words );
```

DESCRIPTION:

Write *num_words* from *local_address* to *vme_address* by performing A32/D32 writes on the VMEbus.

RETURN STATUS:

IOPLUS_NO_ERROR: success.
IOPLUS_NO_RESOURCES_AVAILABLE: All VME windows already allocated

NOTES:

ioplus_vme_write only performs A32/D32 transfers. If other types of transfers are needed, using *ioplus_map_vme* to create a VME-PCI address mapping.

Enough memory must be allocated in *local_address* for performing the read.

DMAs are not performed by this function. If you want to use DMAs to perform the data transfer, use the *ioplus_move_data* function instead.

ioplus_write_data

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_write_data( unsigned int section_id,
                                      unsigned int offset,
                                      unsigned int stride,
                                      unsigned int option,
                                      unsigned int num_words,
                                      unsigned int *ptr );
```

DESCRIPTION:

ioplus_write_data writes the data buffer referenced by *ptr* to the *offset* within the memory section specified by *section_id.* The data may be strided if desired. Num_words is the number of 32-bit words to transfer.

RETURN STATUS:

IOPLUS_NO_ERROR: data written successfully.
IOPLUS_INVALID_PARAM: Unrecognized parameter.

NOTES:

This routine is used to access memory regions by name without specifying their physical address.

A list of section Ids is provided in Table 5.7

Options: 0 – PCI bus, 2 – initial write to flash, 3 – final write to flash. See a description of CMD_WRITE_DATA command in Chapter 5 for a brief summary of the flash write process.

Refer to the command "CMD_READ_DATA" for additional information on using this command.

Chapter 7: Programming the SPEs

7.1 SPE Software Development

Developing software programs to execute on the SPE's on an IXA7 board requires IXAtools, a development environment that can link with C applications, and either a JTAG emulator or a VxWorks workstation with an Ethernet connection to the IXA7.

IXAtools includes several files used in the development process specifically for IXA boards. The IXAbsp object library(libixa.a) provides C-callable functions for managing many board-specific resources like VMEbus access, board interrupts and semaphores, and DMA's. Several example programs that demonstrate how to use the IXAbsp functions are included in the IXAtools release along with *makefiles* for compiling.

> Dy 4 Systems does not provide a make utility as a part of our software. If a make utility is not available, you will either have to enter the compiler commands at the command prompt or construct a batch file to do the compile and link process.

IXAtools also provides for the SPEs: a VxWorks compatible BSP, a stdio library, and a common boot code component. The common boot code is a key element in running applications on a SPE pair that share a local memory.

In addition, IXAtools includes standard C libraries for developing SPE applications without an OS environment like VxWorks. See Applications Note #26 – "Creating a Standalone, Non-OS program for the SPE's" for more information.

7.2 The Common Boot Code

The common boot code (CBC) determines how each processor within a pair shares its memory. Please read this section if you need to change the configuration of SPE memory, or if you need to map in large sections of address space for shared memory buffers or other devices.

The IXA architecture incorporates four PowerPC processors, organized as two pairs. Each pair has its own bridge device and physical memory. Both processors within a pair see this memory as a linearly addressed space, starting at location zero.

The PowerPC exception vector table (EVT) lives in physical memory starting at location zero. Because both processors in a pair see the exact same physical memory, both processors share the exception vector table. This is highly undesirable, because it is often necessary for each processor to take different actions for each exception. For example, the external interrupt event (0x500) indicates that a device has asserted an interrupt to a processor. Since each processor has its own separate interrupt inputs, each processor will need to take different actions when an external interrupt occurs.

The common boot code was designed to establish separate environments for each processor before the processors run their main application. (The main application can be a user program, or an operating system running several user programs). The common boot code does this in a manner that is transparent to the application.

How it Works

At initialization, each processor within a pair sees physical memory starting at location 0 and extending through the end of physical memory, 32, 64 or 128 megabytes in length. The only reserved area is the EVT, defined at locations 0 through 0x2fff. The processors share the EVT and the remainder of physical memory. There is no protection to keep one processor from disrupting the operation of the other. This is shown in Table 7.1.

Table 7.1 - SPE Local Memory Map with no CBC (64 MB)

Begin	End	Processor 1	Processor 2
0x00000000	0x00002FFF	physical EVT	physical EVT
0x00003000	0x02FFFFFF	available memory	available memory

The CBC establishes an environment where each processor has separate, logical address spaces. Each space includes an EVT, starting at logical address zero but mapped to some other physical address. The CBC does this by enabling the memory management unit (MMU), and establishing a page translation table. Physical memory is partitioned into logical segments, as illustrated in table 7.2.

Table 7.2 - SPE Local Memory Map with CBC (64 MB)

Begin	End	Processor 1	Processor 2
0x00000000	0x00002FFF	physical EVT	physical EVT
0x00008000	0x0000FFFF	common boot code	common boot code
0x00040000	0x01F6FFFF	processor 1 memory	
0x01F77000	0x01F7BFFF	processor 1 EVT	
0x01F7c000	0x01F7FFFF	processor 1 stack	
0x01F80000	0x01FFFFFF	processor 1 page table	
0x02000000	0x02F6BFFF		processor 2 memory
0x02F77000	0x02F7BFFF		proc 2 EVT
0x02F78000	0x02F7FFFF		proc 2 stack
0x02F80000	0x02FFFFFF		proc 2 page table

Each processor is assigned its own regions of memory. The MMU limits access to the assigned regions, so that attempts to access the other processor's memory result in an exception.

Each processor sees a single, shared copy of the common boot code starting at location 0x8000. This region of memory is write-protected to insure the integrity of the common boot code. Each processor also shares the single physical EVT located at physical address zero. This EVT is not mapped into either processor's logical address space, so it is protected from unauthorized alteration. Instead, each processor includes its own, separate, logical EVT, which starts at logical address zero.

When an exception occurs, the PPC automatically disables the MMU, and branches to the appropriate offset within the physical EVT. The CBC takes over at this point. It enables the MMU, and branches to the appropriate offset within the logical EVT. This processing adds about 10 instructions to the interrupt service overhead. This mapping is transparent to the application. This method has been successfully tested with the vxWorks operating system.

The MMU requires that each processor construct a *page table* for address translation. The page table has specific alignment and size requirements. A page table must have a size that is a power of two between 64k and 32 megabytes. The page table must start at a physical address that is a multiple of its size. For example, a page table having size 128K bytes must start on a 128K physical boundary. In order to meet these alignment and size requirements, the CBC places the page tables at the end of each processor's memory space. The MMU does not require page tables to be mapped into a processor's logical address space; they are accessed by the MMU based on their physical address. This provides protection from unauthorized alteration. The contents of the page tables may be interrogated and altered through IXA board support library function calls that alter physical memory.

> Although the physical EVT is 0x3000 bytes in length, the logical EVT is actually 0x5000 bytes in length. This length remaps the EVT, plus some processor-specific reserved work areas used by the vxWorks operating system.

> Locations 0x2000 - 0x2100 within the logical and physical EVTs are used to maintain CBC specific information. These locations must not be used or altered by the applications program.

Using the CBC

Initialization

The IOPlus automatically loads the CBC into memory and starts the CBC on reset. No special user action is required to start the CBC. The CBC can be disabled using the Ethernet flash burn utility. When jumper J7 is removed (or, on the IXA-4 board, when SW-7 is OFF), the IOP does not load the CBC or applications programs into the SPEs.

Version Identification

By displaying the contents of the common boot code starting at location 0x8000, you may examine the version of the CBC. This information is stored in ASCII, and immediately follows a branch around the ASCII data. The copyright notice includes a compile time and date.

Loading Applications from Flash Memory

After loading and starting the CBC, the CBC initializes memory data structures, establishes each processor's environment, and then waits for a download from the IOP.

The IOPlus examines its flash memory for SPE applications. When found, the IOPlus waits until the CBC has completed its initialization. The IOPlus then downloads the applications from flash into SPE memory, and informs the CBC to jump to the application.

Loading Applications using an Emulator

After initialization, the CBC enters a polling loop, waiting to receive the starting address of the application. The CBC polls physical location 0x2000 (for processor 1 of each pair, or 0x2008 for processor 2 of each pair), for a nonzero value. The polling loop is designed to minimize processor to memory bandwidth requirements. When the CBC detects a nonzero value at the poll location, it assumes the value is the application entry point, and it calls the function at that entry point. While polling, the CBC displays a distinctive LED blink pattern. This pattern stops when the CBC exits the polling loop for any reason.

Executable images can be loaded using an emulator using the following steps:

1. Using the emulator, begin execution at location 0x8000, the CBC.

2. After the front panel LED begins flashing, stop the processor.

3. Download the application. Do not disturb the physical EVT, page tables, CBC, or MMU configuration.

4. Deposit the application starting address at location 0x2000 (for processor 1 of 2), or 0x2008.

5. Set any breakpoints within the application.

6. Continue execution. The CBC will detect the nonzero starting address and call the application.

The Application Environment

When the CBC jumps to the application, caches (instruction and data) are enabled, and address translation (the MMU) is enabled for instructions and data. The application should NOT disable the memory management unit, as this defeats the protection provided by the CBC. Memory mapping can be adjusted through IXA board support library function calls. The application may disable or flush cache at any time.

The stack pointer is located within the initial stack area. This area is mapped so that its physical and logical addresses are equal. It is normal for an application to switch to a larger stack area within the main memory region.

> Note to vxWorks users: vxWorks must be configured to run with the memory management unit disabled. When configured in this mode, vxWorks will not alter the state of the MMU, leaving it enabled. Enabling the MMU under VxWorks will cause a re-initialization of the MMU, and will interfere with the memory segregation provided by the CBC.

Returning to the CBC

The application can return to the CBC. If the application returns, the CBC will re-initialize, and wait for another download. The application can also force re-initialization by disabling interrupts and jumping to location 0x8000.

Disabling the CBC

An application need not use the CBC. However, the applications programmers must understand that if the CBC is not used, both processors share the EVT and all other memory resources. Applications may disable the CBC by clearing the address translation enable bits within the MSR. The application can then re-initialize the MMU if desired.

Configuring the CBC

Users of this section must have a good understanding of memory management in the PowerPC, including BAT register use, and page table construction. The data structures employed by the PowerPC are complicated and sensitive. In a multi-processor environment with large caches and memory mapping, it is possible to create an environment that has the appearance of working but has some latent bugs that are very difficult to repeat and characterize.

By default, the CBC comes configured to provide each processor with access to about half of the physical memory. The file *cbconfig.c* contains memory size information for all processors. This can be altered. To alter the CBC configuration, perform the following steps:

1. Create a backup copy of *cbc.hex* and *cbconfig.c*, and edit *cbconfig.c* to reflect the new configuration. Data structures likely to need customization are described in the next section.

2. Within the *cbc* directory, *make*.

3. Using the Ethernet flash burn utility, burn the new *cbc.hex* to flash memory.

4. Reset the board. If the familiar blinking pattern appears, you are done. Otherwise, remove jumper 7 (or change the state of SW-7 on the the IXA-4), and reset the card. This allows the IOP to reboot without loading or starting the SPE processors.

5. Correct *cbconfig.c*, *remake*, *reburn*. Restore the state of jumper 7 (or SW-7), and reset the board.

Editing the CBC

Data structures and constants within *cbconfig.c* guide the *cbc* as it constructs the page tables and initializes the MMU. Before editing the file, you should understand the role of these constants and data structures.

Physical Memory Size

PHYS_MEMORY_SIZE is a constant that determines the amount of memory to be divided between the two processors in each pair. For example, if PHYS_MEMORY_SIZE is 0x0200_0000 (32 megabytes), the CBC allocates 16 megabytes to the A processor, and a separate 16 megabytes to the B processor. Processors C and D have a similar allocation from their memory.

PHYS_MEMORY_SIZE should be a multiple of four megabytes, and must not exceed the size of physical memory. This constant is 32 megabytes in the delivered software. Setting this value to a number smaller than the size of physical memory results in an unmapped region of "residual" memory.

If your sole objective in editing the CBC is to increase the amount of memory allocated to each processor, editing this constant is the only change required. Alternately, you may use one of the precompiled *cbc* files for 32, 64, and 128 megabyte configurations, called *cbc32.hex*, *cbc64.hex*, and *cbc128.hex*.

Page Table Size

The page table must be sized to accommodate all pages that a processor will need. This includes pages mapped by the CBC, as well as pages dynamically mapped using functions like *ixa_mmu_map_pages*. The CBC determines the page table size based on the processor's memory size as determined from PHYS_MEMORY_SIZE, plus *ADDITIONAL_PAGE_MEMORY_X*.

ADDITIONAL_PAGE_MEMORY_X, where X is A, B, C, or D, is a constant estimating the total size (in bytes) of other memory regions that will be mapped using page tables. These constants may need to be adjusted if a processor needs to access additional space using pages.

The initial value of this constant allows for 8 megabytes of memory to be mapped through regions or ixa_mmu function calls. This is more than sufficient to accommodate most needs. Large PMC devices, and large shared buffer areas, may dictate that you increase these values.

Processor Configuration Tables

Each processor has a *cbc_config_X* table, where X is A, B, C, or D. This table, illustrated below, points to other data structures that guide the *cbc* in the construction of the page tables and initialization of the MMU. You will probably never change this data structure. However, you may need to alter some of the data structures referenced by this data structure. The bat register data structure defines the initialization of block address translation registers. The cbc uses data in the region tables to construct entries in the page table. Private regions are unique to a particular processor, while all processors will have visibility into the shared regions. You will probably edit the bat register data structure or one of the region tables.

```
struct CBC_CONFIG _cbc_config_A =
{
    STACK_A+STACK_SIZE_A,     /* initial stack pointer   */
    proca_batregs,            /* bat registers           */
    proca_regions,            /* private memory regions  */
    shared_regions,           /* shared memory regions   */
    MAIN_BEGIN_A_PHYS-MAIN_BEGIN_A_LOG  /* log->phys offset */
};
```

BAT Register Table

Each processor has its own bat register table. This table has exactly 8 entries, corresponding to four instruction BAT registers, and four data BAT registers. The number and use of BAT registers is defined by the PPC hardware, and cannot be adjusted.

Blocks are more efficient than pages, but they are few in number, and they must meet more rigid address and size requirements. A block is a contiguous region of physical memory having a length between 128KB and 256MB. The length must be a power of two. The starting logical and physical addresses must be a multiple of the block size. Finally, the entire block shares the same caching and access attributes. Memory areas meeting these requirements can be mapped using blocks.

The CBC loads this table into the corresponding BAT registers at initialization time. Unused entries must be set to BATREG_NULL. Other entries must be set to valid BAT register contents. The macros BATREG_S, BATREG_P, and BATREG_SP define BAT register contents for memory accessible from supervisor state, problem state, or both states.

Note that two of the data BAT register entries are already committed. The first allows access to global (IOP) memory. The second provides access to backplane (VME or CPCI) bridge and window space. Unused data BAT registers can be modified by application software through the function *ixa_mmu_map_block*.

> BAT registers must not define overlapping regions of memory. An exception results when a processor accesses a region of memory mapped by more than one BAT register.

```
static unsigned long proca_batregs[] =
{
    BATREG_NULL,              /* instruction BATS     */
    BATREG_NULL,
    BATREG_NULL,
    BATREG_NULL,

    BATREG_GLOBAL,            /* data BATS            */
    BATREG_BACKPLANE,
    BATREG_NULL,
    BATREG_NULL
};
```

Region Tables

Region tables define areas of memory to be mapped using page table entries. Each processor has a table describing private regions. Each processor also references a table describing shared regions. Region tables must end with the special entry REGION_END.

Individual regions are described using a logical address, a physical address, a length, and access attributes. The address and length parameters must be a multiple of 4096.

Regions must not overlap each other. The special logical address "PAGE_TABLE" identifies a region as a page table. The CBC does not map in such a region, but it does need to know its physical address and length.

Attributes determine whether the region is readable, writable, or executable. Attributes consist of protection bits, and cache control bits. Protection bits can be one of the following:

ATTR_RWX: the region is readable, writable, and executable
ATTR_RO: the region is readable, and executable.

Cache control attributes are specified by OR'ing or adding one or more of the following values:

ATTR_W: if cache is enabled, the region is treated as write-through
ATTR_I: cache is inhibited for the region
ATTR_M: memory coherency is required when accessing the region
ATTR_G: access to the region is guarded

Attributes ATTR_G + ATTR_I should be used for I/O control register spaces. The guarded attribute prevents the processor from re-ordering accesses to the memory area.

ATTR_M can be used for regions of memory shared between processors A and B (or C and D). This attribute causes certain cache management machine operations to synchronize cache contents between the two processors.

ATTR_W can be used at the discretion of the applications engineer. This causes write operations to update the contents of cache and the target memory, rather than waiting until a cache flush operation is required. This attribute is desirable for writing output data to regions of memory that other processors can examine.

The most efficient memory access occurs when none of ATTR_W, ATTR_I, ATTR_M, and ATTR_G are employed. The resulting memory accesses will be fully cached (assuming cache is enabled). Private areas of memory that are not used for I/O or inter-processor communication should be set to this mode.
ATTR_CLEAR is a pseudo-attribute. When the CBC sees this attribute, it zeroes out the memory region. This attribute must not be specified for the stack entry, since the stack contains live data while the CBC executes.

The attributes of pages can be changed by the application during program execution. This is accomplished through a board support library function that changes page attributes. A common practice is to allocate an I/O buffer, and then change the attributes of the buffer to be cache inhibited and guarded. Of course, the buffer must start at a 4k boundary and be a multiple of 4k in length.

The region table for processor A is illustrated below.

```
static struct REGION proca_regions[] =
{
    {0x00000000,  EVT_A,      EVT_SIZE,     ATTR_CLEAR+ATTR_RWX},

    {MAIN_BEGIN_A_LOG, MAIN_BEGIN_A_PHYS,
                              MAIN_SIZE_A,  ATTR_CLEAR+ATTR_RWX},

    {STACK_A,     STACK_A,    STACK_SIZE_A, ATTR_RWX},
    {PAGE_TABLE,  PGTBL_A,    PGTBL_SIZE_A, 0},
    {REGION_END,  REGION_END, 0,            0}
};
```

Adding a Mapped Region or Block

The SPE processors can only access those address spaces defined by BAT registers and page tables. You may have to add address space so that a processor can:

- Access memory owned by the other SPE processors
- Access PMC control registers or memory

When adding address space, consider the following trade-offs:

1. Does the memory area lend itself to block address mapping? (Is the size a power of two between 128K and 256M, and does the area start on an address that is a multiple of its size?) If so, are data BAT registers available? If the answer to both of these questions is "yes", you can use block address translation. Otherwise, you must use page address translation.

2. Can the memory be mapped dynamically (by calling *ixa_mmu* library functions), or must it be mapped statically? If it can be mapped dynamically, then you may not need to edit the CBC files at all.

3. If the memory to be mapped dynamically is mapped using pages, is the size of the area greater than 1 megabyte? If so, you will probably need to edit the CBC file and increase ADDITIONAL_PAGE_MEMORY_X for each processor that needs to map the new space.

4. Is the memory mapped statically, and does the memory space need to be visible to all processors? If so, and if you are using blocks, you must edit the BAT registers. If so, and the memory is not block memory, you may add the region to the shared regions table. Otherwise, add an entry to the processor's private region table.

5. For regions or blocks, select the caching attributes and protection attributes that match the application's need. Some attributes, like inhibiting cache, make sharing memory easier but accessing memory becomes slower.

To add a region or block, determine the size, physical address, and logical address of the region. The physical and logical addresses need not be equal, but often this is the most practical approach. Round addresses down to the next page or block boundary, and round sizes up to the page size or block size. Determine the appropriate attributes. Then, edit the BAT register table, the shared region table, or the private region table. Remember the following rules:

1. There must be four instruction BAT registers and four data BAT registers in the bat register table: this cannot be changed.

2. Blocks of data memory defined by BAT registers may not overlap. Blocks of instruction memory defined by BAT registers may not overlap. However, it is permissible to have a data block overlap an instruction block.

3. Insure that starting addresses and sizes are rounded to their required boundaries.

4. Regions, which are mapped using pages, may not overlap each other. (However, it is permissible for a region and a BAT register to overlap – the processor uses the BAT register mapping in such a case.)

5. The page table, determined by the memory size plus the constant ADDITIONAL_PAGE_MEMORY_x, must be large enough to accommodate all defined regions plus any dynamically defined areas of memory. When adding regions, you may need to increase this constant.

6. The special entry REGION_END must be the last entry in a region table.

Example 1: To add a region to processor A to allow it to access 8K bytes of PMC control register space located at address 0xc000_0000, the new region table would look like this, with the new line italicized:

```
static struct REGION proca_regions[] =
{
    {0x00000000,  EVT_A,      EVT_SIZE,  ATTR_CLEAR+ATTR_RWX},

    {MAIN_BEGIN_A_LOG, MAIN_BEGIN_A_PHYS,
                          MAIN_SIZE_A,  ATTR_CLEAR+ATTR_RWX},

    {0xc0000000,  0xc0000000, 8192,      ATTR_I+ATTR_G+ATTR_RWX},

    {STACK_A,    STACK_A,  STACK_SIZE_A, ATTR_RWX},
    {PAGE_TABLE, PGTBL_A,  PGTBL_SIZE_A, 0},
    {REGION_END, REGION_END, 0,          0}
};
```

Example 2: Assume the PMC has a large memory buffer requiring 64MB of address space. Using a block would be more appropriate. Defining the BAT register contents, and altering the BAT register table, gives the following results, with changes italicized:

```
/*
 *  BATREG for PMC starting at 0xc000_0000 having
 *  Address space requirements of 0x0400_0000 (64M)
 */

#define BATREG_PMC \
     BATREG_SP(0xc0000000, 0xc0000000, 0x04000000,  \
     BATREG_G+BATREG_I+BATREG_RW)

static unsigned long proca_batregs[] =
{
    BATREG_NULL,              /* instruction BATS     */
    BATREG_NULL,
    BATREG_NULL,
    BATREG_NULL,

    BATREG_GLOBAL,            /* data BATS            */
    BATREG_BACKPLANE,
    BATREG_PMC,
    BATREG_NULL
};
```

Error Conditions.

During initialization, the CBC checks the following requirements. If all requirements are met, the CBC completes its initialization, and waits for a starting address. If any condition is not met, the CBC stops with an error code. The CBC checks the following during initialization:

1. Region addresses, including the page table, EVT, and stack, must be multiples of 4k, and start on a 4k boundary.

2. A nonzero physical address for the EVT must be specified.

3. The EVT map length must be at least 0x4000 in length.

4. The page table size must be a power of two, between 128k and 32 megabytes.

5. The page table start address must be a multiple of its size.

6. Mapped regions must not overlap each other (although they may overlap with BAT registers.)

If all of these conditions are met, the CBC will complete initialization, and wait for an application start address.

VxWorks Memory Sizing Implications

The size of a memory region used by VxWorks varies with the CBC configuration. The BSP for VxWorks interrogates the CBC in order to determine the size of the region in which VxWorks lives. Therefore, changes in the CBC configuration are automatically reflected in the BSP, with no need to change the BSP.

7.3 Performance Monitoring Capabilities

Software Development Support

Several performance monitoring capabilities are introduced with release 2.4 of IXATools. These capabilities allow the developer access to the interal performance monitoring features of the 74x0, determine the junction temperature of the processor, and dynamically control the instruction execution rate of the processor.

The 74x0 processors include performance monitoring registers. These registers can be configured to count internal processor events, such as cache hits and misses, instructions

completed, branch operations, and so forth. The processors allow the user to specify four events to be counted chosen from a list of over 100 separate events.

Enabling the performance monitoring does not interfere with real-time execution.

IXATools provides a series of functions (ixa_pm_xxxx) that control the processor's performance monitoring features. Using these functions, software designers define events. An event has a specific starting point within the code, and a specific ending point. Performance monitoring data and timing data are collected for each event. Events can be nested or overlap. Collected data is analyzed and displayed using the performance monitor display functions.

Runtime Support

Also introduced in release 2.4 is the temperature read function (*ixa_temp_read*), and cache throttling functions *(ixa_cache_throttle_read, ixa_cache_throttle_write).*

The temperature read function determines the junction temperature of the part by reading the processor's thermal assist unit (TAU) registers. The accuracy of the TAU is a function of the processor family and may vary between revisions of parts.

Cache throttling can be used to control power use, and, to a limited extent, component temperature. By default, the instruction cache feeds instructions to the execution unit as fast as the execution unit can process them. By adjusting the cache throttling value, the execution rate of the processor can be controlled.

Writing any value to the cache throttling register enables dynamic power management (DPM). DPM places the processor in a mode where unused execution units are not clocked. DPM results in some power savings, even if the cache throttle value is zero (indicating full-speed execution). There are no performance penalties for enabling DPM.

7.4 VxWorks and the SPEs

The VxWorks BSP for SPEs relies on the IOPlus to initialize global board resources and on the Common Boot Code (CBC) to perform initialization for the majority of the local processor resources. Table 7.3 presents a summary of the BSPs features and Table 7.4 provides a list of the BSPs functions. Many BSP functions have IXA counterparts described later in this section. For example, sysBusOpen is the same as ixa_VME_open. For such functions, we recommend using the VxWorks version of the function when writing applications intended to run under VxWorks. However, this document often contains more detail regarding those functions and their parameters, listed under the IXA equivalent function name, than is provided by the VxWorks documentation.

Table 7.3 - SPE VxWorks BSP Features

Feature	Description
Boot ROM Images	N/A: IOP downloads SPE executables
Boot Devices	Flash memory; shared memorym
VxWorks images	vxWorks, vxWorks.st, vxWorks5_2
MMU	- disabled; MMU control is provided by the CBC - uses BAT registers and PTEs for address translation
Cache mode	Instruction, data, L1 & L2 cache in copyback cache mode
Sysclock	using PPC Decrementer
AuxClock	using timer 1 of EPIC
Serial	using shared memory protocol
Timestamp	using timer 0 of EPIC
Ethernet	shared memory network using PCI bus

Table 7.4 - SPE VxWorks BSP Functions

Function	Description
pciConfigInByte()	Read one byte from configuration space
pciConfigInLong()	Read one long from configuration space
pciConfigInWord()	Read one word from configuration space
pciConfigOutByte()	Write one byte to configuration space
pciConfigOutLong()	Write one long to configuration space
pciConfigOutWord()	Write one word to configuration space
pciDeviceShow()	Print information about PCI devices
pciFindClass()	Find the nth device with a given class
pciFindClassShow()	Find device by class, then print
pciFindDevice()	Find the nth device with a given device and vendor ID
pciFindDeviceShow()	Find device by ID then print
pciHeaderShow()	Print a header of a specied PCI device
pciInt()	Interrupt handler for shared PC interrupts
pciIntConnect()	Connect the interrupt handler to the PCI interrupt
pciIntDisconnect()	Disconnect the interrupt handler from a PCI interrupt
pciIomapLibInit()	Initialize the configuration access method and address
pciSpecialCycle()	Generate a PCI special cycle
sysAuxClkConnect()	connect a routine to the auxiliary clock interrupt
sysAuxClkDisable()	turn auxiliary clock interrupts off
sysAuxClkEnable()	turn auxiliary clock interrupts on
sysAuxClkInt()	handle auxiliary clock interrupts
sysAuxClkRateGet()	get the auxiliary clock rate
sysAuxClkRateSet()	set the auxiliary clock rate
sysBspRev()	return the bsp version and the bsp revision number
sysBusClose()	Close a VME master window
sysBusDmaIn()	Start VME dma input operation
sysBusDmaOut()	Start VME dma output operation
sysBusIntAck()	acknowledge/clear interrupt
sysBusIntGen()	generate an interrupt
sysBusMapAddr()	Map a VME address within a window to a local address
sysBusOpen()	Open a VME master window
sysBusPeek()	Read a single byte, word or long from VME bus

sysBusPoke()	Write a single byte, word or long to VME bus
sysBusProbe()	Determine if an address is accessible
sysBusRead()	Read VME data into local memory
sysBusReMap()	Reset the mapping parameters for a window
sysBusTas()	test and set a location across the VMEbus
sysBusToLocalAdrs()	convert bus address to local address
sysBusWrite()	Write VME data into local memory
sysClkConnect()	connect a routine to the system clock interrupt
sysClkDisable()	turn off system clock interrupts
sysClkEnable()	turn on system clock interrupts
sysClkRateGet()	get the system clock rate
sysClkRateSet()	set the system clock rate
sysIntDisable()	disable interrupts
sysIntEnable()	enable interrupts
sysLocalToBusAdrs()	convert local address to bus address
sysMailboxConnect()	Install an interrupt service routine for a mailbox
sysMailboxDisable()	Disable a mailbox interrupt
sysMailboxEnable()	Enable a mailbox interrupt
sysMailboxPost()	Post to a local mailbox, generating an interrupt
sysMemTop()	get the address of the top of VxWorks memory
sysModel()	return the model name of the target card
sysNvRamGet()	get the contents of non-volatile RAM
sysNvRamSet()	write to non-volatile RAM
sysPhysMemTop()	get the address of the top of physical memory
sysProcNumGet()	get the processor number
sysProcNumSet()	set the processor number
sysSerialChanGet()	get the SIO_CHAN device associated with a serial channel
sysTimestamp()	get the timestamp timer tick count
sysTimestampConnect()	connect a user routine to the timestamp interrupt
sysTimestampDisable()	disable the timestamp timer
sysTimestampEnable()	initialize and enable the timestamp timer
sysTimestampFreq()	get the timestamp timer clock frequency
sysTimestampLock()	get the timestamp tick counter
sysTimestampPeriod()	get the timestamp timer period
sysToggleLed()	turn the status LED on or off
sysToMonitor()	transfer control to the ROM monitor

7.5 Function Reference

The following is a description of the functions provided in the IXAbsp and the IXAio. They are organized in alphabetical order. In cases where several functions are grouped together into one description they too are organized alphabetically. The naming convention of the functions will tend to group them together into related services, i.e., *ixa_VME_**** pulls all the VMEbus support functions together. Utility functions are scattered throughout.

getchar

CALLING SEQUENCE:

```
#include <ixa.h>

int getchar (void);
```

DESCRIPTION:

The function *getchar* reads the next character, *c*, from the serial port. If no character is available then the caller is suspended.

RETURN STATUS:

Character value

ixa_cache_enable

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_cache_enable (int type);
```

DESCRIPTION:

The function *ixa_cache_enable* enables L1 instruction cache, L1 data cache, L2 cache, or all as specified by *type*. The parameter, *type*, may be one of the following:

CACHE_DATA	enable data cache
CACHE_INSTR	enable instruction cache
CACHE_L2	enable the L2 cache
CACHE_ALL	enable both instruction cache and data cache

Please note that on power-up, the IXA board boots and then runs the Common Boot Code. This leaves the board in a state with external interrupts disabled and all caches (instruction, data, and L2) enabled.

RETURN STATUS:

None

ixa_cache_disable

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_cache_disable (int type);
```

DESCRIPTION:

The function *ixa_cache_disable* disables L1 instruction cache, L1 data cache, the L2 cache, or all caches as specified by *type*. The parameter, *type*, may be one of the following:

CACHE_DATA	disable data cache
CACHE_INSTR	disable instruction cache
CACHE_L2	disable the L2 cache
CACHE_ALL	disable all caches

After the data cache is disabled, it is flushed and then invalidated. The instruction cache is invalidated but not flushed.

RETURN STATUS:

None

ixa_cache_flush

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_cache_flush (void *addr,
                      long  n);
```

DESCRIPTION:

The function *ixa_cache_flush* flushes *n* bytes of data cache beginning at address *addr*.

RETURN STATUS:

None

ixa_cache_invalidate

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_cache_invalidate (void *addr,
                           long  n);
```

DESCRIPTION:

The function *ixa_cache_invalidate* invalidates *n* bytes of data cache beginning at address *addr.*

RETURN STATUS:

None

ixa_cache_inv_all

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_cache_inv_all (int type);
```

DESCRIPTION:

The function *ixa_cache_inv_all* invalidates the entire L1 instruction cache, L1 data cache, or both as specified by *type*. If the L2 cache is enabled, it is invalidated also. The parameter, *type*, may be one of the following:

CACHE_DATA	enable data cache
CACHE_INSTR	enable instruction cache
CACHE_ALL	enable both instruction cache and data cache

RETURN STATUS:

None

ixa_cache_sync

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_cache_sync (void);
```

DESCRIPTION:

The function *ixa_cache_sync* synchronizes the data and instruction caches. This function is required when loading executable data into a memory area. Because data from this area may still live in the data cache, it must be forced to memory. Then, the instruction cache is invalidated, forcing the instruction cache to pick up the new data from memory.

RETURN STATUS:

None

ixa_cache_throttle_read

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_cache_throttle_read ( void );
```

DESCRIPTION:

The function *ixa_cache_throttle_read* reads the delay interval (expressed in clock cycles) between consecutive instruction forwardings from the instruction cache to the instruction dispatcher.

RETURN STATUS:

Number of clock cycles between consecutive instruction forwardings.

Returns –1 if the processor does not support cache throttling.

SEE ALSO:

ixa_cache_throttle_write

ixa_cache_throttle_write

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_cache_throttle_write ( int delay );
```

DESCRIPTION:

The function *ixa_cache_throttle_write* places the processor in *dynamic power mode* and writes a new delay interval (expressed in clock cycles) between consecutive instruction forwardings from the instruction cache to the instruction dispatcher. This new delay value should be in the range 0 to 255. If the new delay value is less than zero then *ixa_cache_throttle_write* writes 0. Also, if the new value is greater than 255 then *ixa_cache_throttle_write* writes 255.

A value of zero provides maximum processor performance. Nonzero values reduce processor performance and processor power requirements. In some environments, it may be possible to control device power consumption and junction temperature by adjusting the cache throttle value.

RETURN STATUS:

Number of clock cycles between consecutive instruction forwardings.

Returns –1 if the processor does not support cache throttling

SEE ALSO:

ixa_cache_throttle_read, ixa_temp_read.

See the MPC7400 or MPC7410 Risc Microprocessor User's Manuals for a complete description of *dynamic power mode.*

ixa_cmd_VME_error

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_VME_error (CMD_ERROR   *cmd_err);
```

DESCRIPTION:

The function *ixa_cmd_VME_error* returns the error status of an *ixa_cmd_VME_read* or *ixa_cmd_VME_write* operation. For more details, see *ixa_cmd_error*.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_CMD_CHANNEL_BUSY: command channel is active.

ixa_cmd_VME_read, ixa_cmd_VME_write

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_VME_read ( void         *vme_addr,
                                 void         *brd_addr,
                                 unsigned int brd_addr_opt,
                                 unsigned int addr_space,
                                 unsigned int addr_mod,
                                 unsigned int data_size,
                                 unsigned int count,
                                 unsigned int mode);

IXABSP_STATUS ixa_cmd_VME_write ( void         *vme_addr,
                                  void         *brd_addr,
                                  unsigned int brd_addr_opt,
                                  unsigned int addr_space,
                                  unsigned int addr_mod,
                                  unsigned int data_size,
                                  unsigned int count,
                                  unsigned int mode);
```

DESCRIPTION:

ixa_cmd_VME_read and *ixa_cmd_VME_write* perform master VME transfers. These functions send a command to the IOPlus to perform the transfer, in contrast to the *ixa_VME_read* and *ixa_VME_write* functions which processor access the VME interface chip directly. *vme_addr* is the starting VME address and *brd_addr* is the starting board address of the transfer. The *brd_addr* is further defined by the *brd_addr_opt* that specifies whether the address is a PCI address (*PCI_ADDR)* or a SPE address (*LOCAL_ADDR*). The VME access characteristics are determined by the following parameters:

addr_space – specifies address space. Valid values are A16, A24, A32, A64, ACR_CSR, AUSER1, or AUSER2.
addr_mod – specifies address modifier. Valid values are SUPER_PRG_AM, SUPER_DATA_AM, USER_PRG_AM, USER_DATA_AM.
data_size – specifies transfer size. Valid values are D8, D16, D32, D32BLT or D64BLT.

The number of data elements to transfer is specified by *count. mode* indicates whether the function waits until the transfer is complete before returning (SYNC), or returns after passing the request on to the IOPlus (ASYNC). In the ASYNC mode, the status of the VME transfer can be determined by calling the *ixa_cmd_VME_status* function.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_VME_INVALID_ADDR_SPACE: invalid *addr_space* parameter.
IXABSP_VME_INVALID_AM: invalid *addr_mod* parameter.
IXABSP_VME_INVALID_DATA_SIZE: invalid *data_size* parameter.
IXABSP_CMD_ERROR: in SYNC mode, a command channel error occurred.

ixa_cmd_VME_status

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_VME_status (unsigned int *status);
```

DESCRIPTION:

The function *ixa_cmd_VME_status* returns the status of an asynchronous transfer initiated by the *ixa_cmd_VME_read* or *ixa_cmd_VME_write* function. One of the following status values will be written to the *status* pointer:

CMD_STATUS_ACTIVE – if the request is still being processed by the IOPlus
CMD_STATUS_DONE – if the transfer is complete
CMD_STATUS_ERROR – if an error occurred during the transfer

If an error status is indicated, more information regarding the error can be obtained by calling the *ixa_cmd_VME_error* function

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

ixa_delay

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_delay (unsigned long nticks);
```

DESCRIPTION:

The function *ixa_delay* can be used to delay operation until the specified number of clock ticks, n*ticks,* has elapsed. This function must not be called with interrupts disabled since this prevents the timer from advancing.

RETURN STATUS

None

ixa_delay_msec, ixa_delay_sec, ixa_delay_usec

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_delay_msec (unsigned long msec);

void ixa_delay_sec (unsigned long sec);

void ixa_delay_usec (unsigned long usec);
```

DESCRIPTION:

The functions *ixa_delay_msec, ixa_delay_sec,* and *ixa_delay_usec,* can be used to delay operation until the specified time period has elapsed. The time period should be specified in *msec*, *sec*, or *usec* respectively. These functions must not be called with interrupts disabled since this will prevent the timer from advancing.

RETURN STATUS:

None

ixa_dma_init

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_dma_init(int nq);
```

DESCRIPTION:

The function *ixa_dma_init* sets up the hardware and software for control of the DMA channels. The DMAs are located within the bridges for each SPE. The nq parameter determines the number of DMA requests that can be queued to the DMA.

RETURN STATUS:

0 = Success, else cannot allocate memory for queues.

ixa_dma_start

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_dma_start(          void *src,
                            void *dest,
                            int nbytes,
                            unsigned long options,
                            void (*done)(long param, long status),
                            long doneparam);
```

DESCRIPTION:

The function *ixa_dma_start* queues a request to a DMA controller. If the DMA is not busy at the time the function is called, then *ixa_dma_start* starts the DMA.

The request will transfer *nbytes* from *src* to *dest* using *options*. *src* or *dest* values less than 0x08000000 (128 megabytes) are interpreted as references to local memory. Addresses beyond this range are treated as PCI references. Any combination of local memory and PCI references may be used – for example, the source and address values may both be local memory addresses, resulting in a memory to memory move.

src and *dest* must be physical addresses, such as those returned by *ixa_mmu_map_addr*.

Options can be used to specify that the source address or destination address (but not both) be held during the transfer. This is useful when reading from or writing to a FIFO. For a held address, *options* can also specify the width of the source or destination. *Options* may include none or one of the following values:

DMA_MODE_SAHE: The source address should be held during the transfer
DMA_MODE_DAHE: The destination address should be held during the transfer

logically OR-ed with none or one of the following:

DMA_MODE_SATS_1: The held source is a byte
DMA_MODE_SATS_2: The held source is a 16-bit
DMA_MODE_SATS_4: The held source is a word
DMA_MODE_SATS_8: The held source is a double word

DMA_MODE_DATS_1: The held destination is a byte
DMA_MODE_DATS_2: The held destination is a 16-bit
DMA_MODE_DATS_4: The held destination is a word

DMA_MODE_DATS_8: The held destination is a double word

An option value of zero is most often used, and specifies that both the *src* and *dest* should be incremented during the transfer.

When the transfer completes, the DMA interrupt service routine will call *done* passing *doneparam* and *status*. The interrupt service function automatically starts the next queued requests when the DMA completes. The *done* function may take any action that is legitimate from within an interrupt service routine, including calling *ixa_dma_start* to start another transaction. *Doneparam* is a user-defined parameter that is remembered by *ixa_dma_start* function when the request is queued. Status values passed to *done* are the logical OR of any of the following bits:

DMA_STATUS_EOCAI:	end of transfer
DMA_STATUS_EOSI :	end of segment
DMA_STATUS_CB:	channel is busy
DMA_STATUS_PE:	A PCI bus error occurred
DMA_STATUS_LME:	A local memory error occurred

Normal DMA completion is indicated when

```
((status & (DMA_STATUS_PE | DMA_STATUS_LME)) == 0)
```

RETURN STATUS:

-1:	No free queue elements
0:	DMA has been started
1:	DMA is busy, but request has been queued, and will be started in its turn

NOTES:

A return status of -1 indicates that requests are being queued faster than they are being serviced. If *ixa_dma_start* returns –1, the request should be retried, except from within an ISR. In order to avoid running out of queue space within an ISR, the DMA must be initialized with the proper number of queue elements. A good rule of thumb is to have a queue entry for each I/O buffer in the system.

Remember that *src* and *dest* are physical addresses, not logical addresses.

When DMAing into memory, the target memory area should be cache inhibited. Alternately, the target memory area can have cache enabled and the application can invalidate cache prior to accessing the memory area.

When DMAing from memory, the source memory area should be cache inhibited. Alternately, the source memory area can have cache enabled and the application can perform a cache flush operation before startgin the output DMA operation.

ixa_evt_disable

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_evt_disable ( void );
```

DESCRIPTION:

The function *ixa_evt_disable* disables all external interrupts as well as the decrementor timer interrupts. This is accomplished by clearing the EE bit of the MSR register. See the PowerPC manual for a description of the MSR.

RETURN STATUS:

The original value of the MSR is returned. The application should save this value and provide it as an input parameter for a subsequent call to *ixa_evt_enable.*

ixa_evt_enable

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_evt_enable ( void );
```

DESCRIPTION:

The function *ixa_evt_enable* enables all external interrupts. This is accomplished by setting the EE bit of the MSR register. See the PowerPC manual for a description of the MSR.

Before an application first runs, external interrupts are disabled by the Common Boot Code. An application should call this function during initialization if external interrupts are to be used.

RETURN STATUS:

The original value of the MSR is returned.

ixa_evt_get

CALLING SEQUENCE:

```
#include <ixa.h>

void *ixa_evt_get (int vector);
```

DESCRIPTION:

The function *ixa_evt_get* returns the pointer to the exception (interrupt) handler which services the interrupt that vectors to location, *vector*. Valid exception vectors are 0x100 thru 0x2F00, in multiples of 0x100, although some vectors may be reserved. For a discussion of exception vectors, please see the PowerPC user manual for the specific processor that is being used.

RETURN STATUS:

If *vector* is invalid, *ixa_evt_get* returns –1. If an ISR has not been defined for the exception vector, *vector*, this function returns 0.

ixa_evt_set

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_evt_set (  int      vector,
                   void(*f) () );
```

DESCRIPTION:

The function *ixa_ evt_set* installs the exception (interrupt) handler, *f*, to exception vector, *vector*. Valid exception vectors are 0x100 thru 0x2F00, in multiples of 0x100, although some vectors may be reserved. For a discussion of exception vectors, please see the PowerPC user manual for the specific processor that is being used.

RETURN STATUS:

If this function is successful, a zero is returned. Otherwise, an invalid interrupt vector was passed and a non-zero value will be returned.

ixa_evt_restore

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_evt_restore ( int oldstate );
```

DESCRIPTION:

The function *ixa_ evt_restore* restores the MSR to the value contained in *oldstate.* An example of its use is provided below:

```
void function(void)
{
      int evt_state;
      evt_state = ixa_evt_disable();

      /* Critical Code running with interrupts disabled goes here
      */

      ixa_evt_restore(evt_state);
}
```

RETURN STATUS:

The new value of the MSR is returned.

ixa_flash_delete, ixa_flash_read, ixa_flash_write

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_flash_delete (char *name);

IXABSP_STATUS ixa_flash_read ( char         *name,
                               void         *local_addr,
                               unsigned int  count);

IXABSP_STATUS ixa_flash_write ( char        *name,
                                void        *local_addr,
                                unsigned int count);
```

DESCRIPTION:

The *ixa_flash_write* and *ixa_flash_read* functions write/read a block of user flash data. When calling *ixa_flash_write*, *name* specifies a unique character string that identifies the block of data. The data can then be read by calling *ixa_flash_read* and passing the same character string. The *ixa_flash_delete* function deletes from flash memory the data associated with *name*.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPEN: flash memory evice was not initialized correctly.
IXABSP_CMD_CHANNEL_BUSY: command channel already in use.
IXABSP_CMD_ERROR: error occurred while processing the command.

ixa_get_cluster_id

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_get_cluster_id ( void );
```

DESCRIPTION:

The function *ixa_get_cluster_id* returns the cluster ID of the calling processor. Processors A and B are located in cluster 0 while processors C and D are located in cluster 1. If this function is called from the IOPlus processor, a cluster ID of –1 is returned.

RETURN STATUS:

Cluster ID

ixa_get_proc_id

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_get_proc_id ( void );
```

DESCRIPTION:

The function *ixa_get_proc_id* returns the ID of the calling processor. The ID's of the processors are defined below as well as the macros which can be used to reference these ID's.:

IOPlus Processor	PROC_ID__IOP	0
Processor A	PROC_ID_A	1
Processor B	PROC_ID_B	2
Processor C	PROC_ID_C	3
Processor D	PROC_ID_D	4

RETURN STATUS:

Processor ID

ixa_get_proc_info

CALLING SEQUENCE:

```
#include <ixa.h>

PROC_INFO *ixa_get_procinfo ( void );
```

DESCRIPTION:

The function *ixa_get_proc_info* returns a pointer to a processor information structure. This structure is defined below:

```
typedef struct
{
        unsigned int    proc_id         /* Processor ID */
        unsigned int    proc_type       /* Processor type */
        unsigned int    proc_rev        /* Processor rev */
        unsigned int    proc_speed      /* Processor speed */
        unsigned int    board_type      /* Board type */
        unsigned int    board_rev       /* Board rev */
        unsigned int    vme_addr        /* Board VME address */

} PROC_INFO
```

RETURN STATUS:

Processor Information

ixa_get_proc_rev

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_get_proc_rev ( void );
```

DESCRIPTION:

The function *ixa_get_proc_rev* reads processor special purpose register 287, to get the revision information for the calling processor.

RETURN STATUS:

Processor revision

ixa_get_proc_type

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_get_proc_type ( );
```

DESCRIPTION:

The function *ixa_get_proc_type* reads processor special purpose register 287, to determine the processor type for the calling processor. Possible values returned and the macros which may be used to define these values are shown below:

PROCTYPE_MPC_8240	0x0081
PROCTYPE_MPC_750	0x0008
PROCTYPE_MPC_7400	0x000C
PROCTYPE_MPC_7410	0x800C

Please note that this list will expand as other processors are supported by the IXA7 board.

RETURN STATUS:

Processor type

ixa_get_sysproc_id

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_get_sysproc_id ( void );
```

DESCRIPTION:

The function *ixa_get_sysproc_id* gets the system processor ID of the calling processor. This ID is unique across multiple IXA7 boards in the same system.

RETURN STATUS:

System processor ID of calling processor

ixa_init

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_init (void);
```

DESCRIPTION:

The function *ixa_init* initializes the IXAbsp environment. *ixa_init* establishes interrupt vectors, sets up the timer, and performs other housekeeping initialization functions.

ixa_init must not be called from an application program that runs under an operating system. If an application runs in stand-alone mode (i.e., with no operating system), then the application must call *ixa_init*.

This must be the first IXAbsp function called in a stand-alone program.

RETURN STATUS:

IXABSP_SUCCESS: init completed successfully.
IXABSP_IOPLUS_ERR: IOP Runtime code not running.

ixa_int_ack

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_int_ack(int intid );
```

DESCRIPTION:

The function *ixa_int_ack* acknowledges the specified interrupt. The action performed depends upon the interrupt type. Acknowledging an inter-processor interrupt clears a register. Acknowledging a VME interrupt takes different actions. Many interrupt ID values require no acknowledge action.

RETURN STATUS:

None

ixa_int_disable

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_int_disable( int int_id );
```

DESCRIPTION:

The function *ixa_int_disable* disables the interrupt identified by *int_id*.

RETURN STATUS:

0 = success, else *int_id* was invalid

ixa_int_enable

CALLING SEQUENCE:

```
#include <ixa.h>

 int ixa_int_enable ( int int_id );
```

DESCRIPTION:

The function *ixa_int_enable* enables the interrupt specified by int_id.

RETURN STATUS:

Status: 0 means success, else *int_id* is invalid.

ixa_int_getvect

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_int_getvect( int int_id, void **fp, void *param );
```

DESCRIPTION:

The function *ixa_get_vector* gets the interrupt service routine entry point and parameter associated with *int_id*. If no interrupt service routine was installed, then *fp will be zero.

RETURN STATUS:

Status: 0 = success, else invalid int_id.

ixa_int_lock

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_int_lock( void );
```

DESCRIPTION:

The function *ixa_int_lock* locks out external interrupts by clearing the external interrupt enable bit in the calling processor's machine state register (MSR). The value returned by this function is that of the MSR before modification and can be passed to the *ixa_int_unlock* function to restore the original interrupt state. The *ixa_int_lock* function provides the same functionality a*s ixa_evt_disable*.

RETURN STATUS:

The original value of the MSR is returned.

ixa_int_setpri

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_int_setpri( unsigned int int_id,
                    unsigned int priority );
```

DESCRIPTION:

The function *ixa_int_setpri* sets the *priority* of an interrupt, *int_id*, that is managed by the SPE-PCI bridge internal interrupt controller. The priority of other interrupts can not be altered. Valid priority values are 0 through 15 with 15 indicating the highest priority. The interrupts whose priorities can be controlled are:

Timer 0 Interrupt
Timer 1 Interrupt
Timer 2 Interrupt
Timer 3 Interrupt
DMA Channel 0 Interrupt
DMA Channel 1 Interrupt

RETURN STATUS:

Status: 0 indicates success, else *int_id* or *priority* were invalid.

ixa_int_setvect

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_int_setvect( int           int_id,
                     void(*f)      (),
                     unsigned long farg );
```

DESCRIPTION:

The function *ixa_ int_setvect* installs the exception (interrupt) handler*, f,* for the interrupt specified by *int_id*. When the interrupt handler is called, the argument, *farg*, is passed to it.

RETURN STATUS:

Status: 0 means success, else *int_id* is invalid.

ixa_int_unlock

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_int_unlock( unsigned oldmask );
```

DESCRIPTION:

The function *ixa_int_unlock* restores the state of the external interrupt enable bit of the MSR from that which is contained in *oldmask*, a value that reflects the contents of the MSR. Therefore, if external interrupts are enabled in *oldmask* (BIT EE*)*, then they will be enabled after calling this function. Otherwise, if external interrupts are disabled in *oldmask*, they will also be disabled after calling this function. An example of its use is shown below:

```
void function(void)
{
      unsigned evt_state;
      evt_state = ixa_int_lock();

      /* Critical Code running with interrupts disabled goes here
      */

      ixa_int_unlock(evt_state);
}
```

RETURN STATUS:

The original value of the MSR is returned as an interesting but marginally useful side effect.

ixa_ipi_disable, ixa_ipi_enable

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_ipi_disable ( int id );
int ixa_ipi_enable  ( int id );
```

DESCRIPTION:

These functions disable or enable specific inter-processor interrupts. The interrupt *id* must be in the range INT_VECTOR_IPIU0 through INT_VECTOR_IPIU3. These vectors are available for user applications. Other id values should not be used by applications.

RETURN STATUS:

0 indicates success; else id is out of range.

NOTES:

Do not confuse the user inter-processor interrupt vectors (denoted INT_VECTOR_IPIUn) with the older inter-processor interrupt vectors (denoted INT_VECTOR_IPIn). The older vectors are reserved for inter-processor shared memory communications and should not be used by applications programs.

Before responding to inter-processor interrupts, an application must first trap the interrupt (by calling *ixa_int_setvect* and a vector number of `INT_VECTOR_IPIUn`), and then enable the interrupt (by calling *ixa_ipi_enable*).

The library provides no way for the processor receiving an interrupt to tell which processor generated the interrupt. Normally, applications convey such information using shared memory along with the inter-processor interrupts.

ixa_ipi_interrupt

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_ipi_interrupt (int target_proc, int id);
```

DESCRIPTION:

The function *ixa_ipi_interrupt* generates interrupt *id* on processor *target_proc*. *Target_proc* must be one of PROC_ID_IOP, PROC_ID_A, PROC_ID_B, PROC_ID_C, or PROC_ID_D. The interrupt *id* must be one of the user inter-processor interrupts. These interrupts are defined as INT_VECTOR_IPIU0 through INT_VECTOR_IPIU3.

A processor can interrupt itself.

RETURN STATUS:

This function returns 0 if all input parameters were within range. Otherwise, this function returns a nonzero value.

User inter-processor interrupts are available with version number 0x10 and later of the interrupt multiplexer FPGA. Prior versions cause this function to return an error status.

ixa_led_blink

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_led_blink ( unsigned  num_blinks,
                     unsigned  blink_time );
```

DESCRIPTION:

The function *ixa_led_blink* blinks the LED of the calling processor *num_blinks* times, for a duration of *blink_time* cycles.

RETURN STATUS:

None

ixa_led_blink2

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_led_blink2 (   unsigned    num_blinks,
                        unsigned    ontime,
                        unsigned    offtime );
```

DESCRIPTION:

The function *ixa_led_blink2* blinks the LED of the calling processor *num_blinks* times. For each blink, the LED will remain on for *ontime* ticks and off for *offtime* ticks.

RETURN STATUS:

None

ixa_led_off

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_led_off ( void );
```

DESCRIPTION:

The function *ixa_led_off* turns off the LED of the calling processor.

RETURN STATUS:

None

ixa_led_on

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_led_on ( void );
```

DESCRIPTION:

The function *ixa_led_on* turns on the LED of the calling processor.

RETURN STATUS:

None

ixa_mmu_get_page_size

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_mmu_get_page_size ( void );
```

DESCRIPTION:

The function *ixa_mmu_get_page_size* returns the size of a page of memory in bytes. For the PowerPC, a page size is 4096 bytes.

RETURN STATUS:

Page size

ixa_mmu_map_addr

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_map_addr ( void  *ea,
                       void **paddr );
```

DESCRIPTION:

The function *ixa_mmu_map_addr* translates the logical address, *ea*, to a physical address, *paddr*. The resulting physical address is stored at ***paddr*.

This function translates the address as the MMU would, even if the MMU is not enabled. Data BAT registers are checked first. If they do not reflect a mapping for the logical address, then this function consults the page table.

This function assumes that the address points to data (i.e. not the address of an instruction) which means that the IBAT registers are never checked.

RETURN STATUS:

A zero is returned if the function is successful. Otherwise, a non-zero value is returned.

ixa_mmu_map_block

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_map_block ( void          *ea,
                        void          *phys,
                        unsigned       attr,
                        unsigned long nbytes);
```

DESCRIPTION:

The function *ixa_mmu_map_block* updates the data block address translation entries given a logical address, *ea*, the corresponding physical address, *phys*, and the page attributes *attr*.

Please note the restriction for *nbytes*: 128 Kbytes < *nbytes* < 256 Mbytes. Both the logical address, *ea*, and the physical address, *phys*, must be multiples of *nbytes*.

RETURN STATUS:

If the new entry conflicts with an existing entry, a non-zero value will be returned. If all data BAT registers are active, a nonzero error status is returned.

Otherwise, a zero will be returned to indicate success.

SEE ALSO:

ixa_mmu_set_page_attr for a list of valid *attr* values.

ixa_mmu_map_page, ixa_mmu_map_pages

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_map_page (  void        *ea,
                        void        *phys,
                        unsigned     attr );

int ixa_mmu_map_pages ( void        *ea,
                        void        *phys,
                        unsigned     attr,
                        int          npages );
```

DESCRIPTION:

The function *ixa_mmu_map_page* creates a new page table entry given a logical address, *ea*, the corresponding physical address, *phys*, and the page attribute *attr*. Likewise, *ixa_mmu_map_pages*, will create *npages* page table entries.

Please note that the logical address, *ea*, and physical address, *phys*, must be multiples of the page size.

RETURN STATUS:

If the page table is too small or the logical address is already mapped to a physical address, a non-zero value will be returned. Otherwise, a zero will be returned to indicate success.

SEE ALSO:

ixa_mmu_set_page_attr for a list of valid *attr* values.

ixa_mmu_peek_l, ixa_mmu_poke_l

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_mmu_peek_l ( void  *addr);

void ixa_mmu_poke_l ( void     *addr,
                      unsigned  value );
```

DESCRIPTION:

The function *ixa_mmu_peek_l* reads a logical address, *addr*, using the MMU while the function *ixa_mmu_poke_l* writes *value* to a logical address, *addr*, using the MMU.

RETURN STATUS:

None

ixa_mmu_peek_p, ixa_mmu_poke_p

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_mmu_peek_p ( void *addr );

void ixa_mmu_poke_p ( void     *addr,
                      unsigned  value )
```

DESCRIPTION:

The function *ixa_mmu_peek_p* reads a physical address, *addr*, bypassing the MMU while function *ixa_mmu_poke_p* writes *value* to a physical address, *addr*, bypassing the MMU.

Use these functions with caution since they temporarily disable interrupts and the data MMU. Once the physical location is read, the interrupt and MMU states are restored.

RETURN STATUS:

ixa_mmu_peek returns the value read from physical memory.

ixa_mmu_remap_block

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_remap_block ( void             *ea,
                          void             *phys,
                          unsigned          attr,
                          unsigned long     nbytes);
```

DESCRIPTION:

The function *ixa_mmu_remap_block* updates or replaces the data block address translation entries given a logical address, *ea*, the corresponding physical address, *phys*, and the page attributes *attr*.

Please note the restriction for *nbytes*: 128 Kbytes < *nbytes* < 256 Mbytes. Both the logical address, *ea*, and the physical address, *phys*, must be multiples of *nbytes*.

RETURN STATUS:

If the new entry conflicts with an existing entry, a non-zero value will be returned. Otherwise, a zero will be returned to indicate success.

SEE ALSO:

ixa_mmu_set_page_attr for a list of valid *attr* values.

ixa_mmu_remap_page, ixa_mmu_remap_pages

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_remap_page ( void       *log,
                         void       *phys,
                         unsigned    attr );

int ixa_mmu_remap_pages ( void      *log,
                          void      *phys,
                          unsigned   attr,
                          int        npages );
```

DESCRIPTION:

The function *ixa_mmu_remap_page* creates or replaces a page table entry given a logical address, *log*, and the corresponding physical address, *phys* along with the page attributes, *attr*. The function *ixa_mmu_remap_pages* creates or replaces *npages* page table entries. Unlike *ixa_mmu_map_pages*, these functions first remove any conflicting entries from the page table.

RETURN STATUS:

If the page table is full, a non-zero value will be returned. Otherwise, a zero will be returned to indicate success.

SEE ALSO:

ixa_mmu_set_page_attr for a list of valid *attr* values.

ixa_mmu_set_block_attr

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_set_block_attr ( void    *log,
                             unsigned attr );
```

DESCRIPTION:

The function *ixa_mmu_set_block_attr* sets the attributes for the block of memory that contains the logical address*, log.*

RETURN STATUS:

0 indicates successful completion

SEE ALSO:

ixa_mmu_set_page_attr for a list of valid *attr* values.

ixa_mmu_set_page_attr

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_set_page_attr ( void *log, unsigned mask, unsigned attr );
```

DESCRIPTION:

The function *ixa_mmu_set_page_attr* sets the attributes for the page of memory that contains the logical address*, log*. Only attribute bits within mask that are set to one will be altered. The new attribute value is computed according to the following formula:

```
new_attr = (old _attr & ~mask) | (attr & mask)
```

After altering the attribute bits within the page table, the function alters the state of the memory management unit hardware so that the new attributes become effective immediately.

Calling the function with a mask value of zero leaves the page attributes unchanged. Since the function returns the modified attribute value, calling the function with a mask of zero leaves the page attributes unchanged and returns their current state.

The mask and attribute values must be a logical OR of one or more of the following bits:

```
ATTR_R      0x0100      /* this page has been referenced      */
ATTR_C      0x0080      /* this page has been changed         */
ATTR_W      0x0040      /* write-through caching mode         */
ATTR_I      0x0020      /* cache inhibited                    */
ATTR_M      0x0010      /* memory coherency required          */
ATTR_G      0x0008      /* guarded access                     */
ATTR_PP     0x0003      /* page protection: see PPC manual    */
```

Not all combinations are valid. Further information on the page attribute bits can be found in *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors,* chapters 5 and 7.

RETURN STATUS:

This function returns the new state of attribute bits for the given page. If the page is not mapped, then this function returns –1, which is never a valid combination of attribute bits.

ixa_mmu_unmap_block

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_unmap_block ( void          *ea,
                          unsigned long  nbytes );
```

DESCRIPTION:

The function *ixa_mmu_unmap_block* disables the data block address translation register given a logical address, *ea*, and the number of bytes, *nbytes*.

Please note the restriction for *nbytes*: 128 Kbytes < *nbytes* < 256 Mbytes. The logical address must be a multiple of *nbytes*.

RETURN STATUS:

If an error occurs, a non-zero value will be returned. Otherwise, a zero will be returned to indicate success.

ixa_mmu_unmap_page, ixa_mmu_unmap_pages

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_mmu_unmap_page ( void *log );

int ixa_mmu_unmap_pages( void *log,
                         int   npages );
```

DESCRIPTION:

Given a logical address, *log*, function *ixa_mmu_unmap_page* invalidates the page table entry for that address. Function *ixa_mmu_unmap_pages* invalidates *npages* page entries.

RETURN STATUS:

If one of more of the pages specified are not mapped, no pages will be unmapped and a non-zero value will be returned. Otherwise, a zero will be returned to indicate success.

ixa_PCI_config_read, ixa_PCI_config_write

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_PCI_config_read ( unsigned int bus,
                                    unsigned int device,
                                    unsigned int function,
                                    unsigned int address,
                                    unsigned int byte_size,
                                    void        *data);

IXABSP_STATUS ixa_PCI_config_write( unsigned int bus,
                                    unsigned int device,
                                    unsigned int function,
                                    unsigned int address,
                                    unsigned int byte_size
                                    void        *data);
```

DESCRIPTION:

The *ixa_PCI_config_read* and *ixa_PCI_config_write* functions command the IOPlus to perform configuration cycles on the PCI bus. The results are returned to the SPE. The *bus*, *device*, *function*, and *address* specify the target of the configuration cycle. The *byte_size* parameter can be set to 1, 2, or 4 in order to perform byte, word, or long transfers. **data* points to the value to be written (or read).

RETURN STATUS:

IXABSP_SUCCESS: completed successfully.
IXABSP_ERROR: did not complete successfully; invalid address or width.

ixa_PCI_io_read, ixa_PCI_io_write

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_PCI_io_read (void *addr,
                      int   width,
                      void *data);

void ixa_PCI_io_write(void *addr,
                      int   width,
                      void  data);
```

DESCRIPTION:

These functions perform I/O space accesses on the PCI bus. *Addr* is the I/O address. Addresses between 0 and 32767 select the left PMC module; addresses between 32768 and 65535 select the right PMC module. All other addresses are invalid.

Width specifies the width of the transfer: 1 for byte transfers, 2 for 16-bit integer transfers, and 4 for 32-bit transfers. The read function places data read from the PCI I/O address at **data;* the write function writes data from **data* to the I/O address.

RETURN STATUS:

IXABSP_SUCCESS: completed successfully.
IXABSP_ERROR: did not complete successfully; invalid address or width.

ixa_pci_to_local, ixa_local_to_pci

CALLING SEQUENCE:

```
#include <ixa.h>

void *ixa_pci_to_local (void *pci_adrs);
void *ixa_local_to_pci (void *local_adrs);
```

DESCRIPTION:

The function *ixa_pci_to_local* converts a PCI address to a local address while *ixa_local_to_pci* converts a local address to a PCI address. Please note that zero always converts to zero.

From an SPEs point of view, global memory has a starting "local" address of 0xb800_0000. The bridge mapping for SPEs places the SPE local memory at zero. Global memory therefore must be mapped to a different region. This SPE bridge devices map references from SPEs in this range to PCI addresses starting at zero, the true global memory PCI starting address.

Ixa_local_to_pci should NOT be used to translate local addresses to addresses used by other (non-processor) devices, such as the Universe II or PMC devices. Instead, use the function *ixa_local_to_pci_dev* for these translations.

RETURN STATUS:

ixa_pci_to_local returns a pointer to a local address while *ixa_local_to_pci* returns a pointer to a PCI address as would be used by an SPE or the IOP.

ixa_local_to_pci_dev

CALLING SEQUENCE:

```
#include <ixa.h>

void *ixa_local_to_pci_dev (void *local_adrs);
```

DESCRIPTION:

The function *ixa_local_to_pci_dev* converts a local address to a "pure" PCI address. The output address represents the address as would be viewed by PCI devices such as PMCs or the Universe II VME controller.

Addresses in the range 0xb8nn_nnnn are converted back to zero, the true base address of global memory. Addresses in the range 0 through the local processor memory size are adjusted by adding the PCI starting address of the calling processor's memory.

RETURN STATUS:

ixa_local_to_pci_dev returns a pointer to a PCI address as would be used by an I/O device.

ixa_PCI_read, ixa_PCI_write

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_PCI_read (  void         *pci_addr,
                              void         *local_addr,
                              unsigned int count,
                              unsigned int byte_size,
                              BOOLEAN      swap);

IXABSP_STATUS ixa_PCI_write(  void         *pci_addr,
                              void         *local_addr,
                              unsigned int count,
                              unsigned int byte_size,
                              BOOLEAN      swap);
```

DESCRIPTION:

The function *ixa_PCI_read* transfers data from the PCI bus to local memory while the function *ixa_PCI_write* transfers data from memory local to the calling processor to the PCI bus. The PCI address in the transfer is specified by *pci_addr* and the local memory address is specified by *local_addr*. The number of data elements to transfer is specified by *count* and the byte size of a data element is specified by *byte_size*. Accepted byte sizes are 1, 2, or 4 bytes. The *swap* parameter specifies whether 4-byte PCI transfers are byte-swapped (SWAP), or not byte-swapped (NO_SWAP). The PCI transfer is performed by mapping the PCI window and performing CPU core read/writes to the PCI window. Note that *local_addr* may be an internal or external memory address.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

ixa_pm_init

CALLING SEQUENCE:

```
#include <ixa.h>

void * ixa_pm_init( unsigned long number_of_events,
                    unsigned long trace_buffer_entries,
                    unsigned long mode );
```

DESCRIPTION:

ixa_pm_init allocates and initializes data structures required for performance monitoring. Events are points in the source code where the execution performance data is sampled and recorded. The function *ixa_pm_start* denotes the start of a segment of code to be instrumented, while *ixa_pm_stop* indicates the end of the code segment. Execution data are collected by bracketing a segment of code with these functions and a unique *event_id.*

For each event ID, the performance monitoring functions collect data from the processor's internal performance monitoring facility, placing the data in an event table and a trace buffer. The event table tracks performance data by event ID, computing averages and maximums. The trace buffer collects a circular log of execution trace data.

Event Ids are numbered zero through *number_of_events* – 1. When an event is started (by calling *ixa_pm_start*), the performance monitoring functions record certain performance information in the event table, and place an entry in the trace buffer. When the corresponding event is stopped (by calling *ixa_pm_stop*), the event table entry is updated to reflect elapsed time and counter values, maximums, and averages. Stop events are also recorded in the trace buffer. The trace buffer contains *trace_buffer_entries* entries.

The *mode* parameter determines the performance data collected. The PowerPC allows up to four processor-internal measurements to be tracked concurrently. The following macros define specific modes:

MODE_1: the PM registers are configured to count processor cycles, instructions completed, L1 cache data hits and L1 cache data misses.

MODE_2 the PM registers are configured to count processor cycles, instructions completed, L2 cache data hits and L2 cache data misses.

MODE_3 the PM registers are configured to count processor cycles, instructions completed, DTLB misses and DTLB walks.

MODE_X is a generic macro allowing the caller to select any items the processor can count:

MODE_X(pm1, pm2, pm3, pm4)

Where pm1, pm2, pm3, and pm4 are values meaningful to the PowerPC.

When using MODE_X, it is the caller's responsibility to make sure that the individual values are valid for the target processor. More detailed information on PM register initialization can be found in chapter 11 of MPC7400 or MPC7410 RISC Microprocessor User's Manuals.

RETURN STATUS:

This function returns a void pointer. This value becomes a handle for subsequent function calls. A NULL pointer is returned if memory cannot be allocated.

NOTES:

Calling ixa_pm_init more than once in an application overwrites the previous modes, tables, and data.

SEE ALSO:

ixa_pm_start, ixa_pm_stop, ixa_pm_display_stats, ixa_pm_display_trace, ixa_pm_reset, ixa_pm_term

ixa_pm_reset

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_pm_reset ( void* handle );
```

DESCRIPTION:

ixa_pm_reset clears all statistics collected by the performance monitoring functions. *handle* is the value returned by a call to *ixa_pm_init.*

RETURN STATUS:

None.

ixa_pm_term

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_pm_term ( void* handle );
```

DESCRIPTION:

ixa_pm_term terminates performance monitoring. This function frees all memory allocated by *ixa_pm_init* and freezes all counters. *handle* is the value returned by a call to *ixa_pm_init.*

RETURN STATUS:

None.

ixa_pm_display_stats

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_pm_display_stats( void* handle );
```

DESCRIPTION:

ixa_pm_display_stats displays statistical performance data collected since calling *ixa_pm_init.* For each event ID, this function displays the number of times the event was started and stopped, the average and maximum times for the event, and the maximum and average processor performance statistics (cache hits, etc.) for each event as determined by the *mode* parameter passed to *ixa_pm_init. handle* is the value returned by *ixa_pm_init.*

RETURN STATUS:

None.

ixa_pm_display_trace

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_pm_display_trace( void* handle,
                           unsigned long count );
```

DESCRIPTION:

ixa_pm_display_trace displays the *count* last entries in the trace buffer. If count is zero or if it exceeds the number of entries in the trace buffer, the entire trace buffer is displayed. Trace buffer entries are always displayed in chronological order. *handle* is a void pointer returned by a call to *ixa_pm_init.*

RETURN STATUS:

None.

ixa_pm_start

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_pm_start( void* handle,
                   unsigned long eventID );
```

DESCRIPTION:

ixa_pm_start starts an event, enters it into the table and logs it into the trace buffer. The PM registers begin to count. *EventID* must be between 0 and *number_of_events* – 1 as provided to the *ixa_pm_init* function. Handle is the value returned by *ixa_pm_init.*

RETURN STATUS:

None.

NOTES:

An attempt to start an event which is already started, results in an entry in the trace buffer only.

ixa_pm_stop

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_pm_stop( void* handle,
                  unsigned long eventID );
```

DESCRIPTION:

ixa_pm_stop marks the end of an event started by *ixa_pm_start.* This function computes the elapsed time and performance monitor count values, and updates the internal tables. This function also logs the stop event into the circular trace buffer.

RETURN STATUS:

None.

NOTES:

An attempt to stop an event, which is already stopped or to stop an event that has not been started results in the entry in the trace buffer only.

ixa_proc_is_iop

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_proc_is_iop ( void );
```

DESCRIPTION:

The macro *ixa_proc_is_iop* can be used to determine if the calling processor is the IOPlus processor. A one is returned if the processor is the IOPlus. Otherwise, zero is returned.

RETURN STATUS:

A nonzero value indicates the function was executed on the IOP.

ixa_proc_is_750

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_proc_is_750 ( void );
```

DESCRIPTION:

The macro *ixa_proc_is_750* can be used to determine if the calling processor is a 750 processor. A one is returned if the processor is a 750. Otherwise, zero is returned.

RETURN STATUS:

A nonzero value indicates the function was called on a PPC750.

ixa_proc_is_7400

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_proc_is_7400 ( void );
```

DESCRIPTION:

The macro *ixa_proc_is_7400* can be used to determine if the calling processor is a 7400 processor. A one is returned if the processor is a 7400. Otherwise, zero is returned.

RETURN STATUS:

A nonzero value indicates the function was executed on a PPC 7400.

ixa_proc_is_7410

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_proc_is_7410 ( void );
```

DESCRIPTION:

The macro *ixa_proc_is_7410* can be used to determine if the calling processor is a 7410 processor. A one is returned if the processor is a 7410. Otherwise, zero is returned.

RETURN STATUS:

A nonzero value indicates the function was executed on a PPC 7410.

ixa_sem_release

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_sem_release ( unsigned int sem );
```

DESCRIPTION:

The function *ixa_sem_release* releases exclusive access to semaphore *sem*. *Sem* is an integer between 0 and 7 (inclusive), specifying a semaphore that was acquired by *ixa_sem_request.*

RETURN STATUS:

None

NOTES:

Releasing a semaphore that was not first acquired through ixa_sem_request can cause serious confusion in the software.

ixa_sem_request

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_sem_request ( unsigned int sem );
```

DESCRIPTION:

The function *ixa_sem_request* requests exclusive access to semaphore *sem*. *Sem* is an integer between 0 and 7 (inclusive), specifying the semaphore requested. If the semaphore is available at the time of the request, it is marked busy, and a status value of 0 is returned. Otherwise, a nonzero status is returned.

RETURN STATUS:

A zero is returned if the semaphore was successfully acquired. A nonzero value indicates a busy semaphore, or an invalid semaphore number.

NOTES:

Interrupt service is temporarily disabled during the semaphore request operation.

Because semaphores are few in number, they should be used to control access to resources versus allocation of resources. For example, a semaphore can be used to serialize access to a table of buffers, where the table contains allocation status for each buffer. If the semaphore is busy, then a processor is accessing or updating the table. Once the table is accessed or updated, the semaphore should be released. Alternately, an approach that uses a semaphore to indicate that a buffer is free requires one semaphore per buffer, and the semaphores would be busy as long as the buffers are allocated, using too many resources.

Semaphores 0 and 1 are reserved for use by IXAtools, and should not be used by applications.

A good approach for maximizing semaphore availability is to disable interrupts and avoid I/O while a semaphore is owned. The following example illustrates using a semaphore in conjunction with disabling interrupts to access a shared resource. Note that the loop re-enables interrupts while retrying to acquire the semaphore.

```
/*
 * Access a shared resource using semaphores
 * With interrupt disabled.
 */

#define SEM    2          /* semaphore to use      */

unsigned long istate;     /* int enable state      */
unsigned int status;      /* sem req status        */


for (;;)                  /* do forever            */
{
     istate = ixa_int_disable ();  /* int disable */
     status = ixa_sem_request (SEM); /* get sem   */
     if (status == 0)              /* got it?     */
          break;                   /* y: break    */

     /*
      * Request failed, so we need to retry.
      * Briefly enable interrupts during retry.
      */

     ixa_int_enable (istate);
}


/*
 * At this point we have the semaphore, and interrupts
 * are disabled.  We can access the data structure,
 * then free the semaphore, and restore the interrupt
 * enable state.
 */

access_shared_resource ();

ixa_sem_release (SEM);         /* free up sem */
ixa_int_enable (istate);       /* restore ints */
```

ixa_tas_cluster

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_tas_cluster ( unsigned short *addr );
```

DESCRIPTION:

The function *ixa_tas_cluster* performs a test-and-set (TAS) operation on the 16 bit value at address, *addr*. Interrupts are temporarily disabled during this operation. This operation is faster than other forms of TAS since it does not involve the PCI bus. However, because the PCI bus is not used, this operation is limited to TAS arbitration functions between processors within a cluster.

The target address must be located within a cache inhibited space.

RETURN STATUS:

A zero is returned if the value was already set. Otherwise, a non-zero value is returned.

ixa_tas_local

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_tas_local ( unsigned char *addr );
```

DESCRIPTION:

The function *ixa_tas_local* performs a test-and-set (TAS) operation on the byte value at address, *addr*. The byte can be located in local memory or any PCI addressable location. Interrupts are temporarily disabled during this operation.

This function uses a Universe II semaphore to serialize access to the TAS location. All other processors and functions arbitrating for an on-board TAS location must use this same function.

The target address must be within a cache inhibited space on all processors.

RETURN STATUS:

A zero is returned if the value was already set or if the Universe II semaphore is busy. Otherwise, a non-zero value is returned.

ixa_tas_vme, ixa_tas_clear_vme

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_tas_vme ( unsigned char *addr );
void ixa_tas_clear_vme ( unsigned char *addr );
```

DESCRIPTION:

The function *ixa_tas_vme* performs a test-and-set (TAS) operation on the byte value at address, *addr*. The byte must be located in a memory location accessible from the VME bus. Interrupts are temporarily disabled during this operation.

This function uses VME bus locking to serialize access to the TAS location. All other processors and functions arbitrating for a VME TAS location must use this same function.

The function *ixa_tas_clear_vme* clears the byte value at address, *addr*. The byte must be located in a memory location accessible from the VME bus. Interrupts are temporarily disabled during this operation.

For both functions, the target address must be within a cache inhibited space on all processors.

RETURN STATUS:

For function *ixa_tas_vme,* a zero is returned if the value was already set or if the VME bus could not be locked. Otherwise, a non-zero value is returned.

ixa_temp_read

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_temp_read ( void );
```

DESCRIPTION:

The function *ixa_temp_read* reads the junction temperature of the microprocessor. This function uses the processor's on-chip Thermal Assist Unit (TAU). The TAU measures temperatures in the range 0°C to 127°C. The error/tolerance of the on-chip TAU without calibration is (+/-2)°C. Temperatures below 0°C are reported as zero. Because behavior of the TAU is unknown for temperatures over 127°C, the returned value of *ixa_temp_read* for temperatures over 127°C is undefined.

RETURN STATUS:

Junction temperature, in degrees Celsius, in the range 0°C to 127°C.

Returns –1 if the processor does not have a TAU.

ixa_timer_cancel

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_timer_cancel ( unsigned long timer_id);
```

DESCRIPTION:

The function *ixa_timer_cancel* cancels the active timer, *timer_id*.

RETURN STATUS:

If the timer was never created, or if it has already expired or was previously cancelled, this function returns a nonzero value. Otherwise, a zero is returned.

ixa_timer_create

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timer_create ( int           when,
                                 void (*what)  (),
                                 unsigned long param );
```

DESCRIPTION:

The function *ixa_timer_create* creates a timer element with an expiration time of *when*. At the time of expiration, the function, *what*, with parameter *param*, will be executed. The newly created timer is assigned a unique ID that can be used by the timer cancel function.

The timer element is added to a linked list of active timer elements. The list is maintained in order by expiration time. When the timer expires and the function is called, the function will execute within the context of the timer interrupt service routine.

RETURN STATUS:

If a timer element can be allocated, the ID of the timer created will be returned. Otherwise, if a timer element can not be allocated, a zero will be returned.

ixa_timer_get_ticks_per_second

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timer_get_ticks_per_second( void );
```

DESCRIPTION:

The function *ixa_timer_get_ticks_per_second* returns the number of ticks per second for the calling processor.

RETURN STATUS:

Returns ticks per second.

ixa_timer_get_TBL

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timer_get_TBL( void );
```

DESCRIPTION:

The function *ixa_timer_get_TBL* returns the current value of the Time Base Register lower 32 bits (TBL). TBL is incremented at the rate returned by ixa_timer_get_timebase, 25,000,000 Hz on CHAMP-AV products. This corresponds to a frequency of 1/4th the processor's bus clock frequency.

RETURN STATUS:

Returns the value stored currently in TBL

NOTES:

This is a low-latency function facilitating precision time measurements. Call this function before entering the code to be measured, and again after the code, and compute the difference.

ixa_timer_get_time

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timer_get_time( void );
```

DESCRIPTION:

The function *ixa_timer_get_time* returns the elapsed time in processor ticks since the timer was started.

RETURN STATUS:

Returns the time in processor ticks.

ixa_timer_get_usec

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timer_get_usec( void );
```

DESCRIPTION:

The function *ixa_timer_get_usec* returns the elapsed time in microseconds since the timer was started. The return value is truncated to 32 bits.

Because the return value is truncated, it is not useful as an absolute measure of time. Instead, this function is intended to support precise measurements of elapsed time. By calling this function before and after executing the code to be timed, and computing the difference, an accurate execution time is computed.

RETURN STATUS:

Returns the time in processor microseconds.

ixa_timer_get_timebase

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timer_get_timebase( void );
```

DESCRIPTION:

The function *ixa_timer_get_timebase* returns the frequency, in Hz, of the clock used for the timer.

RETURN STATUS:

Returns the frequency, in Hz, of the timer.

ixa_timer_init

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_timer_init( int ticks, int ntimers );
```

DESCRIPTION:

The function *ixa_timer_init* sets up the decrementor timer to interrupt every *ticks* clock ticks and creates *ntimers* timer data structures for the operation of the software timer. The data structures are placed in the free list. If timers were previously created, these timers will be de-allocated before creating the new ones.

This function is automatically called by *ixa_init*. However, the application may call this function during initialization in order to increase the number of timers or to adjust the number of ticks per second.

RETURN STATUS:

None

ixa_timer_msec_to_ticks, ixa*_timer_usec_to_ticks,* ixa*_timer_sec_to_ticks*

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timer_msec_to_ticks( unsigned long msec );
unsigned long ixa_timer_usec_to_ticks( unsigned long usec );
unsigned long ixa_timer_sec_to_ticks ( unsigned long sec  );
```

DESCRIPTION:

The functions *ixa_timer_msec_to_ticks, ixa_timer_usec_to_ticks*, and *ixa_timer_sec_to_ticks*, can be used to convert the number of *msec*, *usec*, or *sec*, respectively, to processor ticks and return this value.

RETURN STATUS:

Processor ticks

ixa_timer_set

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_timer_set( int ticks );
```

DESCRIPTION:

The function *ixa_timer_set* sets the decrementor timer expiration frequency to the specified number of *ticks* per second.

RETURN STATUS:

None

ixa_timern_enable, ixa_timern_disable

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_timern_enable (int n);
void ixa_timern_disable (int n);
```

DESCRIPTION:

The functions *ixa_timern_enable* and *ixa_timern_disable* enable control the interrupt enable state for the specified timer. Before enabling interrupts, an interrupt service routine must be specified (by calling *ixa_timern_trap*). The timer may be started, stopped, or set before or after interrupts are enabled. The recommended sequence for intialization is to set the interrupt service routine (*ixa_timern_trap*), set the timer (*ixa_timern_set*), and then enable interrupts (*ixa_timern_enable*). Note that while any processor may configure the timers, the A and C processors are the only processors that can service timer interrupts.

ixa_timern_get_ticks_per_second, ixa_timern_get_timebase

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timern_get_ticks_per_second (int n);
unsigned long ixa_timern_get_timbase (int n);
```

DESCRIPTION:

The function *ixa_timern_get_ticks_per_second* returns the timer period, in ticks per second, where *n* is an integer between TIMER_N_MIN and TIMER_N_MAX, inclusive. If the time was never initialized (by *ixa_timern_set*), then this function returns 0.

The function *ixa_timern_get_timebase* returns the time base, in Hz, where *n* is an integer between TIMER_N_MIN and TIMER_N_MAX, inclusive. The timebase for all n timers is the same, and is one-fourth the bus clock frequency, 25,000,000 for most implementations.

The *timern* timers reside within the MPC107 bridge device, and should not be confused with the decrementor timer resident within each processor. There are four timers per bridge device. Only the A and C processors can service interrupts generated by the timers.

ixa_timern_msec_to_ticks, ixa_timern_sec_to_ticks, ixa_timern_usec_to_ticks

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timern_msec_to_ticks (unsigned long ms, int n);
unsigned long ixa_timern_sec_to_ticks (unsigned long s, int n);
unsigned long ixa_timern_usec_to_ticks (unsigned long us, int n);
```

DESCRIPTION:

The function *ixa_ timern_msec_to_ticks* converts the input parameter *ms* into timer ticks, using the time base and ticks per second of timer *n*. The function variants *sec* and *usec* convert seconds and microseconds to clock ticks. All functions return the smallest number of clock ticks which would be >= the requested time, so 0 usec = 0 ticks, but 1 usec = 1 tick, no matter how large a tick may be.

The *timern* timers reside within the MPC107 bridge device, and should not be confused with the decrementor timer resident within each processor. There are four timers per bridge device. Only the A and C processors can service interrupts generated by the timers.

ixa_timern_read

CALLING SEQUENCE:

```
#include <ixa.h>

unsigned long ixa_timern_read (int n);
```

DESCRIPTION:

The function *ixa_timern_read* reads the current count for the timer, in bus clock cycles. The count counts down from (time base)/(ticks per second) to zero, at the rate of (time base) Hertz. The high bit, which corresponds to the count enable bit within the EPIC, is always set to zero in the result.

If n is invalid, this function returns zero.

The *timern* timers reside within the MPC107 bridge device, and should not be confused with the decrementor timer resident within each processor. There are four timers per bridge device. Only the A and C processors can service interrupts generated by the timers.

ixa_timern_set

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_timern_set (int ticks, int n);
```

DESCRIPTION:

The function *ixa_timern_set* sets timern to count down at a rate of *ticks* times per second. The count is enabled when the time is set. The interrupt enable state of the timer remains unchanged. This function converts *ticks* into an appropriate countdown value by dividing the time base value by ticks.

The *timern* timers reside within the MPC107 bridge device, and should not be confused with the decrementor timer resident within each processor. There are four timers per bridge device. Only the A and C processors can service interrupts generated by the timers.

ixa_timern_start, ixa_timern_stop

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_timern_start (int n);
void ixa_timern_stop (int n);
```

DESCRIPTION:

The function i*xa_timern_start* enables counter n to begin counting. Note that a processor is automatically started by the set function. This function is provided to restart timing after it has been suspended by ixa_timern_stop.

ixa_timern_stop suspends counting for timer *n*.

ixa_timern_trap

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_timern_trap (int n, void (*f)(), int param);
```

DESCRIPTION:

The function *ixa_timern_trap* installs function *f* as the interrupt handler for timer *n*. When timer *n* interrupts are enabled and timer *n* counts down to zero, function *f* will be called passing parameter *p*. The hardware automatically reloads the countdown register after the counter reaches zero.

The *timern* timers reside within the MPC107 bridge device, and should not be confused with the decrementor timer resident within each processor. There are four timers per bridge device. Only the A and C processors can service interrupts generated by the timers. Calling this function on the B and D processors is meaningless.

ixa_VME_dma_in, ixa_VME_dma_out

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

int ixa_VME_dma_in (  void *src,
                      void *dest,
                      int nbytes,
                      unsigned long attr,
                      void (*done) (long p, long status),
                      long p);

int ixa_VME_dma_out(  void *src,
                      void *dest,
                      int nbytes,
                      unsigned long attr,
                      void (*done) (long p, long status),
                      long p);
```

DESCRIPTION:

These functions use the VME controller's DMA to move *nbytes* of data from *src* to *dest.* If the DMA is free at the time of the call, the function starts the DMA. If the DMA is busy at the time of the call, the function queues the DMA request.

For input transfers, *src* is a VME address, and *dest* is an on-board "pure" PCI address, as returned from *ixa_local_to_pci_dev.* For output transfers, *src* is a "pure" PCI address, and *dest* is a VME address.

attr selects the data size, address size, priviledge mode, program or data selector, and block transfer option. (Block transfers are generally more efficient than non-block transfers, but note that not all boards support block transfers.) *attr* is constructed from the logical-OR of the following options:

Address Width and Data Width (choose one):

VME_A32D64
VME_A32D32
VME_A32D16
VME_A32D8
VME_A24D64
VME_A24D32
VME_A24D16
VME_A24D8
VME_A16D64

VME_A16D32
VME_A16D16
VME_A16D8

Priviledge mode selector (choose one):
VME_PRIV
VME_NONPRIV

Program/Data Space Selector (choose one):
VME_PROGRAM
VME_DATA

Block Transfer Selection:
VME_BLT

If no privilege mode selector is specified, VME_NONPRIV is assumed. If no program/data space selector is specified, VME_DATA is assumed. If VME_BLT is not specified, the transfer uses single-cycle bus transactions.

VME DMA transfer requests are asynchronous. That is, these functions start the DMA operation, and the operation completes on is own at a later time. When a DMA interrupt occurs, the interrupt service routine calls the *"done"* function specified with the DMA request, and the interrupt service routine starts the next queued transaction. The user-specified *done* function may perform any action permissible within an interrupt service routine. The parameter *p* specified with the DMA request is also passed to the *done* function. This parameter can have any significance the application desires. The status value passed to the done function is zero for successful completion, or a logical OR combination of one or more of the following error codes:

VME_DMA_SUCCESS
VME_ERROR_PARAMS invalid parameters
VME_ERROR_BUSERR bus error/address err
VME_ERROR_PCIERR PCI bus error
VME_ERROR_HALT dma halted
VME_ERROR_STOPdma stopped

A *done* function pointer of zero is valid, and specifies no done function. In such a case, the DMA interrupt service routine simply starts the next queued DMA transaction.

Note that this function is meaningful only on IXA VME boards.

RETURN STATUS:

0 indicates the request was queued or the DMA was started. Nonzero values indicate a parameter error or no free queue elements.

NOTES:

ixa_VME_dma_in and *ixa_VME_dma_out* are functionally the same as the VxWorks board support library function *sysBusDmaIn* and *sysBusDmaOut*.

The VxWorks versions of these functions support a special *done* value of *SYSBUS_DMA_SYNC*, which suspends the calling task until the DMA operation completes, returning the DMA status to the caller. This option is not supported by the *ixa_VME* versions of these functions.

The DMA request queue can contain up to 64 pending requests.

Because the DMA interrupt is not sharable, these DMA functions are not sharable. One processor on the IXA board should assume responsibility for managing the VME DMA. Any processor, including the IOP, may do this. No special setup is required to do this; simply have one and only one processor call the *ixa_VME_dma* functions.

ixa_VME_open

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

int ixa_VME_open (      void *busaddr,
                        unsigned long length,
                        unsigned long attr,
                        int shared  );
```

DESCRIPTION:

The function *ixa_VME_open* opens a window to VME bus address space, returning a handle to be used in subsequent *ixa_VME* functions. A window is a mapping from the processor's local address space to VME space. Accessing locations within the window causes accesses on VME bus.

The window start address is determined by *busaddr*. The underlying VME device driver may round *busaddr* down to the next natural boundary imposed by the VME device. For the Universe II, this is 64k.

length determines the window size. If *length* is zero, the device driver chooses the maximum window size, which is 8 megabytes. If the requested *length* exceeds this maximum, the open fails and *ixa_VME_open* returns a null handle.

attr determines the window attributes, and is constructed from the logical-OR of the following options:

Address Width and Data Width (choose one):

VME_A32D64
VME_A32D32
VME_A32D16
VME_A32D8
VME_A24D64
VME_A24D32
VME_A24D16
VME_A24D8
VME_A16D64
VME_A16D32
VME_A16D16
VME_A16D8

Priviledge mode selector (choose one):
VME_PRIV
VME_NONPRIV

Program/Data Space Selector (choose one):
VME_PROGRAM
VME_DATA

If no privilege mode selector is specified, VME_NONPRIV is assumed. If no program/data space selector is specified, VME_DATA is assumed.

If *shared* is zero, then the window is opened for exclusive use. If *shared* is nonzero, then the open function searches for a window that is already opened, perhaps by another processor, having the same *busaddr*, *length*, and *attr*. If *ixa_VME_open* finds such a window, then that window is used. Otherwise, *ixa_VME_open* opens a new window. Because VME devices support a small, fixed number of windows (8 for the IXA family), and because multiple processors may wish to access the same VME region, shared windows offer a way to conserve windows.

Note that these functions are only meaningful on IXA VME boards.

RETURN STATUS:

0 indicates no windows available, or length out of range; else nonzero window ID.

NOTES:

ixa_VME_open is functionally the same as the VxWorks board support library function *sysBusOpen*.

SEE ALSO:

ixa_VME_close, ixa_VME_map, ixa_VME_remap.

EXAMPLE:

The following code fragment opens a VME window, accesses a VME location, and closes the window. Note the use of *ixa_VME_map* to map the VME address to an address within the local processor's address space.

```
/*
 * Read a control register on some other
 * board.  Board located at VBASE on
 * the VME bus.
 */

#define VBASE 0x20000000      /* some other board */
#define VSIZE 0x00001000      /* size of window   */

int read_control_reg (void)
{
     int handle;              /* which VME window */
     long read_value, *ptr;

     handle = ixa_VME_open (VBASE, VSIZE, A32D32, 0);

     /* if open error, return -1 */
     if (handle <= 0)
          return (-1);

     /* map address to a local address */
     ptr = ixa_VME_map (handle, VBASE, 0);

     /* read the register value */
     read_value = *ptr;

     /* close the VME window */
     ixa_VME_close (handle);

     /* return value to caller */
     return (read_value);
}
```

ixa_VME_close

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

int ixa_VME_close ( int handle );
```

DESCRIPTION:

The function *ixa_VME_close* closes a VME window. *handle* is a handle returned by *ixa_VME_open*. If the window was shared, then *ixa_VME_close* does not deallocate the window, since other processors may be using it. If the window was not shared, then close frees up the resources allocated by the window.

RETURN STATUS:

0 indicates successful closure.

NOTES:

ixa_VME_close is functionally the same as the VxWorks board support library function *sysBusClose*.

ixa_VME_map

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

void *ixa_VME_map ( int handle,
                    void *busaddr,
                    unsigned long *length );
```

DESCRIPTION:

The function *ixa_VME_map* translates the VME address *busaddr* to an address within local address space. Accessing this local address space results in direct access to the VME bus through the window whose handle is *handle*.

busaddr must lie within the window boundaries established by the *ixa_VME_open* function call.

If the last parameter *(*length*) is nonzero, then *ixa_VME_map* computes the number of bytes from *busaddr* to the end of the window, and deposits this value at **length*. Applications must not address memory beyond the resulting address plus this length. If the last parameter is zero, *ixa_VME_map* skips this step.

RETURN STATUS:

This function returns the special constant (void *) 0xffff_ffff if *handle* is invalid, or if *busaddr* does not fall within the range of the window as defined by *ixa_VME_open.*

NOTES:

ixa_VME_map is functionally the same as the VxWorks board support library function *sysBusMap*.

ixa_VME_remap

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

void *ixa_VME_remap ( int handle,
                      void *busaddr,
                      unsigned long length,
                      unsigned long attr     );
```

DESCRIPTION:

The function *ixa_VME_remap* resets the mapping of a previously opened window. The VME address, *length*, and *attr* can all be changed. This is functionally the same as closing the window and opening a new window with these new parameters.

As with *ixa_VME_open*, *busaddr* will be rounded down to the next natural boundary, and *length* will be rounded up to the next natural boundary. A *length* of zero specifies the maximum window size.

RETURN STATUS:

0 indicates success; otherwise the length or attributes are invalid.

NOTES:

ixa_VME_map is functionally the same as the VxWorks board support library function *sysBusReMap*.

Remapping a shared window in use by other processors will have disasterous side effects, unless applications programs take steps to coordinate the remapping among all processors that are using the window.

SEE ALSO:

ixa_VME_open for a description of attribute and length parameters.

ixa_VME_read, ixa_VME_write

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

int ixa_VME_read   (  void *src,
                      void *dest,
                      int nbytes,
                      unsigned long attr  );

int ixa_VME_write  (  void *src,
                      void *dest,
                      int nbytes,
                      unsigned long attr  );
```

DESCRIPTION:

These functions move data between local memory and VME memory. The functions do this by opening a temporary window, moving the data, and closing the window.

For read operations, *src* is a VME address, and *dest* is a local memory address. For write operations, *src* is a local memory address, and *dest* is a VME address. Because the processor performs the read or write operations by copying through a window, these functions require no address translation for the local address. This stands in contrast to the DMA functions that require the local address to be an on-board PCI address.

attr selects the data size, address size, priviledge mode, program or data selector, and block and posted write transfer options. *attr* is constructed from the logical-OR of the following options:

Address Width and Data Width (choose one):

VME_A32D64
VME_A32D32
VME_A32D16
VME_A32D8
VME_A24D64
VME_A24D32
VME_A24D16
VME_A24D8
VME_A16D64
VME_A16D32
VME_A16D16
VME_A16D8

Priviledge mode selector (choose one):
VME_PRIV
VME_NONPRIV

Program/Data Space Selector (choose one):
VME_PROGRAM
VME_DATA

Block Transfer/Write Post Selection:
VME_BLT
VME_PWEN

If no privilege mode selector is specified, VME_NONPRIV is assumed. If no program/data space selector is specified, VME_DATA is assumed. If VME_BLT is not specified, single-cycle transfers are assumed. If VME_PWEN is not selected, write posting is not employed.

Enabling write posting greatly improves performance of write operations. However, it decouples processor write operations from VME write operations. This feature introduces pipelining that allows the processor to run ahead of the VME bus transactions. As a result, when a VME bus write with write posting fails, the address does not correlate to the processor's write address. The Universe II signals a posted write VME bus error by raising the INT_VECTOR_VERR interrupt. This interrupt is non-sharable, meaning applications software must coordinate use of this interrupt among multiple processors.

VME read and write operations are synchronous by nature. The functions return a status of one or more of the following error codes OR-ed together:

VME_SUCCESS	(zero) – successful completion
VME_ERROR_PARAMS	invalid parameters
VME_ERROR_BUSERR	bus error/address err
VME_ERROR_PCIERR	PCI bus error

Note that this function is meaningful only on IXA VME boards.

RETURN STATUS:

0 indicates successful completion of the read or write operation.

NOTES:

ixa_VME_read and *ixa_VME_write* are functionally the same as the VxWorks board support library function *sysBusRead* and *sysBusWrite*.

Data movement unit size (byte, word or longword) and address alignment requirements are determined by the data size as indicated in the *attr* fields.

For read operations and non-posted write operations, the function checks for bus errors by sampling status bits for every 256 units of data moved. Because these error status bits can be set by other processors when they generate bus errors, applications should take care to coordinate read and write activities among several processors.

ixa_VME_peek, ixa_VME_poke

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

int ixa_VME_peek   (  void *vmeaddr,
                      void *dest,
                      unsigned long attr  );

int ixa_VME_poke   (  void *vmeaddr,
                      unsigned long src,
                      unsigned long attr  );
```

DESCRIPTION:

These functions move a single byte, word or longword between local memory and VME memory. The functions do this by opening a temporary window, moving the data, and closing the window.

For peek operations, *vmeaddr* is a VME address, and *dest* is a local memory address. For poke operations, *vmeaddr* is a VME address, and *src* is the data to be deposited.

attr selects the data size, address size, priviledge mode, and program or data selector. *attr* is constructed from the logical-OR of the following options:

Address Width and Data Width (choose one):

VME_A32D64
VME_A32D32
VME_A32D16
VME_A32D8
VME_A24D64
VME_A24D32
VME_A24D16
VME_A24D8
VME_A16D64
VME_A16D32
VME_A16D16
VME_A16D8

Priviledge mode selector (choose one):

VME_PRIV
VME_NONPRIV

Program/Data Space Selector (choose one):
VME_PROGRAM
VME_DATA

If no privilege mode selector is specified, VME_NONPRIV is assumed. If no program/data space selector is specified, VME_DATA is assumed.

VME read and write operations are synchronous by nature. The functions return a status of one or more of the following error codes OR-ed together:

VME_SUCCESS	(zero) – successful completion
VME_ERROR_PARAMS	invalid parameters
VME_ERROR_BUSERR	bus error/address err
VME_ERROR_PCIERR	PCI bus error

Note that this function is meaningful only on IXA VME boards.

RETURN STATUS:

0 indicates successful completion of the read or write operation.

NOTES:

These functions are equivalent to the VxWorks board support library functions *sysBusPeek* and *sysBusPoke*.

Data movement unit size (byte, word or longword) and address alignment requirements are determined by the data size as indicated in the *attr* fields.

These functions check for bus errors by sampling status bits after the move operation. Because these error status bits can be set by other processors when they generate bus errors, applications should take care to coordinate read and write activities among several processors.

ixa_VME_probe

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

int ixa_VME_probe  (  void *vmeaddr,
                      unsigned long attr  );
```

DESCRIPTION:

The function *ixa_VME_probe* reads a single byte, word or longword from VME memory at address *vmeaddr,* sensing for bus errors. The function does this by opening a temporary window, reading, and closing the window.

Systems having configurations with optional VME boards should probe for those boards during software initialization in order to determine which boards are present. After initialization, applications programs can use the normal VME read, write, window, or DMA functions to access those boards that are present.

attr selects the data size, address size, priviledge mode and program or data selector. *attr* is constructed from the logical-OR of the following options:

Address Width and Data Width (choose one):

VME_A32D64
VME_A32D32
VME_A32D16
VME_A32D8
VME_A24D64
VME_A24D32
VME_A24D16
VME_A24D8
VME_A16D64
VME_A16D32
VME_A16D16
VME_A16D8

Priviledge mode selector (choose one):

VME_PRIV
VME_NONPRIV

Program/Data Space Selector (choose one):

VME_PROGRAM

VME_DATA

If no privilege mode selector is specified, VME_NONPRIV is assumed. If no program/data space selector is specified, VME_DATA is assumed.

This function returns a status of one or more of the following error codes OR-ed together:

VME_SUCCESS	(zero) – successful completion
VME_ERROR_PARAMS	invalid parameters
VME_ERROR_BUSERR	bus error/address err
VME_ERROR_PCIERR	PCI bus error

Note that this function is meaningful only on IXA VME boards.

RETURN STATUS:

0 indicates successful completion of the read or write operation.

NOTES:

This function is equivalent to the VxWorks board support library function *sysBusProbe*.

Data movement unit size (byte, word or longword) and address alignment requirements are determined by the data size as indicated in the *attr* fields.

This function checks for bus errors by sampling status bits after the read operation. Unlike peek and poke, the probe function uses a semaphore to insure that multiple probe operations do not conflict with each other. However, these error status bits can be set when other processors generate bus errors using window reads or writes. Applications should take care to coordinate read and write activities among several processors.

ixa_VME_int_enable, ixa_VME_int_disable

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_int.h>

ixa_VME_int_enable ( unsigned int vme_intr );
ixa_VME_int_disable ( unsigned int vme_intr );
```

DESCRIPTION:

The functions *ixa_VME_int_enable* and *ixa_VME_int_disable* enable and disable VME interrupts for a specified VME level. *vme_intr* is a value between 1 and 7.

VME interrupts are not shareable. That is, only one processor on a board can service a VME interrupt of a given level. If two processors enable the same VME interrupt, interrupts will be routed only to the last processor that enabled the interrupt. Different processors on a board may, however, enable different interrupt levels without conflict.

Note that this function is meaningful only on IXA VME boards.

RETURN STATUS:

0 indicates successful completion.

NOTES:

ixa_VME_int_enable and *ixa_VME_int_enable* are functionally the same as the VxWorks board support library function *sysBusEnable* and *sysBusDisable*.

ixa_VME_getvect, ixa_VME_setvect

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

ixa_VME_getvect   (     unsigned int level,
                        void **fp               );

ixa_VME_setvect   (     unsigned int level,
                        void (*fp)()            );
```

DESCRIPTION:

These functions establish or retrieve the interrupt service routine entry point for a given VME interrupt level. *level* is a value between 1 and 7 inclusive. *fp* is a pointer to an interrupt service routine (for setvect), or a pointer to the location where the ISR entry point will be deposited (for getvect). The interrupt service function should accept a single integer parameter.

When an interrupt service routine is installed and the interrupt is enabled, a VME interrupt results in the VME controller automatically acknowledging the interrupt, retrieving the interrupt vector from the interrupting card. The VME controller then interrupts the processor that enabled the interrupt. The VME controller automatically inhibits further interrupts at that same VME level as part of the acknowledgement process.

The interrupted processor runs an interrupt service shell within IXA tools. This shell retrieves the vector from the controller, re-enables the controller to service more interrupts, and calls the installed user interrupt service routine, passing the vector as the first and only input parameter. The vector value will be in the range 0 through 255, unless an error was detected while retrieving the vector, in which case the vector value will exceed 255.

RETURN STATUS:

0 indicates success, else a parameter is out of range.

ixa_VME_int_gen

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

int ixa_VME_int_gen     ( unsigned int level,
                          unsigned int vector );
```

DESCRIPTION:

The function *ixa_VME_int_gen* generates a VME interrupt signal. The level of the signal is determined by *level.* Valid values for *level* are 1 through 7. *vector* is the vector associated with the interrupt which may be 0 through 255.

The VME controller on the CHAMP board clears the low order bit of the interrupt vector value, so interrupted boards will always see an even vector.

The VME controller contains a single outgoing vector register. Should a processor generate an interrupt with a vector value of y, and then generate another interrupt having vector z before the first interrupt is acknowledged, then both interrupts will be accompanied by a vector value of z, assuming y is eventually acknowledged.

Note that this function is meaningful only on IXA VME boards.

RETURN STATUS:

0 indicates success, else parameter error.

ixa_VME_mailbox_enable, ixa_VME_mailbox_disable

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_int.h>

ixa_VME_mailbox_enable  ( int mbox );
ixa_VME_mailbox_disable ( int mbox );
```

DESCRIPTION:

The functions *ixa_VME_mailbox_enable* and *ixa_VME_mailbox_disable* enable and disable interrupts for a specified mailbox. *mbox* is a value between 0 and 3.

Mailbox interrupts are not shareable. That is, only one processor on a board can service a particular mailbox interrupt. If two processors enable the same mailbox interrupt, interrupts will be routed only to the last processor that enabled the interrupt. Different processors on a board may, however, enable different mailbox interrupts without conflict.

Note that this function is meaningful only on IXA VME boards.

RETURN STATUS:

0 indicates successful completion.

NOTES:

ixa_VME_mailbox_enable and *ixa_VME_mailbox_enable* are functionally the same as the VxWorks board support library function *sysMailboxEnable* and *sysMailboxDisable*.

ixa_VME_mailbox_getvec, ixa_VME_mailbox_setvect

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

ixa_VME_mailbox_getvect   (     int mbox,
                                void **fp               );

ixa_VME_mailbox_setvect (       int mbox,
                                void (*fp)()            );
```

DESCRIPTION:

These functions establish or retrieve the interrupt service routine entry point for a specific mailbox interrupt. *mbox* is a mailbox identifier between 0 and 3 inclusive. *fp* is a pointer to an interrupt service routine (for setvect), or a pointer to the location where the ISR entry point will be deposited (for getvect). The interrupt service function should accept a single long integer parameter.

When a mailbox interrupt service routine is installed and the interrupt is enabled, a write to the mailbox results in a VME interrupt. The write may be from another board, or from a local processor on the same board through ixa_VME_mailbox_post. The VME controller interrupts the processor that enabled the interrupt.

The interrupted processor runs an interrupt service shell within IXA tools. This shell retrieves the mailbox contents, and passes this 32 bit value as the first and only input parameter.

RETURN STATUS:

0 indicates success, else a parameter is out of range.

ixa_VME_mailbox_setvect is functionally the same as the VxWorks board support library function *sysMailboxConnect*.

ixa_VME_mailbox_post, ixa_VME_mailbox_set

CALLING SEQUENCE:

```
#include <ixa.h>
#include <ixa_vme.h>

int ixa_VME_mailbox_post( int mbox,
                          long value );

int ixa_VME_mailbox_set ( int mbox,
                          long value );
```

DESCRIPTION:

The function *ixa_VME_mailbox_post* posts *value* to the on-board mailbox *mbox*. *mbox* is an integer between 0 and 3. If a processor has enabled interrupts for the specified mailbox, posting will cause an interrupt, causing that processor to call its mailbox interrupt service routine with an input of *value*.

The function *ixa_VME_mailbox_set* is similar to *ixa_VME_mailbox_post*, except that the set function alters the mailbox contents without generating an interrupt. As such, this function can be used to clear a mailbox within a mailbox interrupt service routine.

Note that these functions are meaningful only on IXA VME boards.

RETURN STATUS:

0 indicates success, else parameter error.

ixa_vme_to_local, ixa_local_to_vme

CALLING SEQUENCE:

```
#include <ixa.h>

void *ixa_vme_to_local (void *vme_adrs);
void *ixa_local_to_vme (void *local_adrs);
```

DESCRIPTION:

The function *ixa_vme_to_local* provides a local address that corresponds to a VME bus address. The local address references either on-board memory or a window onto the VME backplane. Windows onto the VME backplane are dynamically created, if needed.

The function *ixa_local_to_vme* provides the VME bus address that corresponds to a local address. The VME address is that which would be seen by another processor on the VME backplane.

These functions are only valid on the IOPlus.

RETURN STATUS:

The function *ixa_vme_to_local* returns a pointer to a local address, either on-board or within a VME window.

The function *ixa_local_to_vme* returns a pointer to a VME bus address.

If the input address is invalid, both functions simply return the input address to the caller.

printf

CALLING SEQUENCE:

```
#include <ixa.h>

void printf ( char *format, ...);
```

DESCRIPTION:

The function *printf* prints the string, *format*, to the serial terminal. Function *printf* adheres to the standard ANSI C printf function and therefore allows multiple args.

Please note that the IOPlus processor must be monitoring the processor that is transmitting the formatted output string in order for the string to appear on the serial port. The IOPlus processor can only monitor one processor at a time.

RETURN STATUS:

None

putchar

CALLING SEQUENCE:

```
#include <ixa.h>

void putchar ( char c );
```

DESCRIPTION:

The function *putchar* transmits the character*, c*, to the serial port. This function should not be called with interrupts disabled.

Please note that the IOPlus processor must be monitoring the processor that is transmitting the character in order for the character to appear on the serial port. The IOPlus processor can only monitor one processor at a time.

RETURN STATUS*:*

None

puts

CALLING SEQUENCE:

```
#include <ixa.h>

void puts ( char *s );
```

DESCRIPTION:

The function *puts* outputs the null terminated string, *s*, to the serial port. This function should not be called with interrupts disabled.

Please note that the IOPlus processor must be monitoring the processor that is transmitting the output string in order for the string to appear on the serial port. The IOPlus processor can only monitor one processor at a time.

RETURN STATUS:

None

sprintf

CALLING SEQUENCE:

```
#include <ixa.h>

void sprintf ( char *result, char *format, ...);
```

DESCRIPTION:

The function *sprintf* prints the formatted string, *format*, to the address pointed to by *result*. Multiple args can be supported by this function since it adheres to standard ANSI C for I/O.

RETURN STATUS:

None

Chapter 8: Programming the Flash Memory

8.1 Introduction

The flash memory on the IXA7 is connected to the IOPlus. It is used to store initialization code, configuration data, the run-time IOPlus software, and user applications. Only the IOPlus can write to the flash memory, but it can be read by anyone. Functions exist in HostAPI (Chapter 8), IOPlusAPI (Chapter 5) and IXAbsp (Chapter 6) that can command the IOPlus to write sections of flash memory. However, all these approaches require an application to be created before they are useful.

This chapter presents a utility installed with IXAtools that permits the viewing of the board configuration information and the modification of the flash memory contents. This software runs on Windows 95/98/2000, Windows NT, or Windows "Me". It requires an Ethernet connection to both the PC running the software and the IXA board. Finally, it requires VxWorks to be running in the IOPlus.

If VxWorks is not operable, a second utility is presented at the end of this chapter (section 7.8). This utility requires a supported Windows based host, and the HostAPI.dll for the device that is installable with IXAtools.

8.2 The IXA Board Configuration Utility

The IXA Board Configuration Utility (also known as the Ethernet Burn Utility) permits the board configuration settings that are programmed into flash memory to be viewed and modified. With this utility, both application programs and board firmware may be installed or upgraded. Flags located in the flash memory can be set to allow user programs to be loaded into both the SPEs and the IOPlus upon board power up or reset.

To start the utility, simply select "Burn Utility" from the IXATools folder under the Programs folder in the Windows Start menu. A screen similar to Figure 8.1 should appear. Note that the screen consists of a single tabbed dialog box. Each page of the dialog box contains related information, and can be accessed by clicking on the tab name.

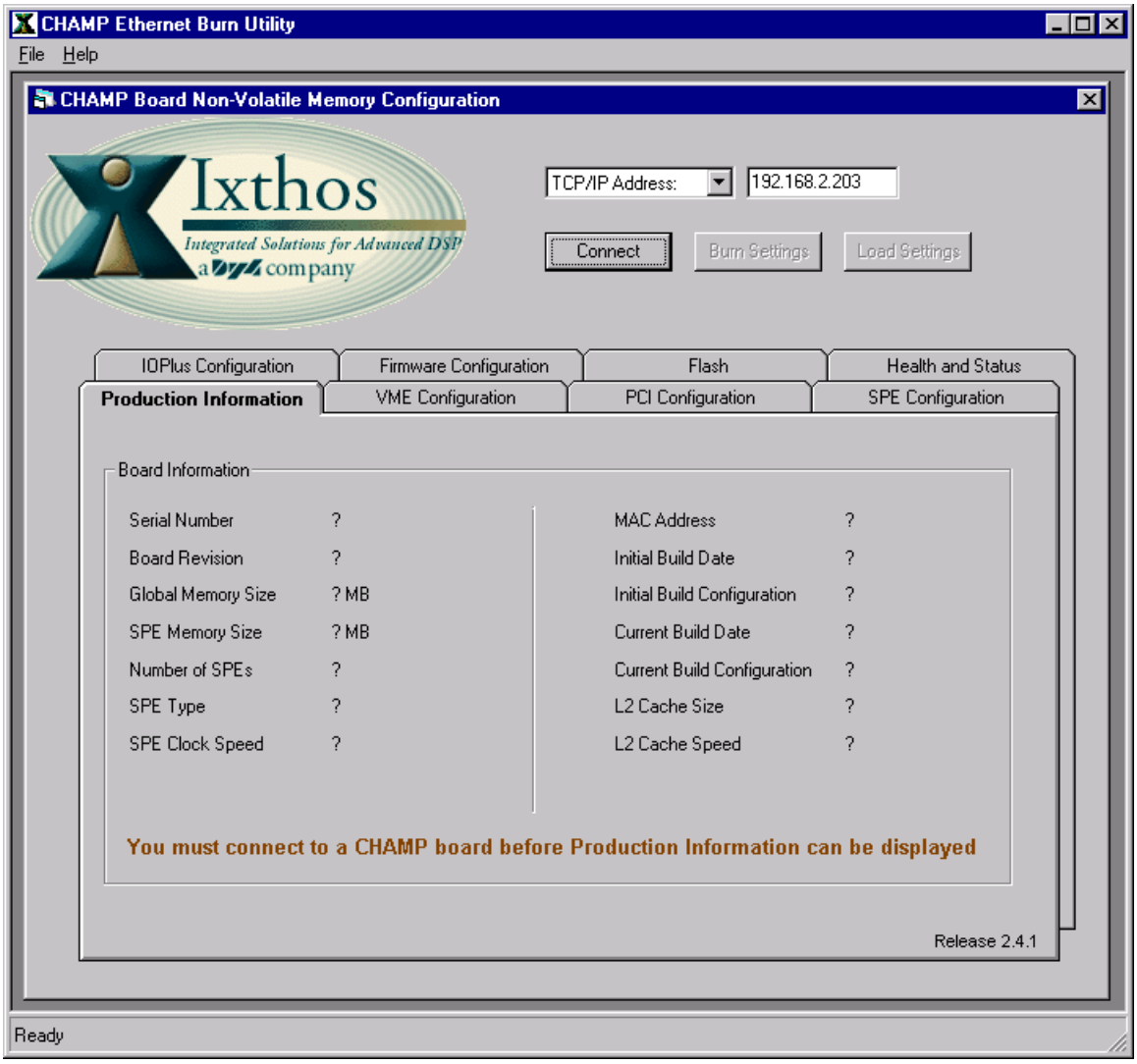


Figure 8.1 - Initial Screen (before connecting to board)

You must establish a connection to the CHAMP board using TCP/IP, before you can use the utility to reburn flash. To begin, type the IP address of the IOPlus on the IXA board whose settings you wish to view/modify (the IP address for the IOPlus is set using VxWorks configuration utilities).

To establish a connection to the board, press the "Connect" button. The message window shown in Figure 8.2 should briefly appear, and then the Product Information screen should update with information read from the CHAMP board (as shown in Figure 8.5).

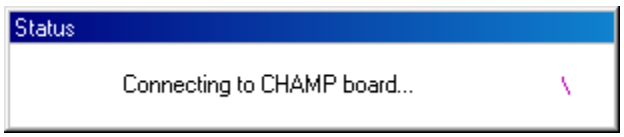


Figure 8.2 – "Connecting" message window

If the dialog box shown in Figure 8.3 appears, the program is having difficulty connecting to the board with the specified IP address. Verify that the correct IP address for the IOPlus has been entered, and that the CHAMP board is running VxWorks in the IOPlus. To retry the connection attempt, press the "Retry" button. To cancel the connection attempt, press the "Cancel" button.

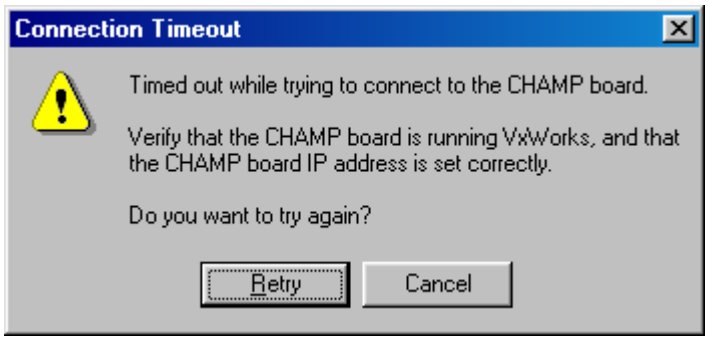


Figure 8.3 – Connection Timeout Dialog Box

If the dialog box shown in Figure 8.4 appears when you are attempting to connect to the CHAMP board, then the version of the VxWorks BSP running on the IOPlus needs to be updated by installing the latest IXATools release. Note that the Ethernet Burn Utility will still function when older VxWorks BSPs are running on the IOPlus, but certain Burn Utility functions will be disabled.

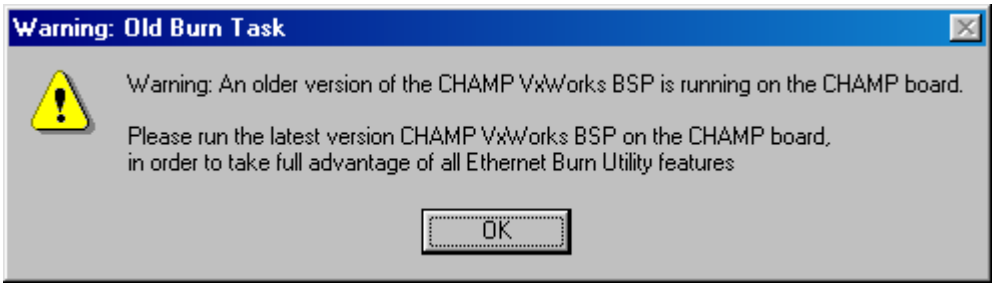


Figure 8.4 – Old Burn Task

After the Burn Utility has successfully connected to the CHAMP board, the Product Information dialog box will display information about the CHAMP board, as shown in Figure 8.5.

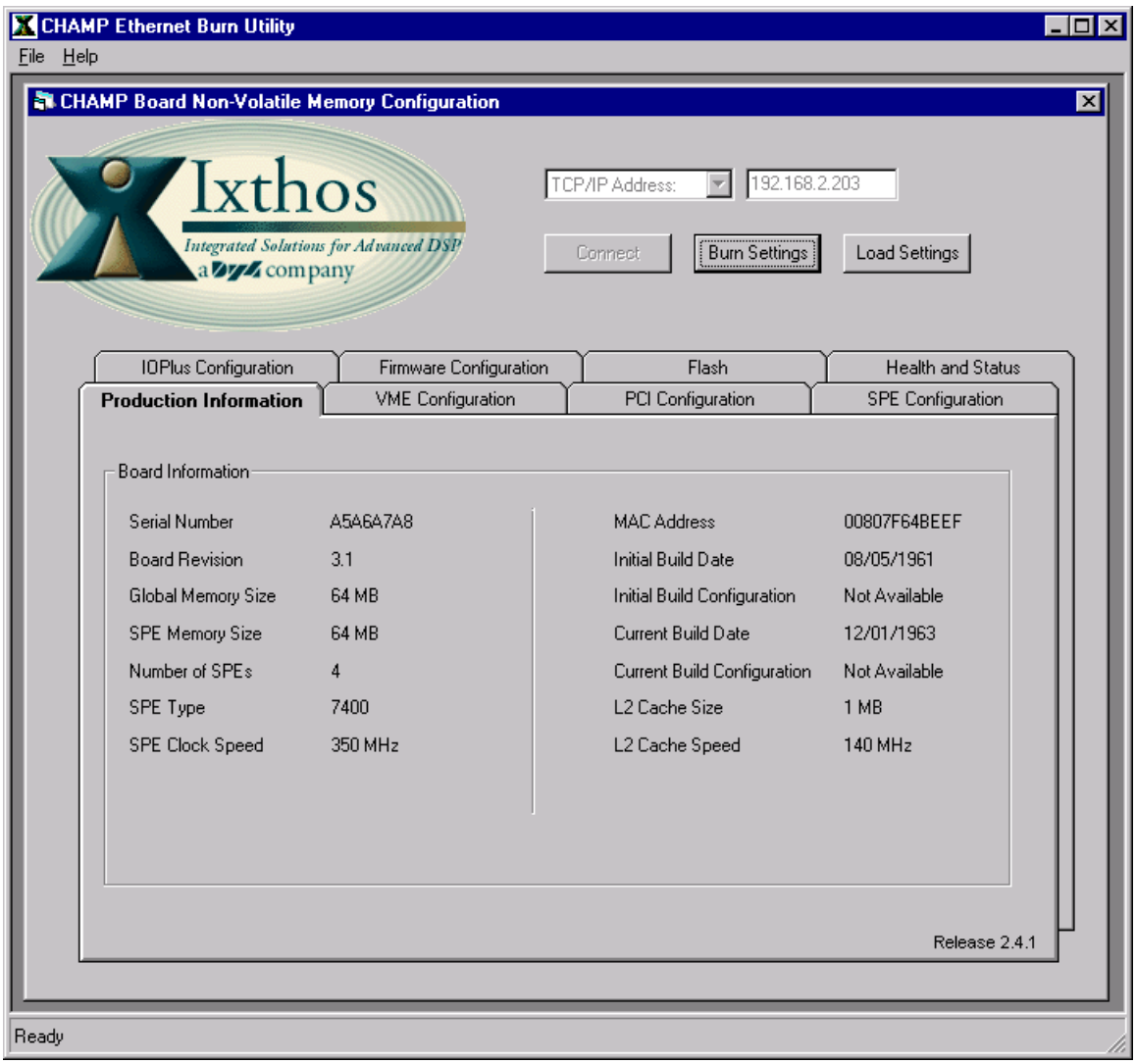


Figure 8.5 – Product Information Page (after successful connection)

A number of buttons are located in the upper right corner of the dialog box. After successfully connecting to the CHAMP board, the "Burn Settings" and "Load Settings" buttons will be enabled. The "Connect" button will be disabled, because you are already connected to the CHAMP board.

Clicking the "Burn Settings" button causes the Burn Utility to burn the current configuration into the CHAMP board with the specified IP address. This button should be clicked after configuration changes have been made. When this button is clicked, changed parameters are written to the IXA board, and stored in non-volatile flash memory.

Clicking the "Load Settings" button causes the Burn Utility to read the current board configuration from the CHAMP. Clicking this button will cause any that you have made

to the displayed configuration to be overwritten. Note that this button changes the Burn Utility dialog boxes, but does not update any settings stored in non-volatile memory on the CHAMP board.

Product Information

Figure 8.5 shows the Product Information screen. This screen displays various read-only board configuration parameters. This information will be useful primarily to Dy 4 Systems customer support.

8.3 VME Configuration

The VME Configuration window, shown in Figure 8.6, allows you to configure the VMEbus interface of the IXAboard. The following VMEbus parameters can be configured:

VME Address Mode

The VME Address Mode setting indicates how the VME base address of the CHAMP board is determined. The VME Address Mode setting can be over-ridden by removing board jumper JM6, in which case the board address is determined by jumpers JM1-JM4, where JM1 indicates bit 31 of the address and JM4 indicates bit 28 of the address. Installed jumpers set the address bit corresponding to the header on which the jumper is installed, while removed jumpers clear the corresponding address bit.

The VME Address Mode may be chosen from the following options:

Fixed at 0x80000000	The VME base address is 0x80000000.
Read from Jumpers	The VME base address is determined by board jumpers JM1-JM4.
Read from Flash	The VME base address is set to the value burned into the VME Base Address parameter.

Geographical Addressing

The VME base address is determined by the geographical address pins settings for the VMEbus backplane slot in which the board resides.

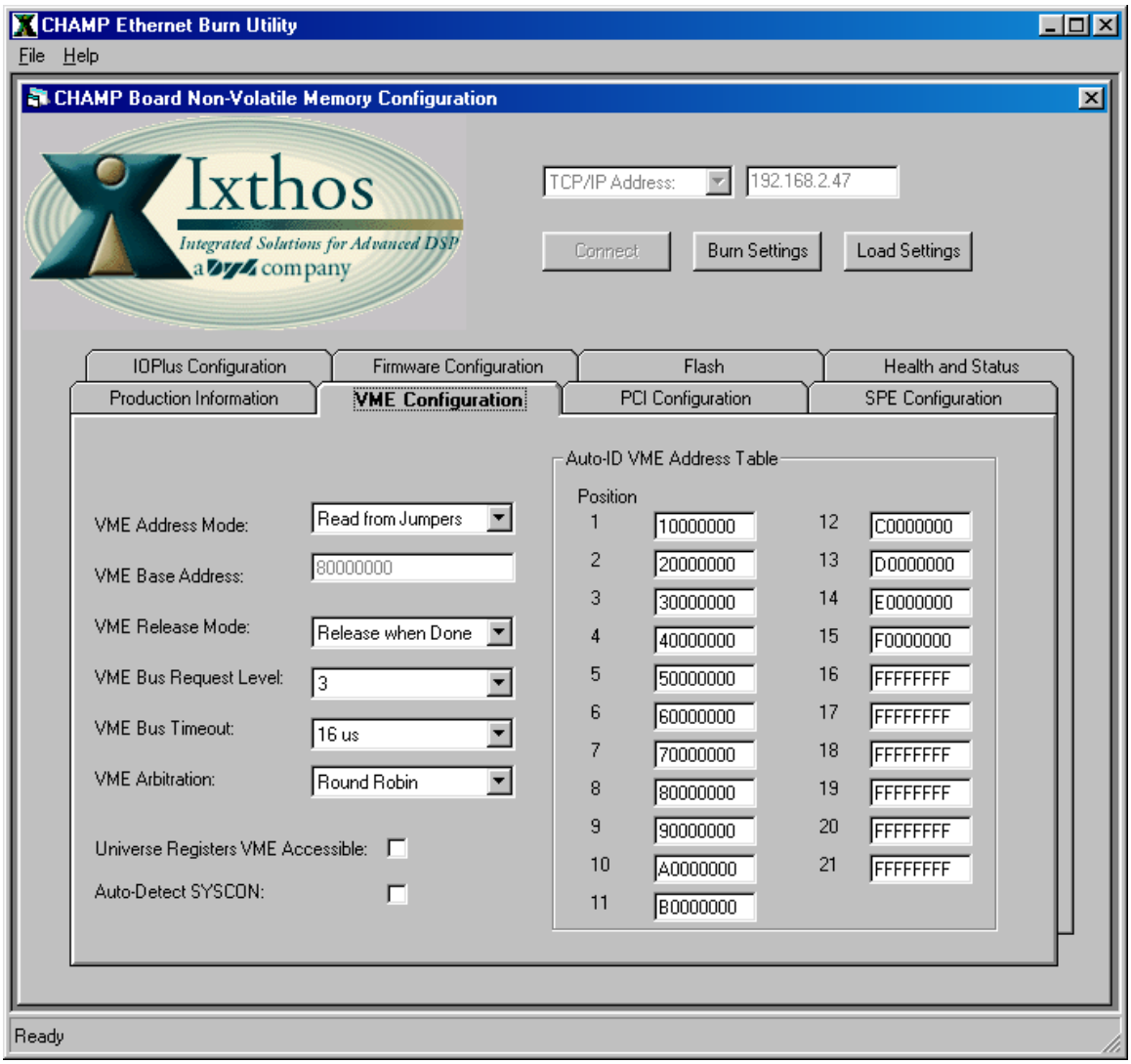


Figure 8.6 - VME Configuration Page

VME64 Auto-ID

Automatic slot identification and VME address configuration occurrs as described in the VME64 Specification.

DY4 Auto-ID

The DY4 Auto-ID function identifies the relative position of each board in the system without using jumpers or on-board information. The resulting ID number indicates the relative postion of the board in the VME backplane starting with the board in slot 1. The ID is then used as an index into the DY4 Auto-ID Address Table to determine the VME base address for the board.

VME Base Address

The VME Base Address parameter determines the VME address of the IXA board when the VME Address Mode is set to **Read from Flash**.

VME Release Mode

The VME Release Mode determines how the board relinquishes ownership of the VMEbus. The VME Requestor Mode may be chosen from the following options:

Release When Done The CHAMP board releases ownership of the VMEbus upon completing a VME transfer.

Release On Request The CHAMP board releases ownership of the VMEbus only when another VME master requests it.

VME Request Mode

The VME Request Mode determines how the CHAMP board requests ownership of the VMEbus. The VME Request Mode may be chosen from the following options:

Fair The board does not request the VME bus until there are no other VME bus requests pending at its bus request level. This mode ensures that every requester on an equal bus request level has access to the bus.

Demand The board asserts its bus request regardless of whether there are any pending bus requests. This may create a bus "starving" condition where requesters further down the daisy chain are prevented from ever obtaining bus ownership.

VME Bus Request Level

The VME Bus Request Level determines the level on which the Champ board requests ownership of the VMEbus. The VME Bus Request Level may be chosen from the following options: **0**, **1**, **2**, or **3**.

VME Bus Timeout

The VME Bus Timeout parameter determines the time-out period for VMEbus transactions. The VME Bus Timeout may be chosen from the following options: **16us**, **32us**, **64us**, **128us**, **256us**, **1024us**, or **Disabled**.

VME Arbitration

The VME Arbitration Mode determines how the CHAMP board arbitrates ownership of the VMEbus when it is the system controller (SYSCON). The VME Arbitration Mode may be chosen from the following options:

Round Robin All bus request levels are arbitrated in a round robin mode, repeatedly scanning from levels 3 to 0. Only one grant is issued per level and one owner is never forced from the bus in favor of another requester.

Priority The arbiter issues a bus grant to the highest requesting level, where Bus Request Level 3 (BR3*) has the highest priority and Bus Request Level 0 (BR0*) the lowest. If a Bus Request of higher priority than the current bus owncer becomes asserted, the arbiter asserts Bus Clear (BCLR*) until the owner releases the bus.

Universe Registers VME Accessible

When checked, this option allows the Universe II VME bus bridge registers to be accessible from the VME bus. Activating this mode provides the capability to have an application manipulate the VME interface dynamically, and permits the use of some of the more advance features contained within the Universe. Use of this mode requires a good understanding of the Universe and the VME bus. Note that certain HOSTAPI functions require Universe registers to be accessible from the VME bus.

Auto-Detect SYSCON

The CHAMP board is capable of automatically detecting whether it is the VME bus System Controller (SYSCON). For auto-detection to operate correctly, the VME chassis must either be auto-configuring, or the bus grant and IACK jumpers on the backplane must be installed correctly. If your chassis is auto-configuring or you have installed the backplane jumpers correctly, then this bit may be enabled. If you are not using the CHAMP board as a system controller, then we recommend unchecking the "Auto-Detect SYSCON" option.

Note 1: This option must be checked for the CHAMP board to operate as a VME system controller.

Note 2: If this option is checked and the VME backplane is not jumpered correctly, then the CHAMP board may erroneously operate as VME system controller, even though the CHAMP board is not in the first VME slot. This will cause contention on the VME backplane, and may damage any boards installed in the VME chassis.

Auto-ID VME Address Table

When either SVME 64 Auto-ID or DY4 Auto-ID VME Address Mode is enabled, the resulting ID indicating the relative position of the board is used as an index into this table to determine the VME base address for the board.

8.4 PCI Configuration

The PCI Configuration page, shown in Figure 8.7, allows you to configure PCI bus related parameters on the Champ board. The following PCI bus parameters can be configured:

Latency Timer

The Latency Timer represents a minimum guaranteed number of clocks that the device can act as a master on the PCI bus. A master's latency timer is cleared and suspended whenever it is not asserting FRAME#. Whenever a master asserts FRAME#, the latency timer is enabled to count. If another device has requested to be master when the current master's latency timer expires, the current master will suspend its transfer and relinquish bus mastership. In effect, the Latency Timer controls the tradeoff between high throughput (higher Latency Timer values) and low latency (lower Latency Timer values). The latency can be set for the following PCI devices: PMC 1, MPC107 A/B, PCI Bridge 1, IOPlus, PCI Bridge 2, MPC107 C/D, and PMC 2.

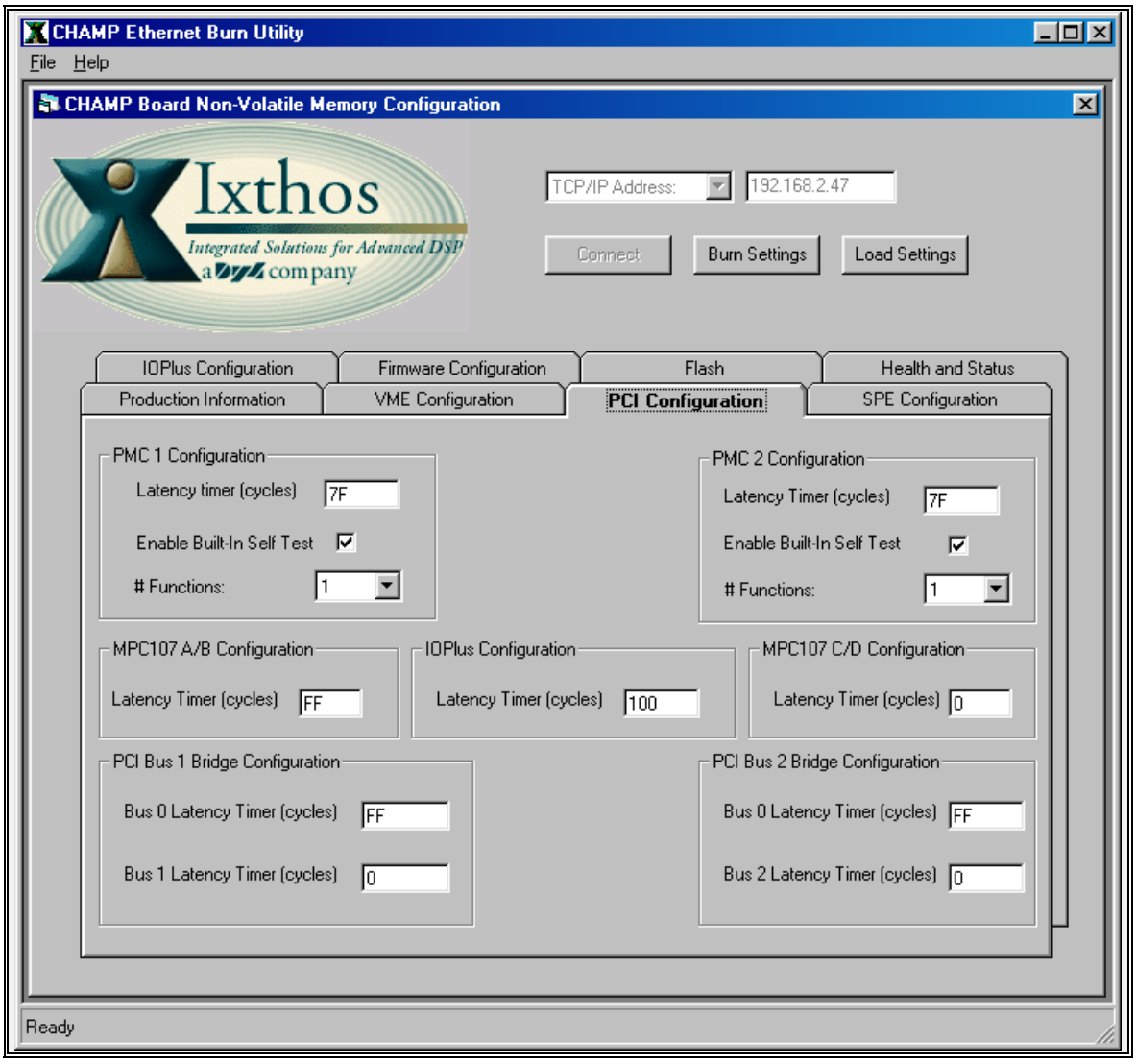


Figure 8.7 - PCI Configuration Page

Enable Built-In-Self Test

This option is used to enable the Built-In-Self Test function of a PMC if the PMC supports this feature. When this option is enabled, the CHAMP board firmware will command the PMC to perform its BIST. If the BIST fails, the red LED on the CHAMP board front panel will be enabled.

Number of Functions

Specifies number of functions on a given PMC that are initialized (maximum of 8). This option may be useful if your PMC has multiple functions, but you only wish to use a subset of these functions.

8.5 SPE Configuration

The SPE Configuration page, shown in Figure 8.4, allows you to configure the SPEs on the Champ board. The following parameters can be configured:

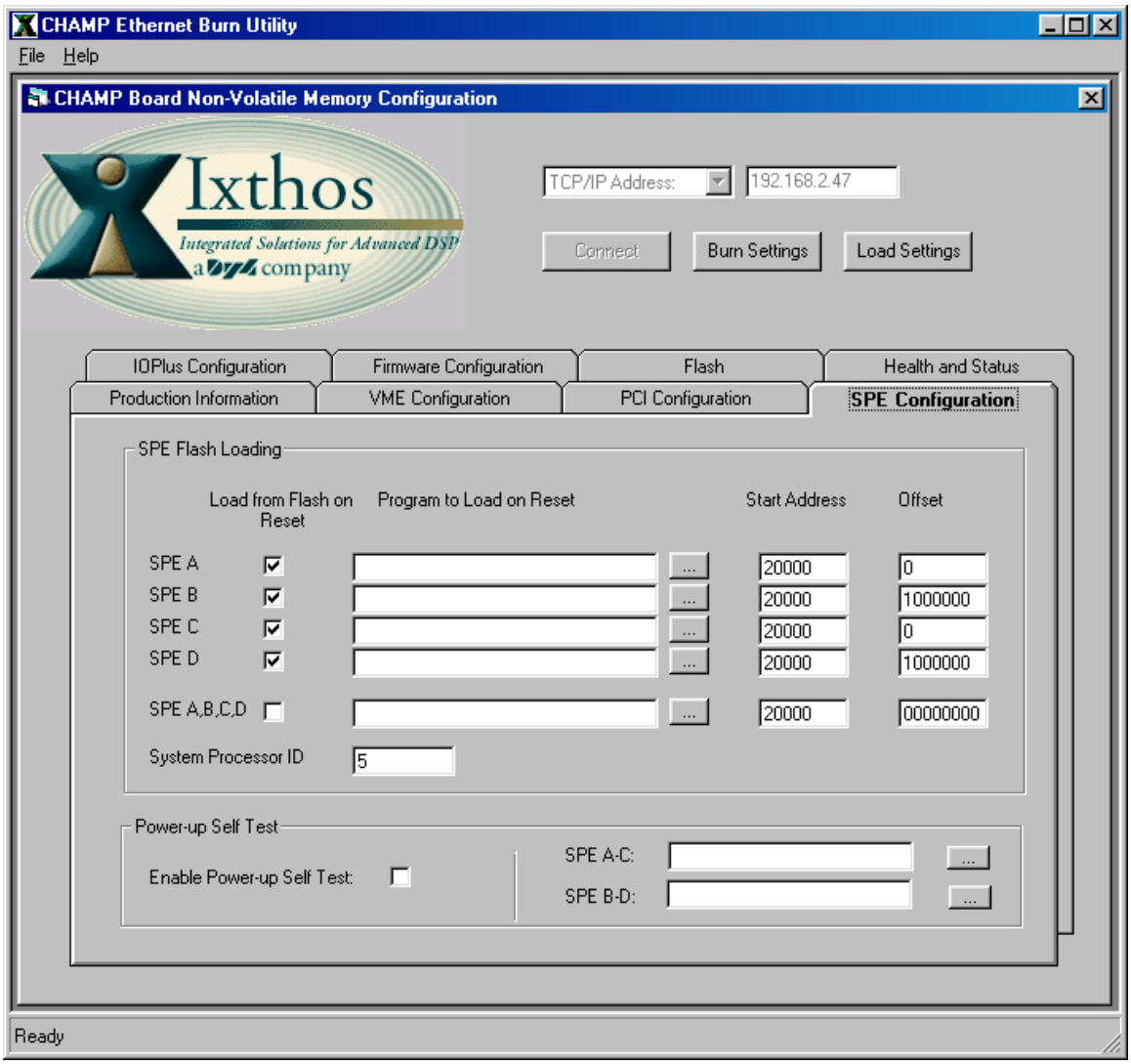


Figure 8.8 - SPE Configuration Page

Load from Flash on Reset

When checked, this option will enable the flash loading of the respective SPE processor upon board reset if jumper JM7 is also installed. Removing jumper JM7 disables all SPE flash loading.

Program to Load on Reset

This field specifies an S-Record format SPE program file to load into the given SPE on board reset. The button to the right of the field can be used to launch a file browser to select the S-Record file. A different program can be specified for each processor. The SPE program download and execution will occur upon board reset if the "Load from Flash on Reset" option is checked and if a jumper is installed on JM7. Removing the jumper from JM7 disables flash load.

Note 1: We recommend burning a single SPE program image into all SPEs, in order to conserve CHAMP board non-volatile memory space. To load the same program into all SPEs, use the "SPE A,B,C,D" field, rather than the individual fields. As you begin entering a filename into the "SPE A,B,C,D" field, the individual fields will disable.

Note 2: The SPE version of VxWorks should always be burned as "SPE A,B,C,D" on a four processor CHAMP board, since four separate copies do not fit in the CHAMP board non-volatile memory.

Note 3: The SPE version of VxWorks should be burned as "SPE A" and "SPE C" on a dual processor CHAMP board.

Note 4: When you burn new SPE program(s) into flash, any existing SPE program(s) will be deleted. Existing programs are deleted to avoid conflicts with the new programs that you are burning. Note that you must therefore burn all SPE program(s) into flash during the same burn. You can not burn a file into SPE A during this burn, then burn another file into SPE B during a subsequent burn. If you attempt to do this, the SPE A program will be deleted when you burn the program for SPE B. If you want to burn programs into flash for both SPE A and SPE B, they must be burned at the same time.

Start Address

The "Start Address" field specifies the address of the entry point for the program loaded into the given SPE. Note that the value entered here is the logical address (i.e. the address as seen by that SPE when its MMU is enabled).

Offset

The "Offset" field specifies the logical to physical address mapping for the given SPE. The value entered here is the offset between logical address 0x00000000 on the given SPE, and address 0x00000000 in the physical memory in which the program is to be loaded. For instance, the common boot code configures SPE A's MMU so that the logical and physical addresses are the same, resulting in an offset of 0x00000000. The common boot code configures SPE B's MMU so that logical address 0x00000000 maps to physical address 0x01000000 in the memory shared by SPE A/B. A similar mapping is performed by the common boot code for SPE C and D

.

If you alter the common boot code address mapping, then this field should be changed from its default values. If you do not alter the common boot code, then the default values do not need to be changed.

System Processor ID

The "System Processor ID" field is used by VxWorks MP when multiple CHAMP boards are installed in a single VME chassis, all of which run VxWorks MP. If you are not using VxWorks MP, you do not need to change this field.

Enable POST

When clicked, this parameter will enable the Power-Up Self Test code that runs in the SPEs on board reset or power-up.

POST A-C, B-D

The "POST A-C" and "POST B-D" fields allow updated versions of the CHAMP board Power-up Self Test software to be loaded in the board non-volatile memory.

8.6 IOPlus Configuration

The IOPlus Configuration page, as shown in Figure 8.9, allows you to configure the IOPlus processor on the Champ board. The following IOPlus parameters can be configured:

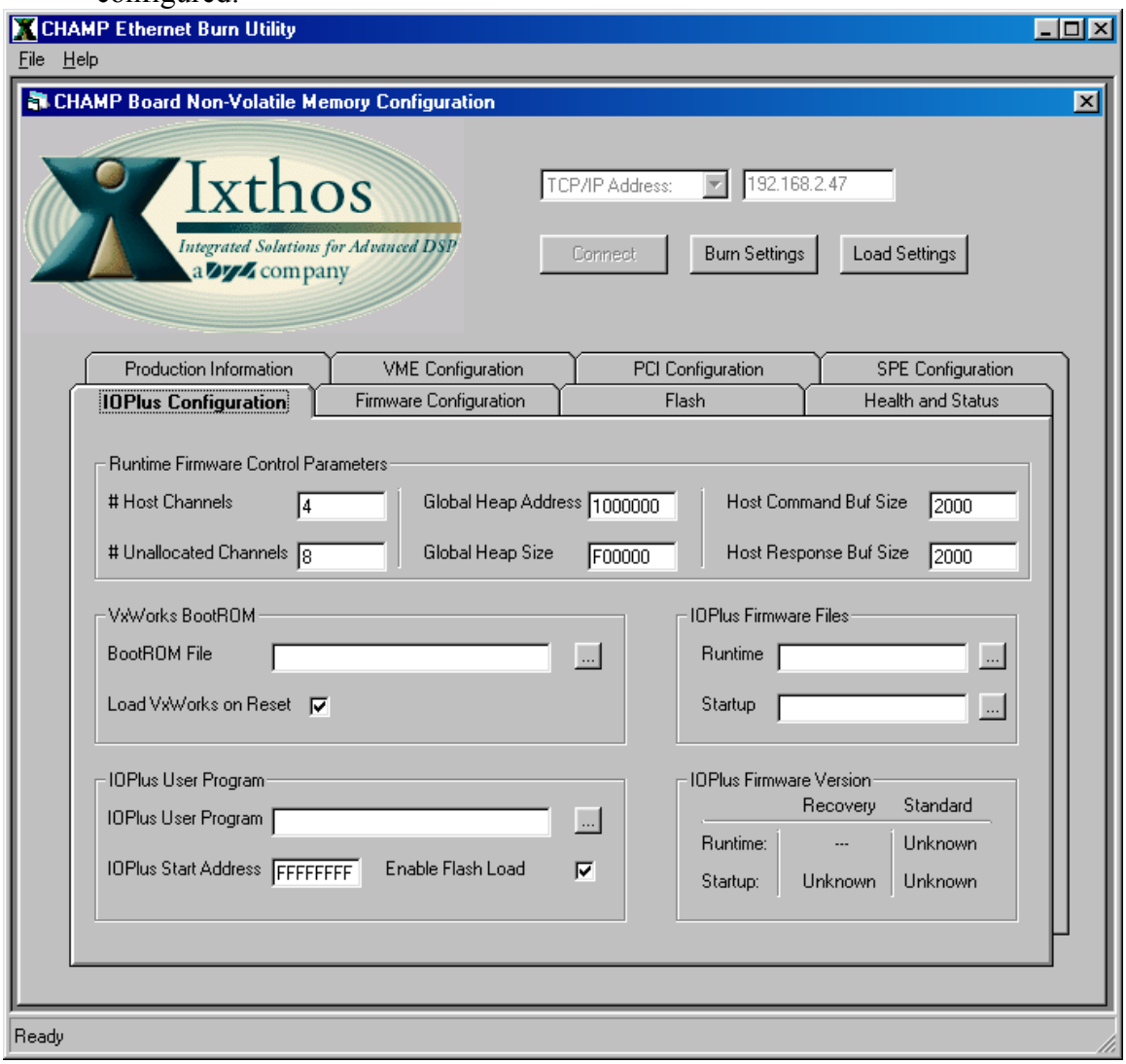


Figure 8.9 - IOPlus Configuration Page

Host Channels

This parameter specifies the number of Command Channels to be allocated for use by host processes. The SmartDMA software running in the IOPlus uses this field.

Unallocated Channels

This parameter specifies the number of Command Channels to be created for use by user applications running on the SPEs. These Command Channels are placed in an unallocated pool, and may be acquired by an SPE application by calling the *ixa_cmd_open* function in the IXA board support library. See Chapter 6 for more infomation on using Command Channels. The SmartDMA software running in the IOPlus uses this field.

Global Heap Address

This parameter specifies the address in global memory of the global shared memory heap. User applications running on the SPE processors can allocate memory from the global heap using the *ixa_malloc*, *ixa_calloc*, or *ixa_realloc* functions. You must ensure that the address range allocated to the global heap does not conflict with memory managed by other programs, such as VxWorks.

Global Heap Size

This parameter specifies the size of the heap in global memory. User applications running on the SPE processors can allocate memory from the global heap using the *ixa_malloc*, *ixa_calloc*, or *ixa_realloc* functions.

Host Command Buf Size

Host applications using the HostAPI library send commands to the IOPlus via the Host Command Buffer located in global memory. This parameter determines the size of the Host Command Buffer. Increasing the size of the buffer allows the host to send larger command blocks, thus possibly increasing the performance of host-IOPlus communications.

Host Response Buf Size

Host applications using the HostAPI library receive command responses from the IOPlus via the Host Response Buffer located in global memory. This parameter determines the size of the Host Response Buffer. Increasing the size of the buffer allows the host to receive larger response blocks, thus possibly increasing the performance of host-IOPlus communications.

VxWorks Boot ROM

This field specifies a VxWorks boot ROM file in S-Record format that will be burned into flash. The button to the right of the field can be used to launch a file browser to

select the VxWorks Boot ROM file. The VxWorks boot ROM program will be loaded and executed by the IOPlus upon board reset if the Enable VxWorks Boot option is checked.

Note 1: Reburning the VxWorks Boot ROM is a potentially dangerous operation. If a bad Boot ROM file is burned into flash, then VxWorks might be unbootable, rendering the Ethernet Burn Utility unusable, since it relies on VxWorks running in the IOPlus. The Boot ROM should only be reburned when upgrading to a new Dy 4 Systems software release, or when burning the final, deployable version of your program into flash (i.e. when you no longer wish to boot VxWorks over Ethernet).

Load VxWorks on Reset

When checked, this parameter will cause the VxWorks Boot ROM to be loaded from flash and executed on the IOPlus, if jumper JM8 is installed.

Note 1: For the VxWorks Boot ROM to execute, this option must be enabled, and JM8 must be installed.

Note 2: Disabling VxWorks booting is an irreversible process, since it renders the Ethernet Burn Utility, which relies on VxWorks, unusable. You should be really, really certain that you want to permanently disable VxWorks booting before selecting this option.

IOPlus Run-Time File

This field is used to upgrade the Run-Time firmware for the IOPlus. The button to the right of the field can be used to launch a file browser to select the file. Upgrading the Run-Time program should only be performed only when upgrading to a new Dy 4 Systems software release, or at the direction of Dy 4 Systems customer support.

IOPlus Start-Up File

This field is used to upgrade the Startup firmware for the IOPlus. The button to the right of the field can be used to launch a file browser to select the file. Upgrading the Startup firmware should only be performed only when upgrading to a new Dy 4 Systems software release, or at the direction of Dy 4 Systems customer support.

Note 1: Reburning the Startup firmware is a potentially dangerous operation. If bad Startup firmware is burned into flash, then the CHAMP board may be unbootable, rendering the Ethernet Burn Utility unusable.

IOPlus User Program

This field specifies a program file in S-Record format that the IOPlus can load from flash memory and execute after completing the board startup. The button to the right of the field can be used to launch a file browser to select the S-Record file. The flash load will occur if the Enable Flash Load option is checked and if a jumper is installed on JM7. Removing the jumper from JM7 disables flash load.

IOPlus Start Address

This field specifies the start address of the IOPlus program that is entered in the IOPlus Flash Program field. Once the program has been loaded after board reset, the IOPlus begins program execution at this address.

Enable Flash Load

When checked, this parameter will enable the flash loading of the IOPlus upon board reset if jumper JM7 is installed. Removing jumper JM7 disables flash load. The IOPlus program to be loaded from flash is specified in the IOPlus Flash Program field.

8.7 Firmware Configuration

The Firmware Configuration page, shown in Figure 8.10, will be primarily used to update the board's firmware to newer releases. You may also need to use this page to restore the board's firmware if the flash gets corrupted. The IXAtools release notes will direct you as to what files need to be used for updates and what files to use for restoring the board's firmware to the current revision.

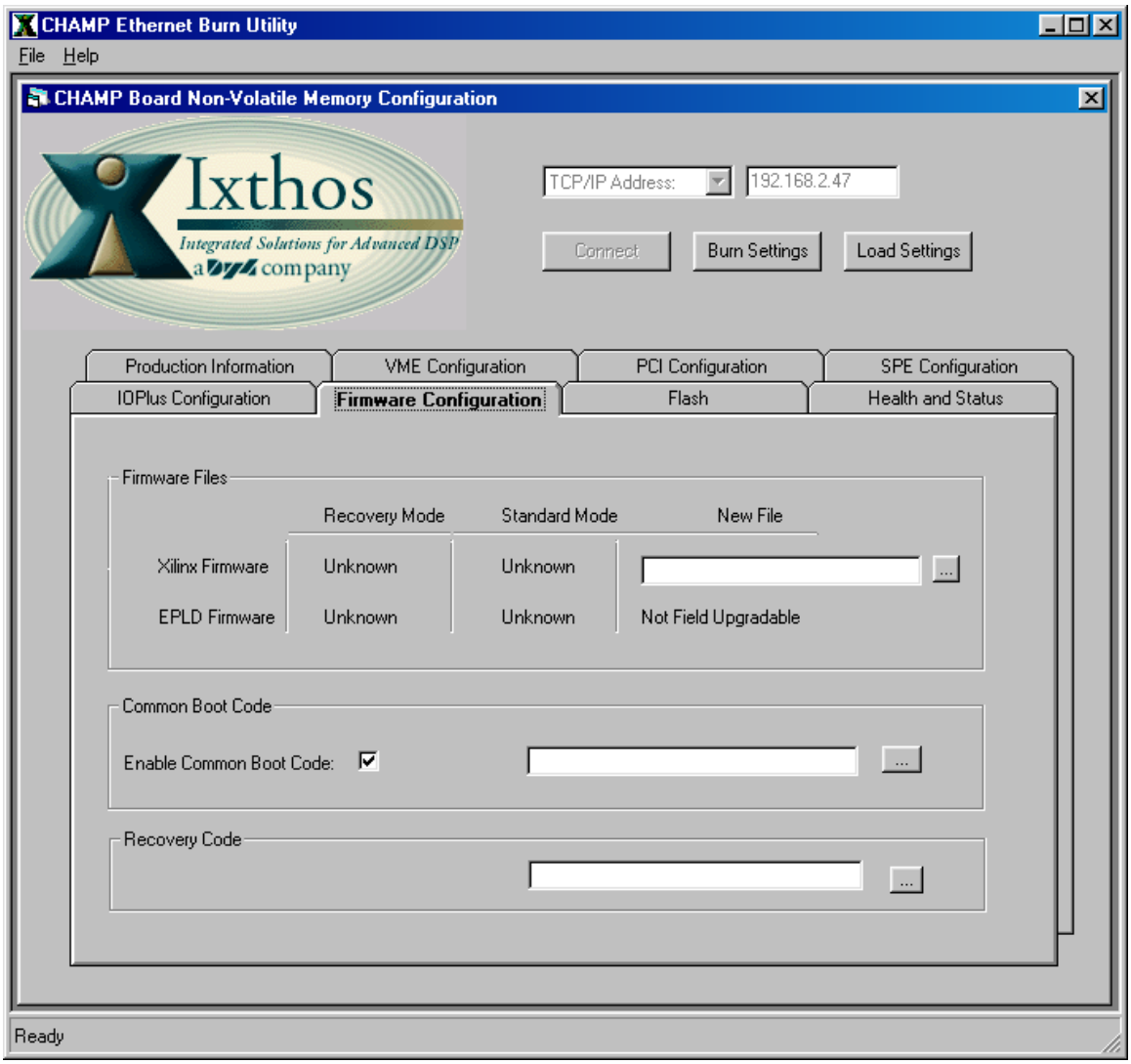


Figure 8.10 - Firmware Configuration Page

Xilinx Firmware

This field allows the Xilinx firmware to be upgraded. The Xilinx firmware implements the Board Resource Manager. This firmware should only be altered when updating to a new Dy 4 Systems software release.

Enable Common Boot Code

Enabling this option causes the CHAMP board common boot code to be run on each SPE as part of the board reset sequence. The common boot code performs various initialization functions on the SPEs, such as enabling the MMU. In most cases, this option should be enabled.

Note 1: The SPE version of VxWorks requires this option to be enabled.

Common Boot Code File Field

This field allows the common boot code to be upgraded.

Recovery Code

This field allows the board recovery firmware to be reburned. The recovery code is used to boot the CHAMP board into a minimal state, so that its flash can be reburned over the VME bus.

8.8 Flash Page

The Flash page, shown in Figure 8.11, displays information about the non-volatile memory on the CHAMP board.

Flash Free / Used Space

This field displays the following information about the state of the CHAMP board non-volatile memory:

Total Space: The total size (in bytes) of the CHAMP board flash memory

Used Space: The total flash memory space (in bytes) currently in use.

Free Space: The total free flash memory space (in bytes).

Maximum contiguous block: The largest free block in flash (in bytes). This is the size of the largest program/data file which can be burned into flash.

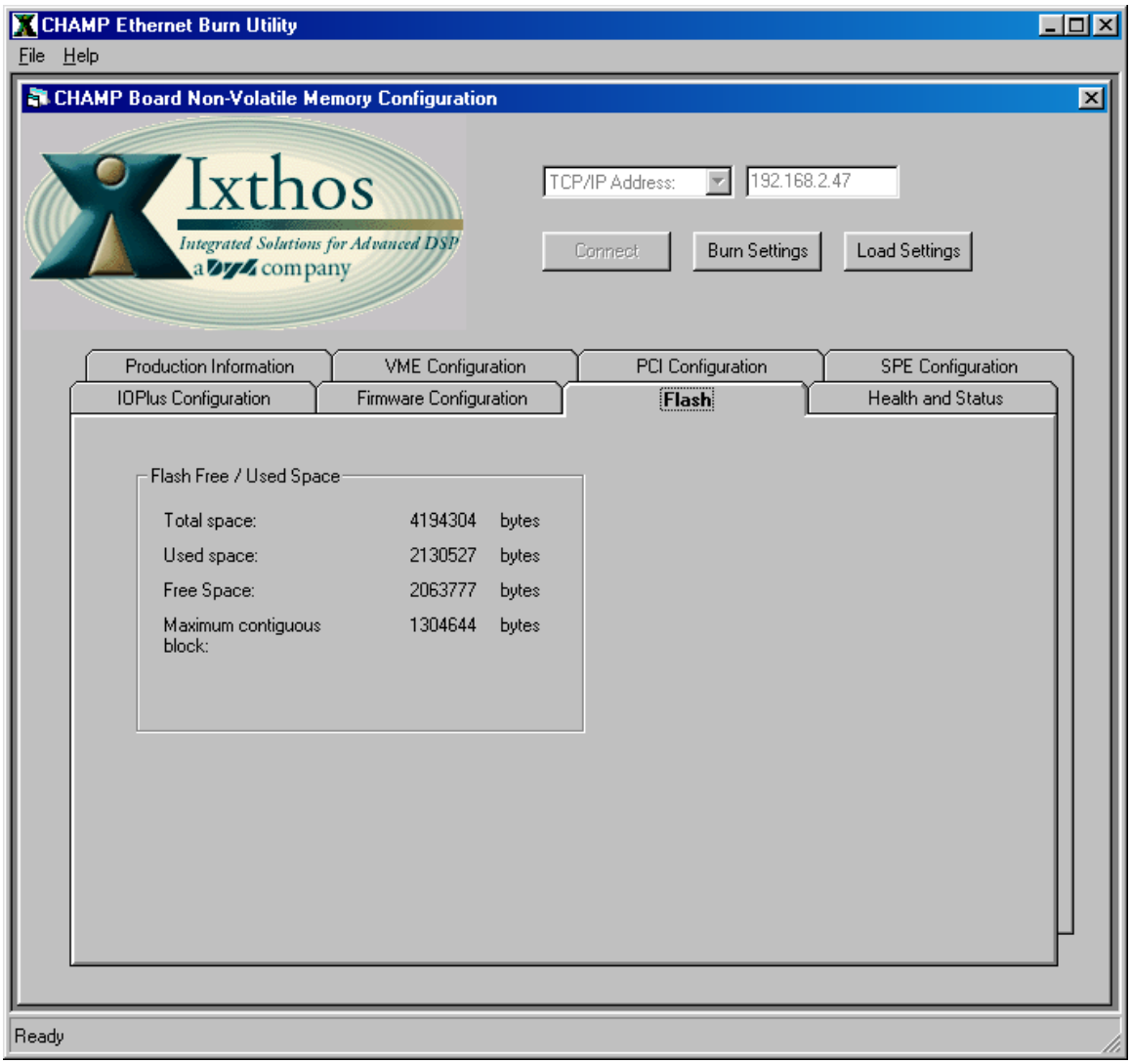


Figure 8.11 - Flash Page

8.9 Health and Status Page

The Health and Status page, shown in Figure 8.12, is not currently implemented. It will eventually display information describing the health and status of the CHAMP board.

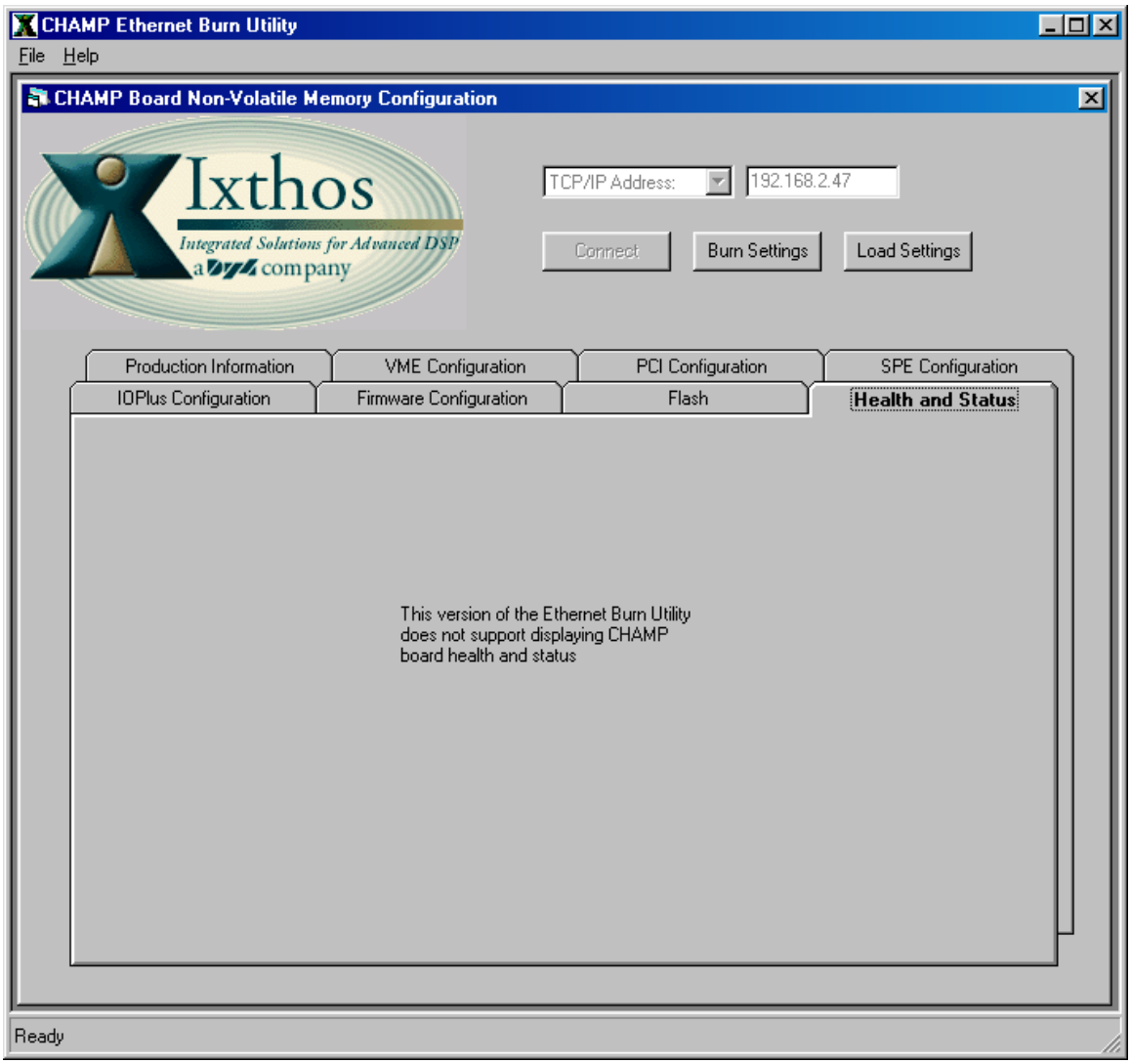


Figure 8.12 – Health and Status Page

8.10 Burning Flash using the Ethernet Burn Utility

After specifying the changes that you wish to make to the CHAMP board non-volatile memory configuration, you must burn these changes into flash. Changes made to the board configuration are burned into the flash by clicking the "Burn Settings" button, located in the upper right corner of the Burn Utility dialog box.

Depending on the changes that you are attempting to make to the CHAMP board, one or more of the following dialog boxes may appear.

If you are reburning SPE programs, you will see the dialog box shown in Figure 8.13.

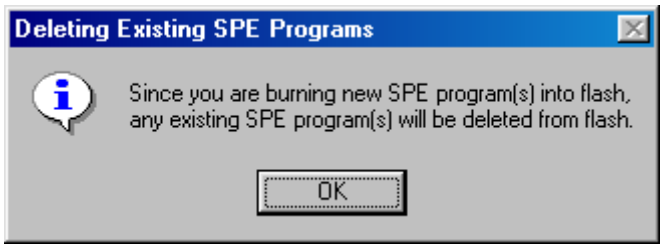


Figure 8.13 – Deleting Existing SPE Programs

This dialog box informs you that any SPE programs already in flash will be deleted before the new SPE files are burned. Existing SPE programs are deleted to avoid conflicts with the new programs. Note that this means that you must burn all SPE program(s) into flash during the same burn, since subsequent burns will erase any SPE programs already stored in flash.

If you attempt to change the CHAMP board Startup firmware, you will see the dialog box shown in Figure 8.14.

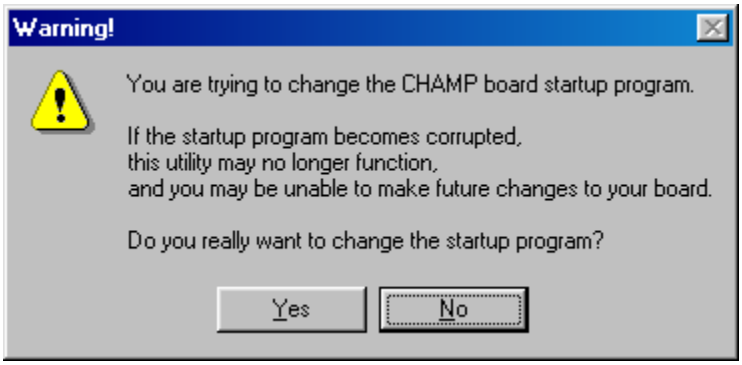


Figure 8.14 – Changing Startup Code

This warning reminds you of the danger of reburning the startup code. Please verify that you are burning the correct startup code file into the board.

If you attempt to change the VxWorks Boot ROM, you will see the dialog box shown in Figure 8.15.

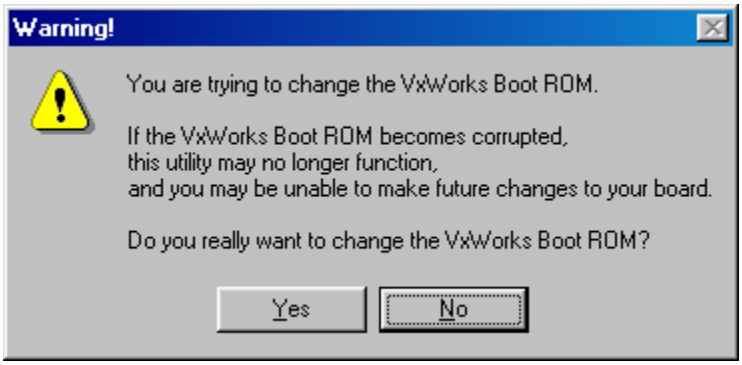


Figure 8.15 – Changing VxWorks Boot ROM

This warning reminds you of the danger of reburning the VxWorks Boot ROM. Please verify that you are burning the correct VxWorks Boot ROM file into the board.

If you attempt to disable VxWorks booting, you will see the dialog box shown in Figure 8.16.

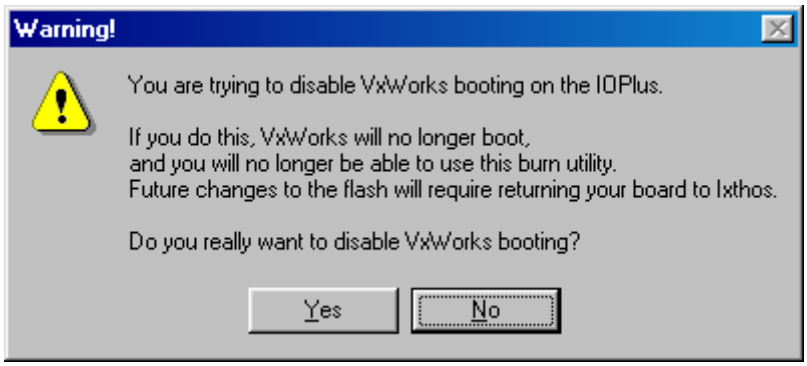


Figure 8.16 – Changing VxWorks Boot ROM

This warning reminds you that this is an irreversible process, and asks you to verify that you really want to do this.

If you attempt to burn a corrupted S-Record file, you may see the dialog box shown in Figure 8.17.

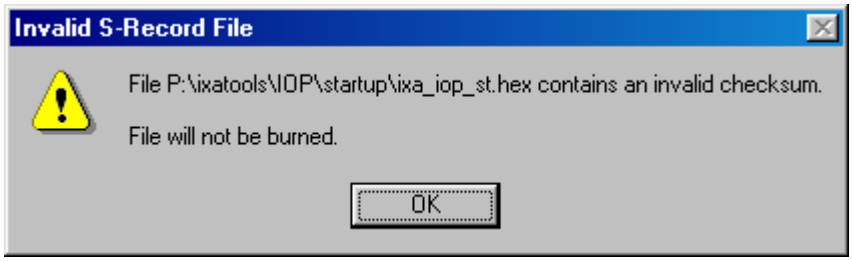


Figure 8.17 – Invalid Checksum

If you see this message, the file you are trying to burn is corrupted, and should be replaced. Contact Dy 4 Systems customer support for assistance.

While the board is being reburned, the Ethernet Burn Utility will display the status window shown in Figure 8.18.



Figure 8.18 – Burning Program

While burning, the status window will display which information in flash is currently being burned. The "spinner" on the right side of the status window will update, to indicate that the program is working. Note that patience is sometimes required; burning large SPE files can be slow (i.e. can take up to five minutes per file on a slow computer/network).

Figure 8.19 shows the burn utility screen while a burn is in progress. Note that both the status window, and the status bar at the bottom of the screen update to indicate the state of the burn operation. Note that the status bar conveys additional information (i.e. "loading" or "burning" file).

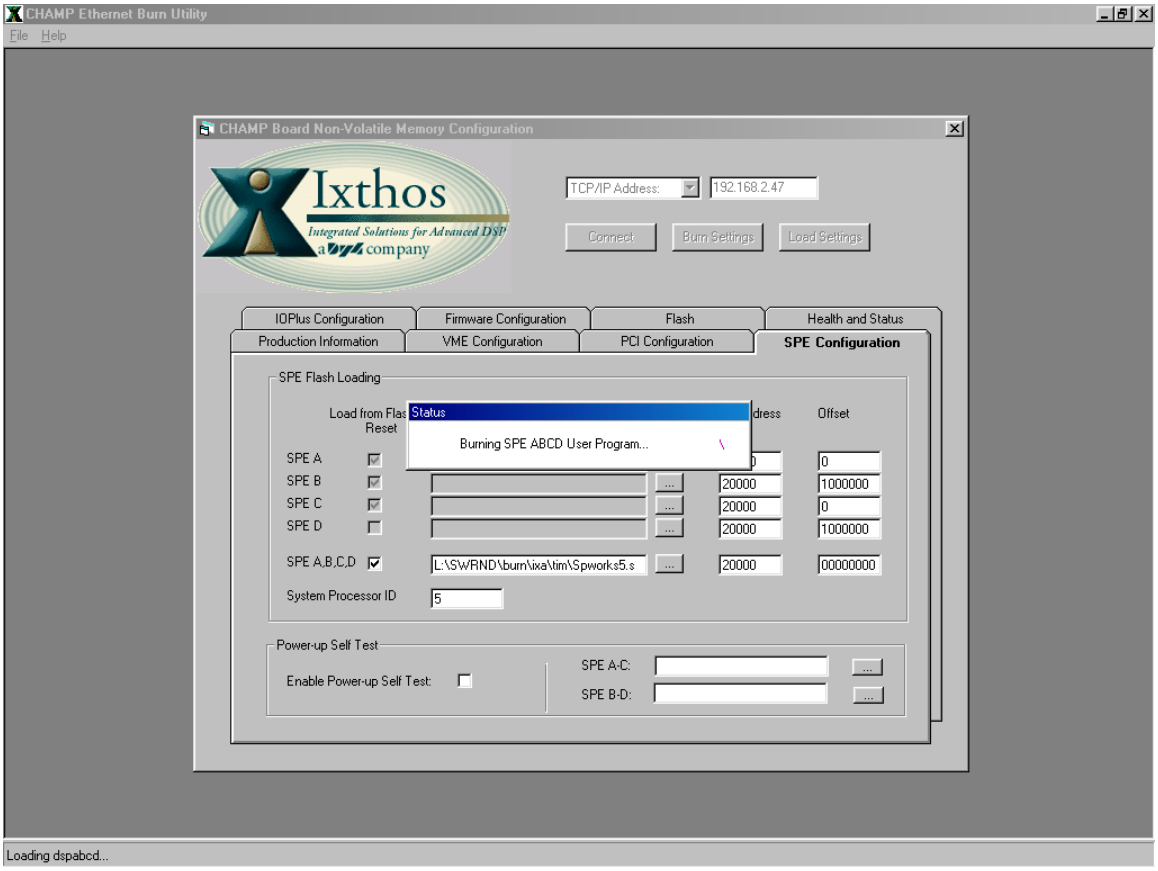


Figure 8.19 – Screen During Burn Operation

During a burn, you may see the dialog box shown in Figure 8.20, which indicates that an error occurred while burning flash.



Figure 8.20 – Error Writing to Flash

This dialog box typically indicates that the flash is full, although another type of error might have occurred. The flash can become full when burning large individual SPE files. If this occurs, consider loading a single executable into all SPEs, rather than individual executables into each SPE.

After a burn completes with errors, the dialog box shown in Figure 8.21 will be displayed.



Figure 8.21 – Burn Completed with Errors

If this dialog box is displayed, then one or more errors occurred during the burn operation. Most likely, some of the changes that you requested were made, while others were not. We recommend not resetting the CHAMP board in this situation. Instead, click "OK" to dismiss the dialog, then fix the problems that caused the errors, and then repeat the burn by clicking the "Burn Settings" button.

After a burn completes successfully, the dialog box shown in Figure 8.22 will be displayed.



Figure 8.22 – Burn Completed Successfully

For the changes to take effect, you must first close the Ethernet Burn Utility program, and then reset the CHAMP board. If you reset without closing the burn utility, the utility will think that it is still connected to the CHAMP board, even though the connection was broken by the reset. Unexpected behaviour will result from this situation.

8.11 The IXA Flash Burn Utility

The IXA Flash Burn Utility *ldflash* may be used to reburn flash on a board only in the following circumstances:

The Ethernet Burn Utility will not operate on your board.

You have a host computer in your VME bus chassis, with a working HOSTAPI port.

You have placed the CHAMP board into recovery mode by removing jumper JM5 and resetting the board (the red LED on the front panel will flash rapidly for a brief period after reset, indicating the board is in recovery mode).

The *ldflash* utility processes commands that are contained in a command file (which can be created or edited using any ASCII text editor). The syntax of the commands that can be placed in this file is described in Table 8.1. Note that command keywords are printed in **bold**, while parameters are printed in *italics*.

Table 8.1 - *ldflash* Commands

Command	Description
Attach *vme_base_address*	This should be the first command in your command file. It directs *ldflash* to target all subsequent commands to the board with the specified *vme_base_address*.
Erase	Erases all user-modifiable portions of the flash memory. Warning! This is a dangerous command, and should only be used at the direction of Dy 4 Systems customer support.
file *filename section_type*	The **file** command burns the specified S-Record format file named *filename* into flash. The *section_type* tells the board firmware how to use the file. A list of *section_type*s that have special meaning to the IOPlus firmware is provided below.
load *filename section_type*	The **load** command loads the specified S-Record format file directly into the SPEs (bypassing the flash); the SPEs immediately begin executing the loaded code. The *section_type* tells the board firmware where to load the file. The following *section_type*s are valid for this command: *dspa*, *dspb*, *dspc*, *dspd*, *dspab*, *dspac*, *dspad*, *dspbc*, *dspbd*, *dspcd*, *dspabc*, *dspabd*, *dspbcd*, *dspabcd*, *iopprog*.
Config *param_number value*	This command allows board configuration parameters to be set or altered. A list of configuration parameters is provided in a table below.
prod *param_number value*	This command is reserved for use by Dy 4 Systems. It allows production parameters to be set or altered. Note that production parameters are placed in the flash only when the board is initially built or when it is returned to Dy 4 Systems for service.
vxworks_nvram *config_string*	This command loads a VxWorks boot configuration string into flash memory. Note that the configuration string can contain spaces, but must be on a single line. Note also that the configuration string is typically changed using the VxWorks tools boot monitor, rather than with the *ldflash* utility.
; *comment text*	Comments can be placed after a command, or on a separate line. All text on the remainder of the line following a semicolon "**;**" character is interpreted as a comment and therefore ignored.

Flash Section Types with Special Meanings

Tables 8.2 - 8.4 list all flash section types that have special meaning. Note that there are three different classes of section: those that can be freely modified by the user, sections that should be modified by the user only at the direction of Dy 4 Systems customer support, and sections which can not be modified by the user (i.e. they can only be modified by Dy 4 Systems).

Table 8.2 - Flash Sections that can be Modified by Users

Section Type Name	Description
vxworks_bootrom	The "VxWorks Boot ROM" is executed after the startup code when the board is configured to boot VxWorks (i.e. when configuration parameter 24 is set to 1).
Vxworks_nvram	Contains the VxWorks boot string. This section is typically modified using the VxWorks boot monitor.
Gmemdata	The contents of this section are loaded into global memory on board reset. If a program was loaded into global memory, it can be optionally executed by setting configuration parameter 30.
Common_boot_code	The common boot code was designed to establish separate environments for each processor before the processors boot and run the main application.
Iopprog	This section type name can be used only with the **load** command, not with the **file** command. It loads a program into global memory (bypassing the flash), and immediately begins executing it.
Dspa or spea	Loads the program into SPE A and executes it.
Dspb or speb	Loads the program into SPE Band executes it.
Dspc or spec	Loads the program into SPE Cand executes it.
Dspd or spcd	Loads the program into SPE Dand executes it.
Dspab or speab	Loads the program into SPEs A and B and executes it.
Dspac or speac	Loads the program into SPEs A and C and executes it.
Dspad or spead	Loads the program into SPEs A and D and executes it.
Dspbc or spebc	Loads the program into SPEs B and C and executes it.
Dspbd or spebd	Loads the program into SPEs B and D and executes it.
Dspcd or specd	Loads the program into SPEs C and D and executes it.
Dspabc or speabc	Loads the program into SPEs A, B and C and executes it.
Dspabd or speabd	Loads the program into SPEs A, B and D and executes it.
Dspbcd or spebcd	Loads the program into SPEs B, C and D and executes it.
Dspabcd or spebcd	Loads the program into SPEs A, B, C and D and executes it.

Table 8.3 - Flash Sections that can be Modified by Users at the Direction of Dy 4 Systems

Section Type Name	Description
Xilinx	Xilinx firmware used when board is in normal operating mode.
Orca	Only used on IXC boards. Orca firmware used when board is in normal operating mode.
startup	Board initialization firmware that configures board on power up or reset when board is in normal operating mode.
runtime	Board firmware that implements IOPLus "SmartDMA" when VxWorks is not loaded into IOPlus.

Table 8.4 - Flash Sections not Modifiable by Users

Section Type Name	Description
recovery_xilinx	Xilinx firmware used when board is in recovery mode.
recovery_orca	Only used on IXC boards. Orca firmware used when board is in recovery mode.
iop_copy	Initial boot program which copies board startup firmware into flash and begins executing it.
Recovery	Board initialization firmware that configures board on power up or reset when board is operating in recovery mode.

Configuration Parameters

Table 8.5 lists all configuration parameters that can be modified by the user. The first column contains the *parameter_number*, while the second column describes the parameter.

Table 8.5 - Flash Configuration Parameters

Configuration Parameter Number	Description
0	Specifies how VME base address is set: 0: Board defaults to 0x80000000 1: Address set using JM1-JM4 2: Address stored in flash (using parameter 1) 3: Geographical address pins (not currently implemented) 4: VME64 auto ID (not implemented) 5: DY4 auto ID (not implemented)
1	VME base address of board (only used when parameter 0 is set to 2.
6	Maximum command channels used by IOPlus for processing commands (defaults to 6)
7	Maximum number of host processes that can be attached to IOPlus simultaneously (defaults to 4)
8	Enables loading of SPEs (using *dspxxxx or spexxxx* section type) and/or IOPlus/Global memory (using *gmemdata* section type). A 1 in the proper bit location enables loading while a 0 disables the load. Bit 0: Enables flash loading of IOPlus global memory Bit 1: Enables flash loading of SPE A Bit 2: Enables flash loading of SPE B Bit 3: Enables flash loading of SPE C Bit 4: Enables flash loading of SPE D
9	IXC only: Determines whether DSP internal memory is cache or memory. 0: DSP internal memory is memory 1: Internal memory is cache
12	Controls whether Universe II configuration registers are visible on VME bus 0: Registers are visible on VME 1: Registers are not visible on VME

Table 8.5 - Flash configuration parameters (cont.)

Configuration Parameter Number	Description
13	IXC only: Determines speed at which DSPs operate. 0: 118 MHz 1: 125 MHz 2: 133 MHz 3: 143 MHz 4: 154 MHz 5: 167 MHz 6: 182 MHz 7: 200 MHz
17	VME Bus request level 0: Bus request level 0 1: Bus request level 1 2: Bus request level 2 3: Bus request level 3
18	VME arbitration mode (only valid when board is VME SYSCON) 0: Round robin 1: Priority
19	VME release mode 0: Release when done 1: Release on request
20	VME Timeout (only valid when board is VME SYSCON). Not implemented.
21	Size (in bytes) of host command buffer (defaults to 0x2000)
22	Size (in bytes) of host response buffer (defaults to 0x2000)
24	Controls whether the VxWorks boot ROM, or the IOP Runtime code is loaded into the IOPlus and executed. 0: Load and execute Runtime code 1: Load and execute VxWorks boot ROM
25	Controls whether POST is loaded into SPEs and executed (not currently implemented). 0: POST is not loaded and executed 1: POST is loaded and executed
26	IXC only: Controls whether SBSRAM memory is run at half-speed or full speed. 1: SBSRAM runs at half speed 0: SBSRAM runs at full speed
27	Starting address in global memory of shared heap managed by IOP Runtime code
28	Ending address in global memory of shared heap managed by IOP Runtime code
29	Controls PMC latency timer settings, enabling of PMC BIST, and enabling of PMC bus mastering. The bits are defined as follows: 31-24: PMC 2 latency timer 23-16: PMC 1 latency timer 3: enable PMC 2 BIST 2: enable PMC 1 BIST 1: enable PMC 2 bus master 0: enable PMC 1 bus master

Table 8.5 - Flash configuration parameters(cont.)

Configuration Parameter Number	Description
30	If a *gmemdata* section is loaded into global memory (note that parameter 8 must be set to FFFFFFFF) and if this parameter is set to an address other than FFFFFFFF, then the IOPlus will begin executing code from global memory at the specified address upon power up or reboot.
31	IXC only: This parameter sets IXStar latency timers using the following bits: 15-8: IXStar CD latency timer 7-0: IXStar AB latency timer
32	This parameter sets the AB PCI-PCI Bridge latency timer using the following bits: 15-8: Secondary side latency timer 7-0: Primary side latency timer
33	This parameter sets the CD PCI-PCI Bridge latency timer using the following bits: 15-8: Secondary side latency timer 7-0: Primary side latency timer
34	This parameter sets the IOPlus PCI-PCI Bridge latency timer using the following bits: 7-0: Primary side latency timer
36	Unique system processor ID for assignment to IOPlus processor
40	Specifies starting offset in local SDRAM for SPE A application load from flash
41	Specifies starting offset in local SDRAM for SPE B application load from flash
42	Specifies starting offset in local SDRAM for SPE C application load from flash
43	Specifies starting offset in local SDRAM for SPE D application load from flash
44	Start address of SPE A application code as seen by CBC
45	Start address of SPE B application code as seen by CBC
46	Start address of SPE C application code as seen by CBC
47	Start address of SPE D application code as seen by CBC
64	Slot 1 VME base address, or board 1 VME base address (only used for VME auto ID or geographical VME addressing)
65	Slot or board 2 VME base address
66	Slot or board 3 VME base address
67	Slot or board 4 VME base address
.	…
95	Slot or board 32 VME base address

Sample *ldflash* Command File

The sample *ldflash* command file shown below performs the following common tasks:

Attaches to a board at VME base address 0x80000000

Burns an SPE program named *progac.hex* (stored in s-record format) into flash. The program will be loaded into SPEs A and C and begin executing upon board power up or reset.

Burns an SPE program named *progbd.hex* (stored in s-record format) into flash. The program will be loaded into SPEs B and D and begin executing upon board power up or reset.

Burns an IOPlus program named *iop.hex* (stored in s-record format) into flash. The program will be loaded into global memory upon board power up or reset.

Sets configuration parameter 8 to FFFFFFFF, which enables loading of the SPEs and global memory from flash (set this parameter to 0 to disable loading).

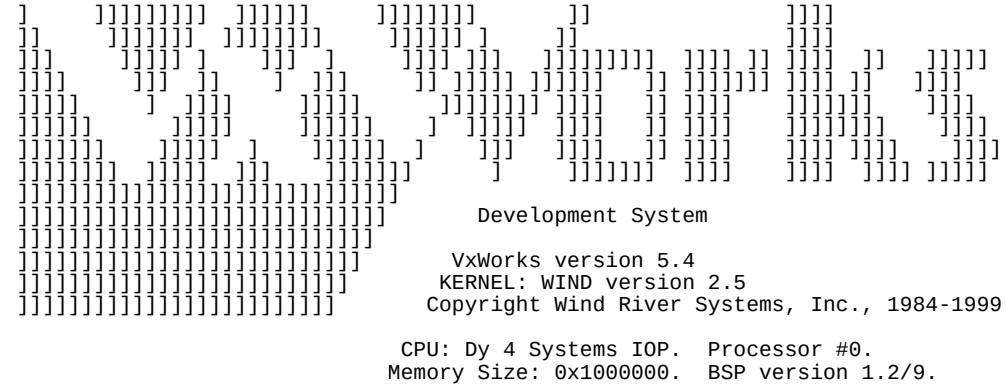
Sets configuration parameter 30 to global memory address 0x50000. After loading the global memory with the program stored in *iop.hex*, the IOPlus firmware will begin executing at global memory address 0x50000 (which is assumed to be the entry point for the program stored in *iop.hex*).

```
attach  80000000              ; Attach to board at VME address 0x80000000
file  progac.hex  dspac       ; "progac.hex" will be loaded into SPE A & C
file  progbd.hex  dspbd       ; "progbd.hex" will be loaded into SPE B & D
file  iop.hex     gmemdata    ; "iop.hex" loaded into global memory
config      8     FFFFFFFF    ;  Enable loading of SPEs and global memory
config      30    50000       ;  Run IOPlus program at global memory 50000
```

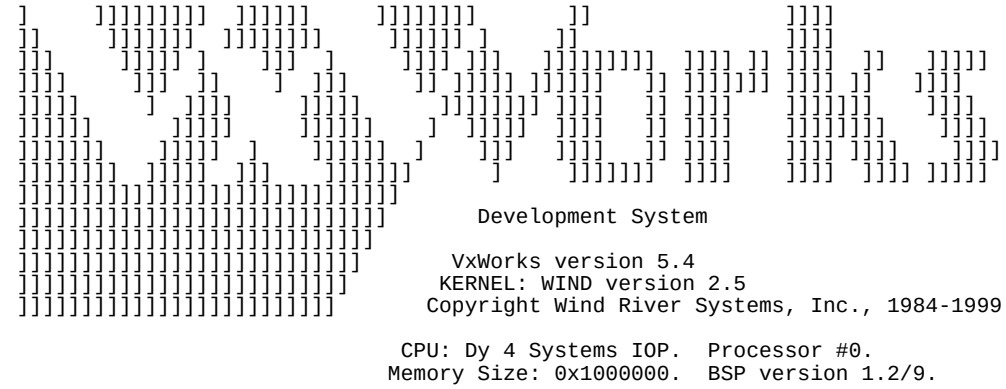
8.12 Flash Validation

Under certain circumstances, the flash memory on your CHAMP board may become corrupted. This could occur, for instance, if the board is accidentally reset or if the power is turned off while flash is being reburned. If your CHAMP board flash becomes corrupted, the board may not boot properly. CHAMP board flash memory contains redundant directory information as well as redundant boot code. This redundancy minimizes the cases where the CHAMP board becomes unbootable due to flash corruption. CRCs are calculated on each file in flash, and stored in the flash directories. These CRCs allow corrupted files to be detected.

When VxWorks loads on the IOPlus, the integrity of CHAMP board flash memory is validated. This check is performed immediately after the VxWorks banner is displayed. The following message should appear on the VxWorks console:



If flash validation passes, then no further action is required. However, if flash corruption is detected, a message similar to the following will be displayed:



```
------------------------------------------------

 Please contact Dy 4 Systems customer support for
 further assistance.


 Burntask() version 5 accepting connections on port 1001...
```

If a flash corruption warning is displayed, please contact Dy 4 Systems customer support for assistance in repairing the damage. Note that even though the board might appear to boot and operate properly, flash corruption should not be ignored. If further damage were to occur to a corrupted flash, the CHAMP board might be rendered unbootable. In most cases, flash recovery is quick and relatively painless. For instance, the CHAMP board in the example shown above has a corrupted primary directory. No errors are detected in the backup flash directory, indicating that it is valid. To repair the flash corruption, the secondary directory can simply be copied over the primary directory.

Several new commands have been added to the VxWorks BSP for the IOPlus to support flash corruption diagnosis and recovery. Two of these commands are described below.

The **validateFlash** command causes integrity checks to be performed on CHAMP board flash memory. Any detected errors are reported. The checks performed by this command are identical to the validation performed when VxWorks boots. Below is the output of the **valiateFlash** command on a CHAMP board that was accidentally reset during the burning of the SPE Boot ROM.

```
-> validateFlash

  Validating flash...        failed!


  +-----------------------------------------------------+
  |                                                     |
  | !!!  WARNING  WARNING  WARNING  WARNING  WARNING !!! |
  |                                                     |
  |      Your CHAMP board flash memory is corrupted.     |
  |        Please contact Dy 4 Systems customer support       |
  |                for further assistance.              |
  |                                                     |
  | !!!  WARNING  WARNING  WARNING  WARNING  WARNING !!! |
  |                                                     |
  +-----------------------------------------------------+



 ------------------------------------------------
     The primary flash directory has the following error(s):

         The following files had a checksum error in the primary directory:
          dspabcd

 ------------------------------------------------
     The secondary flash directory has the following error(s):

         The following files had a checksum error in the secondary directory:
          dspabcd

 ------------------------------------------------
```

```
  Please contact Dy 4 Systems customer support for
  further assistance.


value = 131074 = 0x20002
->
```

To repair the corrupted SPE Boot ROM, simply reburn it (no resets this time, please!).

The **printFlashDir** command displays the contents of the primary flash directory. This command shows where individual files are stored in flash, as well as their sizes and checksums. The output of this command assists Dy 4 Systems customer support in advising you on the best procedure for recovering a corrupted flash. The output of a typical **printFlashDir** is shown below (note the *BAADBAAD* checksum on the "*dspabcd*" file).

```
-> printFlashDir
  Checksum    Offset        Size    Name
  --------    ------        ----  ---------
  45cb0846    00000000         0  null
  3e459461    00000000     10000  xilinx
  154803cc    00010000     20000  orca
  5c342ecd    00030000      2000  config
  3f5277fb    00032000       400  vxworks_nvram
  54d36e0c    00032400      2758  common_boot_code
  ab951b56    00034fac      afa0  runtime
  baadbaad    0003ff4c     32538  dspabcd
  --------   1 empty entries -----
  0f17ef70    0007ffc8      d3ac  startup
  --------   22 empty entries -----
  b0fb310d    00100100      4000  iop_copy
  7d2d201c    00105000      a000  recovery
  75d13c04    0010f000      1000  prodparams
  3e459461    00110000     10000  recovery_xilinx
  154803cc    00120000     20000  recovery_orca
  171beebc    00140100     3ff00  vxworks_bootrom
  15372ac6    00180100     3ef00  recovery_bootrom
  ffffffff    001c0000      1000  bkup_directory
  ffffffff    001c1000      f000  bkup_dir_buffer
  --------   22 empty entries -----
  ffffffff    007ff000      1000  directory

  Total space:        8388608 (0x800000) bytes
  Total used space:   1368540 (0x14e1dc) bytes
  Total free space:   7020068 (0x6b1e24) bytes
  Maximum free block: 6483968 (0x62f000) bytes
```

Chapter 9: Host Software

9.1 Introduction

The IXAtools software includes a collection of functions, HostAPI, that permit an application to load, start, stop, and exchange data with SPEs on the IXA7. These functions are provided as "C" source code permitting this capability to be obtained over a broad range of host environments. Figure 9.1 shows the relationship of the HostAPI library to the application and the VMEbus. To use HostAPI requires a VMEbus interface driver that conforms to an Dy 4 Systems convention, shown in the figure as VMElib. Dy 4 Systems has implemented VMElibs for a number of systems and may be able to provide one for your configuration. Contact Dy 4 Systems to determine if a VMElib for your configuration already exists.

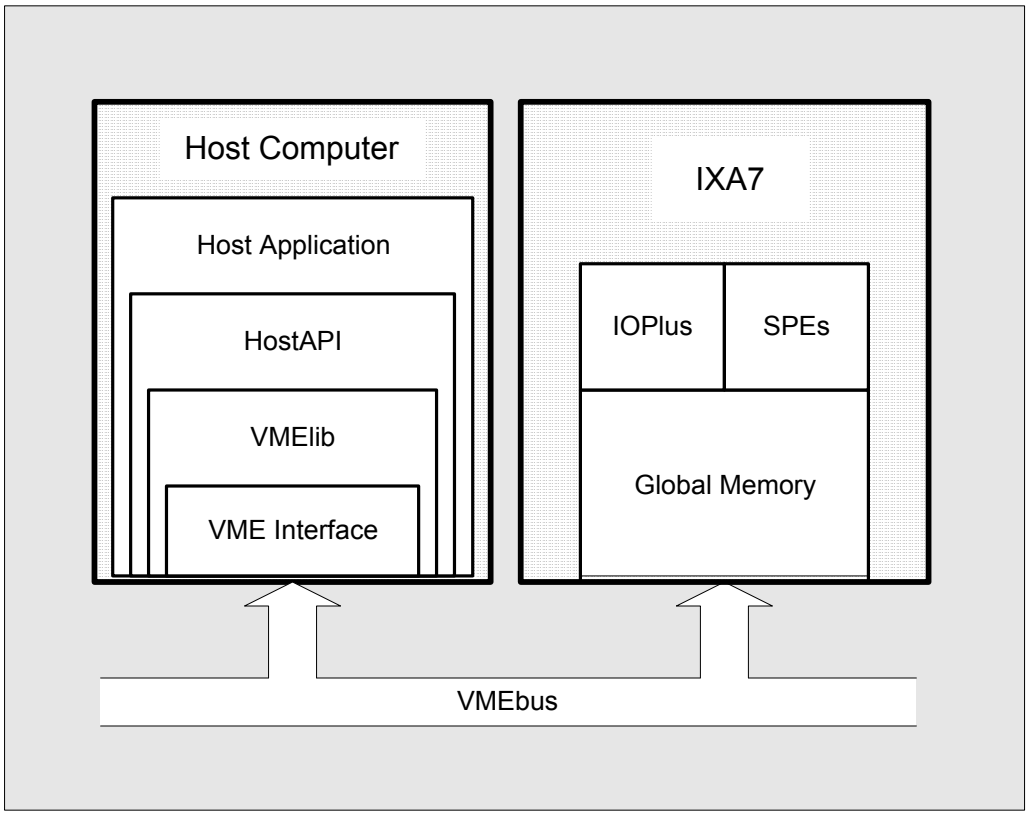


Figure 9.1 - HostAPI Relationship Diagram

If a VMElib doesn't exist you will need to develop one for your host. This chapter will detail what needs to be done. The effort is not that major but it will require you to research the documentation that came with your host to find out what software driver capability is already available.

9.2 Porting HostAPI

HostAPI, communicates with the IXA7 using the VMEbus. Dy 4 Systems has ported HostAPI to a number of VME based hosts. These ports are not only hardware dependent, as to the manufacturer and model number of host processor board, but also dependent on the version of operating system used by the host. Since hardware and software products change at a rapid rate, your system design may require a configuration of VME host for which Dy 4 Systems has not implemented a port. The following paragraphs address porting to a new host system.

Porting HostAPI consists of three basic steps:

1. Developing a VMEbus interface driver.
2. Constructing a set of functions (VMElib) to interface the VMEbus interface driver to the HostAPI software.
3. Compiling, linking, and testing the HostAPI software with the VMElib and VMEbus interface driver software.

First contact Dy 4 Systems to determine if the configuration needed by your application is currently available. One of the existing configurations may be suitable to your application, or can be easily modified.

If you must port to a new configuration, you must consider the following:

Does the hardware come with a VMEbus interface driver for the required operating system?

Does the operating system come with a VMEbus interface driver for the required hardware?

If the answer to either of these questions is yes, then you may have been spared the task of developing a VMEbus interface driver for your application. You should still review the next section to determine if the VMEbus interface driver is capable of satisfying the requirements of a HostAPI port.

VMEbus Interface Driver Requirements

A VMEbus interface driver for HostAPI needs only support basic VMEbus read/write capabilities. The host software does not require that VME interrupts be issued or serviced by the driver. However, the VMEbus interface driver must support A32/D32 data accesses to VME address space. If D32 is not obtainable, the driver will have to perform a translation to communicate with the IXA boards. Contact Dy 4 Systems for more information as to what this translation involves.

The driver should support any required byte swapping between the host and SPE word formats. If not, you must implement byte swapping as part of the development of the

VMElib routines. Likewise, if the driver does not support more than one open device at a time, the VMElib routines will have to handle that aspect.

VMElib Interface Routines

For software portability and maintainability, Dy 4 Systems has separated the VMEbus interface driver software from the HostAPI software. The interface between the HostAPI and the VMEbus interface driver is a small collection of functions grouped together and referred to in this document as VMElib. This library is implementation specific and "glues" the host software calls to the driver specific calls. Therefore, you must develop a VMElib library for all HostAPI ports.

VMElib consists of five function calls. Four of these functions, *vme_open, vme_write, vme_read*, and *vme_close,* are used to adapt the VMEbus interface driver specific calls to the rest of the host software. An additional function, *hostlib_get_info,* is an optional convenience function to permit driver identification for your application. ANSI C function prototypes and constants defining these functions can be found in the file *ixhost.h*. The definitions of these functions must therefore be maintained to simplify any port. The operation of these functions and their required arguments are detailed in the following sections. Please consult your distribution diskette to obtain the latest copy of this source file, *ixhost.h*.

VME_IO Data Structure

The primary data structure used by VMElib is the VME_IO structure. This structure is allocated and initialized by *vme_open*, and referenced by address by all other function calls. The host software does not reference any fields within this data structure. The use of fields inside this data structure is optional; your device driver port may not require the use of all fields.

```
typedef struct vme_io
{
       int              unix_fd;          /* handle from "open" call      */
       char             addr_mode;        /* address mode requested       */
       char             transfer_mode;    /* byte, word, or longword      */
       char             swap_mode;        /* big_endian or little_endian  */
       unsigned long    start_addr;       /* starting VME address          */
       unsigned long    size;             /* size (bytes) of VME window   */
       unsigned long    mapstart;         /* address of  window           */
       int              maplen;           /* length of window             */
       unsigned long    virtual_ptr;      /* virtual addr of vme window   */
       char             mem_seg_name[16]; /* memory segment name          */
} VME_IO;
```

The **unix_fd** field is provided for those implementations that must make an "open" system call and remember a file handle.
addr_mode, **transfer_mode**, **swap_mode**, **start_addr**, and **size** correspond to input parameters for the *vme_open* function call. These fields are set by *vme_open* for subsequent reference. Definitions for setting these fields are contained in *ixhost.h.*
The **mapstart**, **maplen**, and **virtual_ptr** fields are provided for those implementations that must map VME address space into a logical address space within the process.

mem_seg_name is provided for those systems that manage VME address space using named, shared memory segments.

A VMElib implementation may not require all fields within this data structure. Unused fields need not be initialized or referenced.

Initialization Function (vme_open)

The initialization function is called *vme_open.* It sets fields within the VME_IO data structure and performs any device driver initialization required by the host operating system. In UNIX operating systems, this function consults the environment variables to obtain the name of the device driver, opens the device driver, and verifies that the addressing mode is valid. This function saves the file handle returned by the open call in the VME_IO data structure.

Calling Sequence:

```
void *vme_open(char              admod,
               char              mode,
               char              swap,
               unsigned long     start,
               unsigned long     size,
               int               *status );
```

Inputs:

admod is the address modifier. This must be one of the following values:

0x08 + S + D for A32 space,

0x30 + S + D for A24 space,

0x20 + S + D for A16 space,

where S = 4 for privileged mode, 0 for non-privileged mode, and D = 2 for program memory access, 1 for data memory access.

At a minimum, your device driver must support A32 data accesses to VME address space. Because Dy 4 Systems boards do not differentiate between privileged mode accesses and non-privileged accesses, or program memory or data accesses, your device driver may use the most convenient mode (e.g., privileged data memory) to satisfy the initialization request.

mode indicates whether subsequent transfers will take place as byte, word, or longword transfers. Mode must have one of the following values: **byte_mode**, **word_mode**, or **long_word_mode** as defined in *ixhost.h.* The HostAPI software uses **long_word_mode** as its primary transfer format.

swap sets the ordering of bytes within words and longwords for the VME address region being initiated. The value of swap can be either **big_endian** or **little_endian** as defined in *ixhost.h*.

start_addr is the starting VME bus address requested by the host software.

size is the length, in bytes, of the window requested by the host software.

Processing:

The *vme_open* function should validate the input parameters, confirming that the addressing mode, starting address, size, and mode are supported by your system.

The *vme_open* call allocates and initializes the VME_IO structure **vme_io** as follows:

```
vme_io-> unix_fd      = (set to what driver requires
                           to id device open);
vme_io->addr_mode     = admod;
vme_io->transfer_mode = mode;
vme_io->size          = size;
vme_io->swap_mode     = swap;
vme_io->start_addr    = start_addr;
vme_io->virtual_ptr   = ( driver definable ) ;
vme_io->mapstart      = ( driver definable );
vme_io->maplen        = ( driver definable );
```

Outputs:

The *vme_open* function returns a pointer to the VME_IO structure that is used when calling the other VMElib functions. The *vme_open* function also returns a status through the ***status** parameter. Zero indicates success. Other return values are defined in *ixhost.h*:

```
1 = unsupported_transfer_mode
2 = invalid_addressing_mode
3 = addressing_mode_not_supported
4 = invalid_transfer_mode
5 = invalid_swap_mode
```

Notes:

Host software will call *vme_open* for each board command issued. The starting address will correspond to the base address of the board, plus an offset corresponding to root controller memory.

Some systems may require no processing within *vme_open*. For example, a single board computer that provides full time, consistently mapped access to VME address space need not perform any special initialization in the *vme_open* function.

Read Function (vme_read)

The read function is called *vme_read*. It copies data from the specified address into local memory. The copy is performed using the address mode, transfer type, and byte-swapping options obtained from the VME_IO structure.

Calling Sequence:

```
void vme_read(  void              *vme_io,
                void              *dest_data,
                unsigned long      count,
                unsigned long      vme_get_addr,
                int               *status);
```

Inputs:

***vme_io** is a pointer to the VME_IO structure that was previously initialized by *vme_open*.

***dest_data** is the address where data is to be copied.

count is the number of bytes, words, or longwords to be transferred. The selection of byte, word, or longword is based upon **vme_io->transfer_mode**.

vme_get_addr is the VME address from which data is to be copied.

Processing:

Using parameters stored in **vme_io**, this function copies data from **vme_get_addr** to **dest_data**. This function may validate the starting address and length, and may detect bus errors. The starting address and length fields can be checked for validity, although such error checking is completely optional.

Outputs:

This function returns status through the ***status** parameter. Zero indicates success. Other return values are as defined in *ixhost.h*:

```
12 = buserr_detected
13 = address_out_of_range
14 = transfer_too_large
```

Notes:

Many UNIX systems report bus errors through signals (such as segmentation violation). In these systems, the VME read and write functions need not detect bus errors.

Many systems do not support unaligned transfers. In such systems, transferring long word data to or from an address that is not a multiple of 4 will cause a bus error. Similarly, transferring word data to or from an address will cause a bus error. Your driver software may detect this condition and return an error status.

Write Function (vme_write)

The write function is called *vme_write.* It copies data from local memory to the specified VME address. The copy is performed using the address mode, transfer type, and byte-swapping options obtained from the VME_IO structure.

Calling Sequence:

```
void vme_write( void              *vme_io,
                void              *src_data,
                unsigned long     count,
                unsigned long     vme_put_addr,
                int               *status);
```

Inputs:

vme_io is a pointer to the VME_IO structure that was previously initialized by *vme_open*.

***src_data** is the address from which data will be copied.

count is the number of bytes, words, or longwords to be transferred. The selection of byte, word, or longword is based upon **vme_io->transfer_mode**.

vme_put_addr is the VME address to which data are to be copied.

Processing:

Using parameters stored in **vme_io**, this function copies data from **src_data** to **vme_put_addr**. This function may validate the starting address and length, and may detect for bus errors. The starting address and length fields can be checked for reasonableness, although such error checking is completely optional.

Outputs:

This function returns status through the ***status** parameter. Zero indicates success. Other return values are as defined in *ixhost.h*:

```
12 = buserr_detected
13 = address_out_of_range
14 = transfer_too_large
```

Notes:

See notes for *vme_read* regarding bus error detection and unaligned transfers.

Close Function (vme_close)

The close function is called *vme_close.* This restores whatever resources were allocated by *vme_open*.

Calling Sequence:

```
void vme_close ( void      *vme_io);
```

Inputs:

vme_io is a pointer to the VME_IO structure that was previously initialized by *vme_open.*

Processing:

Using parameters stored in **vme_io**, this function frees up resources that were allocated by *vme_open*. Depending upon the requirements of your system's VME device driver interface, this function may have to unmap memory, deallocate shared memory segments, close the device driver file **vme_io->unix_fd**, or perhaps do nothing at all.

Outputs:

None.

Identify Driver (hostlib_get_info).

The identify driver function is called *hostlib_get_info.* This function is available for host software to identify the VMElib implementation. This function is not required by HostAPI.

Calling Sequence:

```
void hostlib_get_info(char **driver_name,
                      char **driver_version );
```

Inputs:

****driver_name** location for the address of the **driver_name** character string

****driver_version** location for the address of the **driver_version** character string

Processing:

This function defines the two character strings and returns the addresses to them. The contents of the character strings are for informational use only, no computations are based upon the contents of this field. These strings can be of any length, and must be null terminated. For the string contents to remain in memory upon return, the strings must be declared as static.

You should update these strings as you update your driver software. This will make it much easier to confirm that your version of host software has been linked with the most recent device driver library.

Outputs:

****driver_name** location of the address of the **driver_name** character string

****driver_version** location of the address of the **driver_version** character string

Testing

Before linking a VME device driver with the HostAPI library software, the VME device driver should be thoroughly tested. You can test by using a driver program which calls the open, close, read and write routines, writing and reading known patterns to specified addresses. A VMEbus analyzer can be used to confirm proper address modifier and address mode values.

Test Equipment.

Testing will require a host chassis, the VME host computer, the test software, test driver software, and a slave VME card.

A software test harness should be written that accesses the slave card from the host. The test harness should perform *vme_open* requesting A32 access, and transfer data words to and from the slave memory card. The following test algorithm is suggested:

```
begin

call vme_open

initialize a test array of 32 bit values containing
       vme_address, vme_address+1, etc

call vme_write to copy the array to the slave board

call vme_read to read the slave board into a different array

verify that the array contains vme_address, vme_address+1, etc.

call vme_close

end
```

This algorithm tests basic input and output. Array lengths of at least 8k long words make a sufficient test. A VMEbus analyzer can be used to confirm the address, length, and address modifier bits for all read and write operations. If no VMEbus analyzer is available, then you should have a means for dumping the contents of the slave board to make sure that the test pattern was successfully written to the slave board.

The following test algorithm is recommended to test the ability for the driver to have several VME regions opened concurrently. This capability is important; this is the way that the host library will manage more than one board at the same time:

```
#define NVME 8

define an array of VME_IO vmes[NVMES]

begin

       for n = 0;  while n < NVME
       Call vme_open, using vmes[n, and a vme base address plus 8k * n
              n = n + 1
       end_for


       for n = 0; while n < NVME
              Initialize a test array of 2k longwords to have values of
                     base address + 8k*n + 0, +1, +2...
              call vme_write using vmes[n]
       end_for


       for n = 0; while n < NVME
              call vme_read using vmes[n], reading from
                     base_address + 8k*n +0, +1, +2...

              Verify the contents of the array

              call vme_close using vmes[n]
       end_ for

end_begin
```

9.3 HostAPI Functions

The following are the function definitions for the HostAPI library. These functions are provided in both source code and link-able object format. Note: link-able object format is supplied for supported host platforms, for other platforms you must build the HostAPI library from source code. Source code for HostAPI is always provided with IXAtools to accommodate potential differences with C compilers and to allow flexibility with developer's applications. Where as the modification of the HostAPI source code is permitted, it is not recommended. Changing the HostAPI library will make the adoption of future updates of IXAtools difficult and also complicate the customer support activity.

Using the HostAPI library in an application is relatively straightforward. Each IXA7 board is opened by calling *host_board_open*, has programs loaded and started with *host_load_program*, exchanges information using *host_memory_read* and *host_memory_write*, and at the conclusion each board gets closed with *host_board_close*. A number of useful utility functions are also provided. The main goal of HostAPI is to provide a very simple effective mechanism for the host and IXA7 to interact. Elaborate abstractions have been purposely avoided.

host_board_close

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS host_board_close( void *board_pointer );
```

DESCRIPTION:

host_board_close detaches from an IXA which was previously opened by *host_board_open*. If the board was not previously opened, an error is returned.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.

NOTES:

An opened board should be closed using *host_board_close* in order to free any resources that may have been allocated by *host_board_open*.

host_board_open

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_board_open( unsigned long   vme_address,
                                  void          **board_pointer );
```

DESCRIPTION:

host_board_open attaches to an IXA board with the specified VME address. It returns a board pointer, which must be passed to all other functions that access the board (the board pointer is similar to a file pointer). If no board is found at this VME address, an error is returned and the board pointer is set to NULL. A board must be opened using this function before it can be accessed by any other HostAPI functions.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_ERROR_OPENING_BOARD: no board was found at the specified VME address.

NOTES:

A board must be opened using this function before it can be accessed by any other HostAPI functions.

host_board_reset

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_board_reset( void            *board_pointer ,
                                   HOST_RESET_TYPE  type );
```

DESCRIPTION:

host_board_reset performs a hardware reset on a board. It is equivalent to pressing the front panel reset button.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.

NOTES:

HOST_RESET_TYPE can be:	HOST_RESET_VME_BUS, HOST_RESET_BOARD, HOST_RESET_PCI_BUS

host_board_status

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_board_status( void              *board_pointer,
                                    HOST_BOARD_STATUS *status );
```

DESCRIPTION:

host_board_status returns the status of the IXA7. The board determines its status by performing a power-up self-test (POST) whenever the board is reset.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.
HOST_DEVICE_ERROR: Check contents of *status*

NOTES:

HOST_BOARD_STATUS return codes:
- HOST_ERR_PROGRAMMING_XILINX
- HOST_ERR_CD_BRIDGE
- HOST_ERR_AB_BRIDGE
- HOST_ERR_TOP_BRIDGE
- HOST_ERR_CD_CLUSTER_MEMORY
- HOST_ERR_FLASH_MEMORY
- HOST_ERR_GLOBAL_MEMORY
- HOST_ERR_AB_CLUSTER_MEMORY
- HOST_ERR_UNIVERSE_II
- HOST_ERR_PCI_BUS

host_close_flash_params

CALLING SEQUENCE:

```
#include "_host_api.h"

HOST_API_STATUS  host_close_flash_params( void *param_ptr );
```

DESCRIPTION:

This function should only be called when the flash utility can not be used to update flash parameters. *host_close_flash_params* flushes the parameter changes which have been made by calls to *host_write_config_param* to non-volatile flash memory. Note that flash memory parameters are used by Dy 4 Systems firmware (startup and runtime code). The user should exercise caution when modifying these parameters.

RETURN STATUS:

HOST_NO_ERROR: flash parameter channel opened successfully.

NOTES:

Caution should be used when modifying flash parameters.
param_ptr must have been previously allocated by *host_open_flash_params*.

host_free

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_free(   void              *board_pointer,
                              void              *ptr, );
```

DESCRIPTION:

host_free releases memory which has previously been allocated by *host_malloc*. The pointer *ptr* refers to the PCI memory address returned by *host_malloc*.

RETURN STATUS:

HOST_NO_ERROR:	board opened successfully.
HOST_INVALID_BOARD_POINTER:	board pointer is invalid.

NOTES:

Be sure to pass this function the PCI address of the allocated memory, not the VME address of the allocated memory.

host_get_board_type

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_get_board_type( void            *board_pointer,
                                      HOST_BOARD_TYPE *board_type );
```

DESCRIPTION:

host_get_board_type returns an identifier which specifies the type of CHAMP board on which the code is running. Currently, the response types are:

HOST_IXC_BOARD,
HOST_IXA_BOARD

RETURN STATUS:

HOST_NO_ERROR:	success.
HOST_INVALID_BOARD_POINTER:	board pointer is invalid.

NOTES:

This function is useful only when mixing IXC and IXA boards.

host_load_flash

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_load_flash( unsigned long vme_address,
                                  char          *filename,
                                  char          *section_name );
```

DESCRIPTION:

host_load_flash burns a program or file in s-record format into the IXA7 board non-volatile flash memory. This routine can be used when burning files into flash using the flash utility is not desired.

The following section_names have special meaning to the IOPlus software, and should be used with caution:

HOSTAPI name	Text name	Description	Dy 4 Systems Reserved
HOST_NAME_VXWORKS_ BOOTROM	vxworks_bootrom	VxWorks boot ROM code, which is executed after the startup code when vxWorks booting is enabled	No
HOST_NAME_SPE_A	spea	Program loaded into SPE A on board reset	No
HOST_NAME_SPE_B	speb	Program loaded into SPE B on board reset	No
HOST_NAME_SPE_C	spec	Program loaded into SPE C on board reset	No
HOST_NAME_SPE_D	sped	Program loaded into SPE D on board reset	No
HOST_NAME_SPE_AB	speab	Program loaded into SPEs A & B on board reset	No
HOST_NAME_SPE_AC	speac	Program loaded into SPEs A & C on board reset	No
HOST_NAME_SPE_AD	spead	Program loaded into SPEs A & D on board reset	No
HOST_NAME_SPE_BC	spebc	Program loaded into SPEs B & C on board reset	No
HOST_NAME_SPE_BD	spebd	Program loaded into SPEs B & D on board reset	No

HOSTAPI name	Text name	Description	Dy 4 Systems Reserved
HOST_NAME_SPE_CD	specd	Program loaded into SPEs C and D on board reset	No
HOST_NAME_SPE_ABC	speabc	Program loaded into SPEs A, B and C on board reset	No
HOST_NAME_SPE_ABD	speabd	Program loaded into SPEs A, B, and D on board reset	No
HOST_NAME_SPE_BCD	spebcd	Program loaded into SPEs B, C, and D on board reset	No
HOST_NAME_SPE_ABCD	speabcd	Program loaded into SPEs A, B, C, and D on board reset	No
HOST_NAME_XILINX	xilinx	File used to program on-board Xilinx hardware	Yes
HOST_NAME_COPY	iop_copy	Initial boot code which loads startup program from flash into memory	Yes
HOST_NAME_STARTUP	startup	Startup code which initializes board resources	Yes
HOST_NAME_RECOVERY	recovery	Recovery code which allows flash to be reburned	Yes
HOST_NAME_RUNTIME	runtime	I0Plus runtime code	Yes

RETURN STATUS:

HOST_NO_ERROR: burn completed successfully.
HOST_BURN_ERROR: Error writing to flash.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.
HOST_INVALID_FILE_FORMAT: Specified file is not a valid S-record file.
HOST_WRONG_MODE: board must be in recovery mode to burn the flash.

NOTES:

1. S-Record format is used to simplify loading various processor types.
2. A list of valid section names is located in *host_api.h.*
3. This routine requires the board to be in recovery mode.
4. Different programs can be loaded into SPEs from flash in different combinations (i.e. both “spead” and “spebc” can be in flash simultaneously); however, two programs cannot be loaded into the same SPE (i.e. “spead” and “specd” is *not* valid).

host_load_program

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_load_program( void              *board_pointer,
                                    char              *filename,
                                    HOST_PROCESSOR_ID  processor_id );
```

DESCRIPTION:

host_load_program loads a program in s-record format into the specified processor. The program starts executing immediately after it has been loaded.
Where:

HOST_SPE_A:	8
HOST_SPE_B:	12
HOST_SPE_C:	16
HOST_SPE_D:	20

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_FILE_NOT_FOUND: problem with filename.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.
HOST_INVALID_FILE_FORMAT: Specified file is not a valid S-record file.

NOTES:

1. S-Record format is used to simplify loading various processor types.
2. A list of valid processor Ids is located in *host_api.h*
3. Processor Ids can be OR-ed together to load multiple processors with the same executable.
4. This routine requires the IOPlus Runtime code to be executing.

host_malloc

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_malloc( void              *board_pointer,
                              void              **ptr,
                              unsigned long     num_bytes );
```

DESCRIPTION:

host_malloc allocates the specified number of bytes from the shared global memory heap, and returns the PCI address of the allocated global memory. This PCI address must be converted to the VME address of the allocated region before accessing the memory through the VME bus. The PCI address can be converted to a VME address using the following formula:

VME_Address = VME_base_address + 0x01000000 – 0x40000 + PCI_Address

RETURN STATUS:

HOST_NO_ERROR:	success.
HOST_INVALID_BOARD_POINTER:	board pointer is invalid.
HOST_NO_RESOURCES_AVAILABLE:	Insufficient memory.

NOTES:

This function can be used to allocate memory which is shared between a VME host and the DSPs.

host_map_resource

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_map_resource( void               *board_pointer,
                                    HOST_RESOURCE_TYPE resource,
                                    unsigned long      vme_address,
                                    unsigned long      length_in_bytes,
                                    HOST_ADDRESS_SPACE address_space,
                                    void               **resource_ptr );
```

DESCRIPTION:

host_map_resource maps an IXA7 board resource onto the VMEbus, so that the board resource can be accessed from other boards on the VMEbus. The type of resource, VME address at which the resource should be mapped, the length (in bytes) of the VME address window to map, and the VME address space, must all be specified. The function returns a resource pointer, which is used in calls to *host_unmap_resource* (which unmaps the resource when the mapping is no longer needed). A maximum of four resource mappings per IXA7 board can be created.

RETURN STATUS:

HOST_NO_ERROR:	success.
HOST_INVALID_BOARD_POINTER:	board pointer is invalid.
HOST_NO_RESOURCES_AVAILABLE:	More than four resource mappings are active

NOTES:

Valid resources are:

HOST_RES_GLOBAL_MEMORY	= 2
HOST_RES_GLOBAL_USER_MEMORY	= 3
HOST_RES_PMC_AB	= 4
HOST_RES_PMC_CD	= 5
HOST_RES_SPE_MEMORY	= 8
HOST_RES_SPEA_SDRAM	= 10
HOST_RES_SPEB_MEMORY	= 12
HOST_RES_SPEB_SDRAM	= 14
HOST_RES_SPEC_MEMORY	= 16
HOST_RES_SPEC_SDRAM	= 18
HOST_RES_SPED_MEMORY	= 20

HOST_RES_SPED_SDRAM = 22
HOST_RES_SPEAB_SDRAM = 37
HOST_RES_SPECD_SDRAM = 38

Valid VME address spaces are:

HOST_VME_A16 = 0x00000
HOST_VME_A24 = 0x10000
HOST_VME_A32 = 0x20000

host_memory_read

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_memory_read( void                *board_pointer,
                                   void                *buffer,
                                   HOST_RESOURCE_TYPE   res_type,
                                   unsigned long        offset,
                                   unsigned long        num_words );
```

DESCRIPTION:

host_memory_read copies memory from the specified on-board resource to a memory buffer on the host. An offset within the on-board resource may be specified. The number of 32-bit words to read must also be specified.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.
HOST_INVALID_RESOURCE: invalid resource type.

NOTES:

The following HOST_RESOURCE_TYPEs are available:

HOST_RES_GLOBAL_MEMORY
HOST_RES_GLOBAL_USER_MEMORY
HOST_RES_PMC_AB
HOST_RES_PMC_CD
HOST_RES_SPEA_MEMORY
HOST_RES_SPEA_SDRAM
HOST_RES_SPEB_MEMORY
HOST_RES_SPEB_SDRAM
HOST_RES_SPEC_MEMORY
HOST_RES_SPEC_SDRAM
HOST_RES_SPED_MEMORY
HOST_RES_SPED_SDRAM
HOST_RES_SPEAB_SDRAM
HOST_RES_SPECD_SDRAM

host_memory_write

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_memory_write( void              *board_pointer,
                                    void              *buffer,
                                    HOST_RESOURCE_TYPE res_type,
                                    unsigned long      offset,
                                    unsigned long      num_words );
```

DESCRIPTION:

host_memory_write copies from a memory buffer on the host to the specified on-board resource. An offset within the on-board resource may be specified. The number of 32-bit words to write must also be specified.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.
HOST_INVALID_RESOURCE: invalid resource type

NOTES:

See *host_memory_read* for a list of HOST_RESOURCE_TYPEs.

host_open_flash_params

CALLING SEQUENCE:

```
#include "_host_api.h"

HOST_API_STATUS  host_open_flash_params( unsigned long vme_address,
                                         void          **param_ptr );
```

DESCRIPTION:

This function should only be called when the flash utility cannot be used to update flash parameters. *host_open_flash_params* opens a channel which is used to add or update parameters in the non-volatile flash memory. Note that flash memory parameters are used by Dy 4 Systems firmware (startup and runtime code). The user should exercise caution when modifying these parameters.

RETURN STATUS:

HOST_NO_ERROR: flash parameter channel opened successfully.
HOST_WRONG_MODE: board must be in recovery mode to use this function.
HOST_DEVICE_ERROR: flash parameter area could not be accessed.

NOTES:

Caution should be used when modifying flash parameters.

host_read_board_info

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_read_board_info( unsigned long vme_address,
                                       void          *ptr,
                                       unsigned long max_words );
```

DESCRIPTION:

host_read_board_info reads the specified data item from the IXA7 board information structure. The function requires the VME base address of the IXA7 board and a pointer to a memory buffer of max_words 32-bit words. The function fills this structure with the board information structure from the specified IXA7 board.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.
HOST_INVALID_ITEM: item not valid.

NOTES:

A definition of BOARD_INFO_ITEM_TYPE is provided in *iop_board_info.h*.

host_read_error_log

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_read_error_log( void          *board_pointer,
                                      unsigned long *log_ptr,
                                      unsigned long *num_items );
```

DESCRIPTION:

host_read_error_log reads a log of errors that were detected during board start up from the IOPlus board information structure. This log is created by the power-on self-test, and can be used to list and diagnose multiple errors found during board testing. The caller must allocate memory for the *log_ptr*, and pass the address of this memory buffer to the function. A maximum size of 16 32-bit words should be allocated. *Num_items* is returned by the function to indicate the number of 32-bit error codes contained in the board error log and copied into the *log_ptr*.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.
HOST_INVALID_BOARD_POINTER: board pointer is invalid.

NOTES:

The error log consists of a series of 32-bit error values, as defined in host_api.h.

host_set_start_address

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_set_start_address(  void     *board_pointer,
                                          HOST_PROCESSOR_ID proc_id,
                                          void *start_address );
```

DESCRIPTION:

host_set_start_address sets the address at which code loaded by *host_load_program* begins executing. This function is not implemented on the IXC board, although it is provided for compatibility with the IXA board family. On an IXC board, code always starts executing from DSP address 0x00000000.

RETURN STATUS:

HOST_NO_ERROR:	success.
HOST_INVALID_BOARD_POINTER:	board pointer is invalid.

NOTES:

This function performs no action on an IXC board (although you can call it and link with it when developing IXC code).

host_unmap_resource

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_unmap_resource( void  **resource_ptr );
```

DESCRIPTION:

host_unmap_resource destroys a mapping that was created with *host_map_resource*. This function should be called when a VME-to-board-resource mapping created by *host_map_resource* is no longer required.

RETURN STATUS:

HOST_NO_ERROR: board opened successfully.

NOTES:

Only pass a *resource_ptr* to this function that was created by *host_map_resource*.

host_vme_read

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_vme_read( unsigned long address,
                                unsigned long num_words,
                                void          *ptr );
```

DESCRIPTION:

host_vme_read reads the specified number of 32-bit words from the VME bus, starting at the specified VME address. The data read from the VME bus is placed into a buffer pointed to by *ptr*.

RETURN STATUS:

HOST_NO_ERROR:	completed successfully.
HOST_VME_READ_ERROR:	VME bus error occurred while reading

NOTES:

It is the responsibility of the caller to allocate the memory pointed to by *ptr*.
Note that a board does not need to be opened using *host_board_open* in order to call *host_vme_read*.

host_vme_write

CALLING SEQUENCE:

```
#include <host_api.h>

HOST_API_STATUS  host_vme_write( unsigned long address,
                                 unsigned long num_words,
                                 void          *ptr );
```

DESCRIPTION:

host_vme_write writes the specified number of 32-bit words to the VME bus, starting at the specified VME address. The data written to the VME bus is placed into a buffer pointed to by *ptr*.

RETURN STATUS:

HOST_NO_ERROR:	completed successfully.
HOST_VME_WRITE_ERROR:	VME bus error occurred while writing

NOTES:

It is the responsibility of the caller to allocate the memory pointed to by *ptr*.
Note that a board does not need to be opened using *host_board_open* in order to call *host_vme_write*.

host_write_config_param

CALLING SEQUENCE:

```
#include "_host_api.h"

HOST_API_STATUS  host_write_config_param( void          *param_ptr,
                                          unsigned long  offset,
                                          unsigned long  value );
```

DESCRIPTION:

This function should only be called when the flash utility can not be used to update flash parameters. *host_write_config_param* updates a parameter located in the non-volatile flash memory. Note that flash memory parameters are used by Dy 4 Systems firmware (startup and runtime code). The user should exercise caution when modifying these parameters.

RETURN STATUS:

HOST_NO_ERROR: flash parameter channel opened successfully.

NOTES:

Caution should be used when modifying flash parameters.
param_ptr must have been previously allocated by *host_open_flash_params*.

Appendix A: SmartDMA Implementation

The IOPlus implements a feature called "SmartDMA". This feature allows the IOPlus to act as a smart DMA controller, asynchronously moving data between board resources, generating interrupts, and handling interrupts. The SPEs control the IOPlus SmartDMA by creating linked lists (or chains) of commands, which the IOPlus then processes upon command. This chapter will first present an overview of how the IOPlus SmartDMA works, and then describe how to use this feature of the IOPlus.

A.1 Command / Response Packet Format

All commands and responses are formatted into standard data packets. These packets consist of a packet header and packet data. The packet header consists of five 32-bit words. The packet data is variable-length. The format of a packet is shown in the figure below.

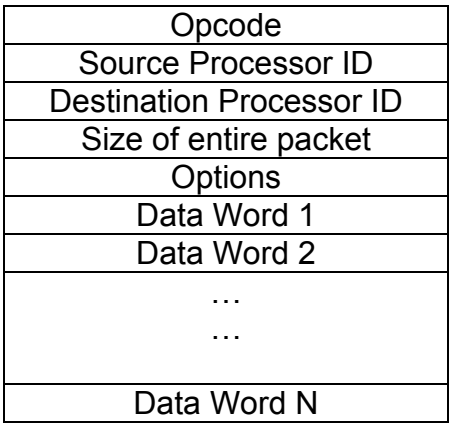


Figure A.1 - Command Packet

The packet header fields are defined as follows:

Opcode:	Defines the command to be performed (or identifies the response)
Source Processor ID:	Identifies the processor (SPE, IOPlus,or host process) which initiated the command.
Destination Processor ID:	Identifies the processor (SPE, IOPlus, or host process) which should receive and process the command
Size of entire packet:	Specifies the number of 32-bit data words in the entire packet, including both the packet header and any attached data. The meaning of the data words appended to the packet header depends on the Opcode.

Options: Specifies options that modify how the command should be processed.

- Bit 0: 1 means respond to this command with a CMD_ACK
  0 means do not respond to this command.
- Bit 1: 1 (atomic operation) means that after the command is processed, the next command in this list will be processed
  0 (fair operation) means that other in-progress lists will be processed after this command has been processed.
- Bit 2-31: reserved (must be set to zero).

Data words: Zero or more data words are attached to the packet, which provide additional information necessary for processing the command. The meaning of the data words is command-specific. Refer to the list of commands processed by the IOPlus (provided later in this chapter) for more information.

A.2 Packet Routing and Processor IDs

The term "packet routing", as used in this manual, is defined simply as the process of getting a packet where it needs to be, so that it can be processed. Packet routing can be either direct or indirect. When the initiating software component places the command packet in a place where it can be accessed and processed by the destination software component, this is referred to as direct packet routing. When the initiating software component places the command packet in a place that cannot be accessed directly by the destination software component, this is referred to as indirect packet routing. In this situation, one or more intermediate software components must transfer the packet from where the initiator placed it to a location that can be accessed by the destination software component.

The IOPlus software supports direct packet routing only. Indirect packet routing is not supported. This implies that software components initiating command packets will place these packets where they can be processed directly by the destination software component.

Supporting only direct packet routing greatly simplifies the assignment of processor IDs. Processor IDs need to be unique only on a specific board, rather than within an entire system. This type of processor ID is sometimes referred to as a relative processor ID or board processor ID. The IOPlus assigns the processor IDs to the processors on an IXA7 board as shown in Table A.1.

Table A.1 - Assignment of Processor IDs

Relative Processor ID	Processor Name
0	IOPlus
1	SPE A
2	SPE B
3	SPE C
. . . .	. . . .
N	SPE N – last SPE on board as defined by flash parameter
N+1	Host process 1
N+2	Host process 2
. . . .	. . . .
N+M	Host process M – last host process that can access board simultaneously as defined by flash parameter

There are situations where a processor ID that is unique within an entire system is required (this type of processor ID is referred to as an absolute processor ID or system processor ID). For this reason, the IOPlus only examines the lower 16-bits of processor ID fields; the upper 16-bits are ignored. Thus, an absolute processor ID can be placed in the upper 16 bits of any processor ID fields, when this information is required by the application. The format of the source and destination processor ID fields in the packet header is provided in Figure A.2.

Absolute processor IDs could be used to discriminate a multi-board IXA7 system. In such a system the first board (board #0) would use an absolute processor ID of 0x0000 for its IOPlus, and the second board (board #1) would use an absolute processor ID of 0x0100 for its IOPlus.

Source Processor ID field:

Bit					
32	……..	16	15	…….	0
Ignored by IOP			Board processor ID		

Destination Processor ID field:

Bit					
32	……	16	15	…….	0
Ignored by IOP			Board processor ID		

Figure A.2 - Source and Destination Command Packet Fields

Host processes also are assigned "processor IDs". This ID is used for generating response packets, as well as for generating interrupts and using semaphores. Each host process accessing the board must have a unique "processor ID" (how these IDs are assigned is described in the next paragraph). The number of simultaneous host processes that can access the board is controlled by a parameter in flash, and thus is variable (the number of SPEs on the board is also variable). As shown in Table A.1, the host processor IDs are assigned immediately after the last SPE processor ID.

A.3 Assignment of IDs to Host Processes

Each host process "attached" to a board must have a unique "processor ID" number for communicating with software components on that board. Note that the "processor ID" which a host process uses to communicate with one board may differ from the "processor ID" that the same host process uses to communicate with a second board.

Host Process "Processor IDs" are assigned on a first-come, first-served basis. A host process must "attach" to a board before it communicates with any of the software components on the board. After all communications with a board are completed, the host process should "detach" from the board. Failure to detach from a board will result in board resources being wasted, since they will not be properly released for use by other host processes.

Attaching to a board:

1) Get address of host list table from board information structure (see section 5.5)
2) Get maximum number of allowed host processes from board information structure
3) Scan host list table for free entries (entries which are non-zero)
4) When a non-zero entry is found, perform a VME read/modify/write cycle to set the zero-value to a any non-zero value. An atomic access must be used to test the zero-value and write the non-zero value; otherwise, it is possible for two host processes to be assigned the same ID.
5) The "processor ID" for the host process is determined using the following equation:
   "processor ID" = offset of non-zero entry + "maximum # of SPEs on board" + 1

Detaching from board:

1) Get address of host list table from board information structure.
2) Determine offset in "host list table" using the following formula:
   Offset = "processor ID" – "maximum # of SPEs on board" - 1

3) Write a zero to this offset in the "host list table" (note that this write does not need to be atomic).

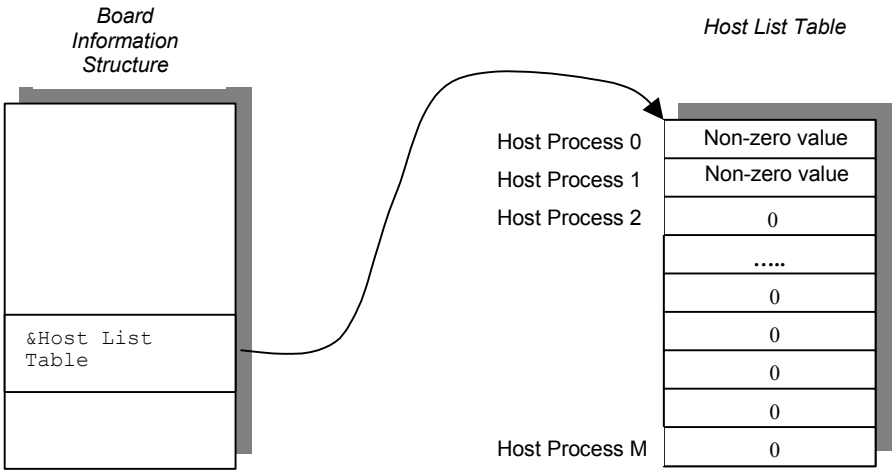


Figure A.3 - Host Process ID Assignment with Two Host Processes Already Attached

A.4 Board Information Structure

The board information structure is a global repository of information that describes the configuration of a board. The structure is accessible from a host process (through the VMEbus), from the IOPlus (through its local bus), and from the SPEs (through the PCI bus). See 4.3 for the memory mapping of the board information structure.

A.5 Linked Command List Overview

The interface supported by the IOPlus uses linked lists of commands and linked lists of responses. Each processor can create multiple linked lists of commands, but the IOPlus can process only one linked list of commands at a time. Linked command lists can point to other lists, and can be used to create complicated command sequences, which can be "played" by the IOPlus upon command. Linked lists can be located either in global memory (the IOPlus' SDRAM) or local SPE memory.

A linked list of commands or responses consists simply of a sequence of command / response packets. The packets are encapsulated in a simple data structure, with a "next" pointer preceding the packet header. The "next" pointer is used to "link" a command to the next command in the list (see Figure A.4 for an example of this structure).

The link lists organizational structure used by the IOPlus consist of a "master list address table" and groups of "list address tables".

Master List Address Table

The "master list address table" is a list of pointers to individual "list address tables". A "list address table" contains pointers to linked lists of commands created by software components for processing by the software component that owns the "list address table", as well as pointers to linked lists of responses created when the owning software component processes the command lists. A single "list address table" is provided for sending commands to the IOPlus. Slots for additional "list address tables" are provided for each SPE, and for the host processes. **Note that SPE and host linked list command support are for future expansion. However, the IOPlus will create the "list address tables" for these software components.**

The address of the "master list address table" is located in the board information structure and is always located in global memory. The "list address table" associated with each software component is also located in global memory, although individual entries in a "list address table" may be relocated into SPE local memory. Linked command lists may be located in either global or local memory, but a single command list must reside entirely in either local or global memory. Figure A.4 illustrates a "list address table" with all the linked command list entries in global memory. Figure A.5 illustrates a "list address table" that has the entry for SPE 2 linked command list relocated into SPE 2's local memory so that the table can be accessed without using the PCI bus.

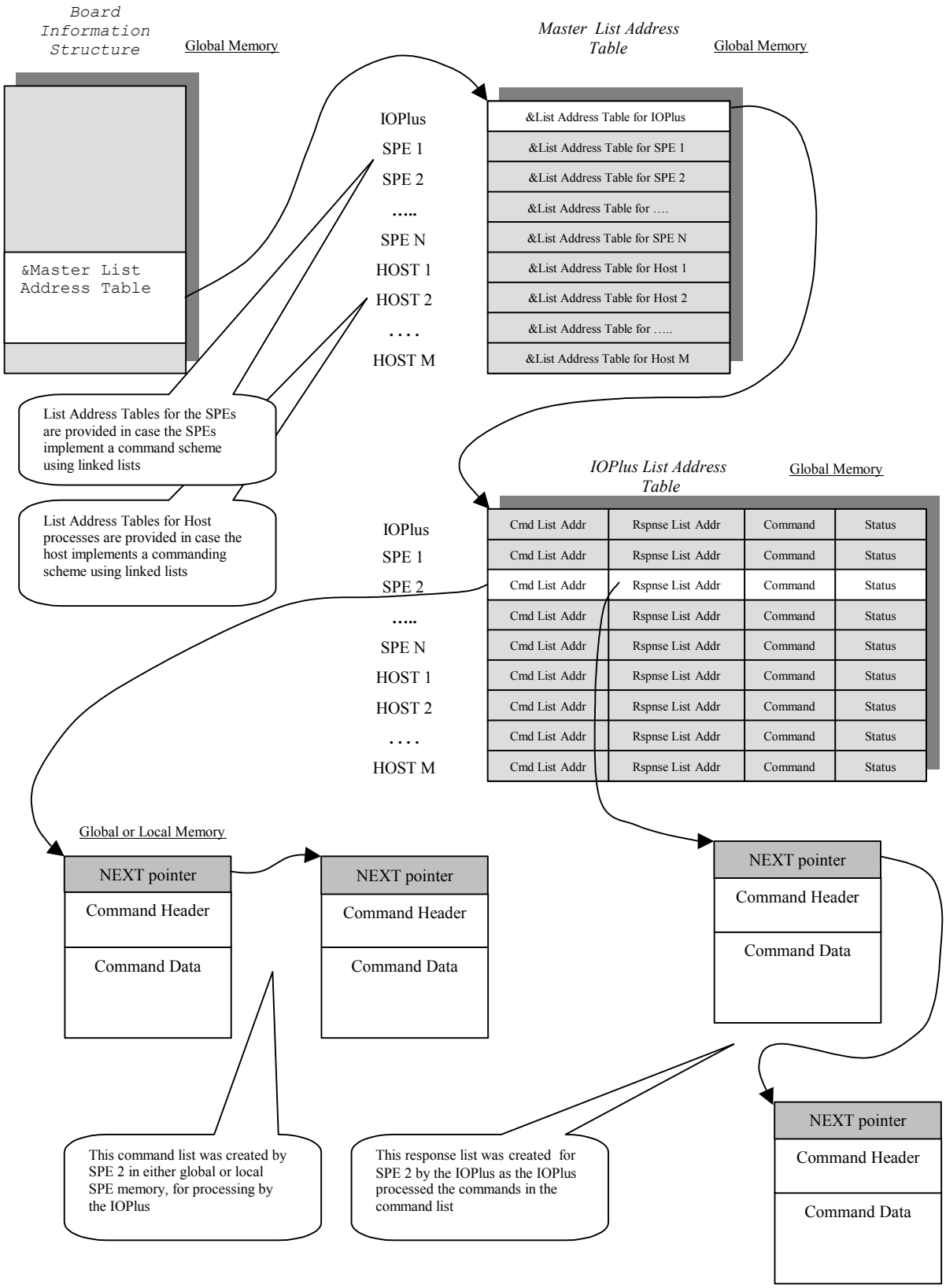


Figure A.4 - All List Address Table Entries in Global Memory

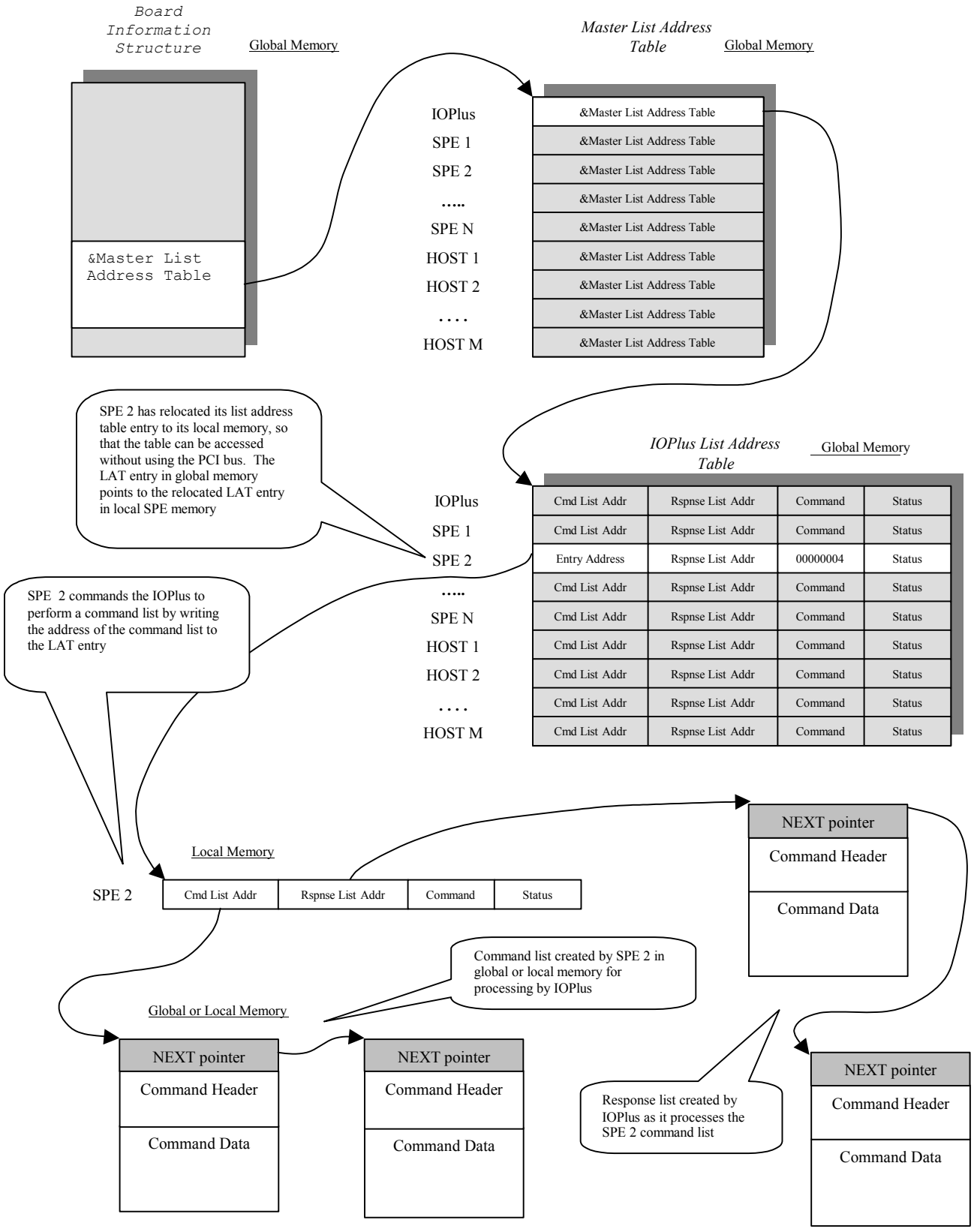


Figure A.5 - List Address Table Entry Relocated to Local Memory

List Address Table

The "list address table" for the IOPlus, located in global memory, is a list of pointers to linked command lists created by other software components, and linked response lists created by the IOPlus as it processes command lists, as shown in Figure A.4. The "list address table" also contains command modifiers and status indicators for each command/response list. When a software component wishes to send a command to the IOPlus, it creates a linked list of commands in either global or local memory. It places the address of the linked command list into the IOPlus's "list address table" using its relative processor ID as an index into the table (eg. SPE 2 will write into the table entry reserved for SPE 2). After writing the command list address into the table, it generates an interrupt to the IOPlus.

> After writing the command list address into the table, an interrupt to the IOPlus must be generated to start the command processing.

A "list address table" entry for a software component can be relocated from global memory to SPE local memory by setting the "relocate" bit in the "command option" field of the "list address table" entry being relocated. When an entry is relocated, the "command list address" field in the global memory "list address table" entry must contain the address in SPE local memory of the relocated "list address table" entry. The relocated "list address table" entry contains pointers to the actual linked command and response lists, which can be located in either global or local SPE memory. Figure A.5 illustrates SPE 2 relocating its "list address table" entry to its local memory.

Why relocate a "list address table" entry from global into local memory? Depending on the application, it may be valuable to locate the commands "closer" to the IOPlus (i.e. in global memory), or "closer" to the SPE (i.e. in SPE local memory). For instance, to minimize PCI traffic used for IOPlus commanding, pre-configure the IOPlus command lists in global memory. The command lists can then be started (and restarted) by doing a single PCI write.

A.6 Linked List Management Protocol

The linked list interface has been designed to provide a simple, low latency mechanism for commanding the IOPlus. The protocol for using the linked list interface is straightforward. A description of the process of initiating a command sequence using a linked command list and responding to a linked command list is provided below.

Relocate the "List Address Table" entry if necessary:

1) Determine whether the "list address table" entry should be located in global or local memory. If the table entry does not need to be relocated, then skip to the "Initiating" portion of this sequence. To relocate the table entry from global to local memory, perform the following steps:
2) Allocate 16 bytes of local memory to hold the relocated "list address table" entry.
3) Locate the "list address table" using the pointer contained in the "master list address table".
4) Locate the appropriate "list address table" entry using the software component's relative processor ID as an index into the "list address table".
5) Relocate the "list address table" entry from global to local memory by writing the SPE local address of the 16 byte region allocated in step 2 into the "command list address" field of the "list address table" entry in global memory.
6) Set the relocation bit in the "command option" field of the "list address table" entry in global memory.
7) The table entry is now relocated from global to local memory.

Initiating:

1) To initiate a command sequence, a linked list of commands must be created in memory that belongs to the initiating software component. This list can be located either in local memory, in the user portion of global memory, or in a reserved area of global memory that is dedicated to the linked lists belonging to each software component. The address of this reserved area of global memory is found in the board information structure. The "command list address" field of the "list address table" entry is used to indicate whether the linked lists are located in local(1) or global(0) memory.
2) Verify that the IOPlus has finished the previous command sequence by checking that the active bit in the "status" field of the "list address table" entry is zero. If it is non-zero, then try again later.
3) Write the address of the first packet of the command sequence to the appropriate "command list address" field in the "list address table" entry belonging to the software component generating the command. Note that if the "list address table" entry was relocated into local SPE memory, then the address of the first packet should be written to "command list address" field in the local "list address table" entry, rather than in the global memory "list address table" entry.
4) If you wish to specify where the IOPlus should place responses, write the address of the response area to the "response list address" field in the "list address table" entry. If you want the IOPlus to manage the memory for response packets, write a zero to this location.
5) Generate an interrupt to the IOPlus.
6) The IOPlus will clear the "active" bit of the status word in the "list address table" entry when the command sequence has been processed.

Responses:

1) The IOPlus will respond to all commands with a CMD_ACK packet (see the description of this packet below). If the address in the appropriate "response list address" is zero, then the IOPlus will place responses in a reserved area of global memory. If the address is non-zero, then the IOPlus will place responses at the specified address. The address can be either in global or local SPE memory, as specified by a bit in the command option field of the "list address table" entry.
2) A list of responses will be created at the "response list address" as the IOPlus performs the command sequence. This list will grow as the command sequence is processed. Upon successful completion of the command sequence, the IOPlus will clear the "active" bit of the appropriate status word in the "list address table". If an error is encountered while processing the command sequence, the IOPlus will set the "error" bit of the appropriate status word in the "list address table". The initiating software component can determine the type of error by examining the linked list of responses (if it doesn't care, it can simply ignore the response list).

A.7 Command Option and Status Register Definition

The command option register associated with each software component in the "list address table" is used to specify how linked command lists are processed. Each bit in the register is defined in the following table.

Table A.2 - Command Option Register

Bits	Name	Function
0	Command List Address Location	0 ← the address in the "command list address" field refers to global memory 1 ← the address in the "command list address" field refers to local memory
1	Response List Address Location	0 ← the address in the "response list address" field refers to global memory 1 ← the address in the "response list address" field refers to global memory
2	Relocation flag	0 ← this entry is not relocated; the addresses in the "command list address / response list address" fields point to linked command list packets 1 ← this entry is relocated; the address in the "command list address" field points to the address of the relocated entry in local SPE memory; the "response list address" field is unused.
3	Halt on Error flag	0 ← IOPlus should continue processing subsequent commands when an error occurs in one command 1 ← IOPlus should not process subsequent commands when an error occurs in a command
4	Only save errors flag	0 ← Save all response packets 1 ← Only save response packets when they contain an error
5-12	SPE ID	Indicates which SPE the command list belongs to. SPE Ids are: SPE A: 0x00 SPE B: 0x40 SPE C: 0x80 SPE D: 0xC0
13-30	3-30	Reserved.
31	Stop	Commands the software component to stop processing the chain. 1 ← Stop chain processing 0 ← Do not stop chain processing

The status register (see Table 5.3) associated with each software component in the "list address table", is used by the IOPlus to report summary status information to each software component.

Table A.3 - Status Register

Bits	Name	Function
0	Active bit	0 ← IOPlus is not currently processing the command list pointed to by this table entry 1 ← IOPlus is currently processing the command list pointed to by this table entry
1	Error bit	0 ← no error occurred during the processing of the command list pointed to by this table entry 1 ← an error occurred during the processing of the command list pointed to by this table entry Note: The contents of this bit are only valid when the "Active bit" is zero
2	Done bit	0 ← IOPlus has not completed this command chain 1 ← IOPlus has finished processing this command chain
3	Blocked bit	0 ← This command channel is not blocked 1 ← This command channel is blocked on a shared resource
4	Halted bit	0 ← This command channel has not been halted 1 ← This command channel has been halted
5	Waiting for Interrupt bit	0 ← Command channel is not blocked on an interrupt 1 ← Command channel is blocked waiting for an interrupt
6	Interrupt found	0 ← No interrupt found 1 ← The interrupt that this command channel was waiting for (if any) has occurred.
7-31	reserved	Reserved for future use

A.8 Interrupt Protocol

The SPEs interrupt the IOPlus when they want the IOPlus to begin processing a command list. When the SPEs are running VxWorks, interrupts are generated by calling *ixa_ipi_interrupt*. It the host wishes to interrupt the IOPlus, it writes to the attention flag in the board information structure.

A.9 Flash Memory Management Protocol

The IXA7 provides from 4 MB to 16 MB of on-board non-volatile flash memory. This memory needs to store a variety of different data types as defined in Table A.5.

SPEs should not access the flash memory directly (even though it is possible to do this through the PCI bus); rather, they should use the commands in IXAtools to access flash memory. This section is provided as support information for using those commands.

Table A.5 - Flash Memory Data Types

Single or Multiple	Data type Name	Directory Entry Name	Description
S	Directory	directory	Specifies all items stored in flash memory. The format of the directory is provided below
S	Initial boot	iop_copy	Executes from flash; copies startup code to memory
S	Minimal boot/recovery code	iop_recovery	This item is located at the IOPlus boot address. It either jumps to the initialization code, or performs minimal initialization and then waits for the flash to be reburned. This sector is electrically write-protected.
S	Initialization/startup code	iop_startup	This code initializes the hardware attached to the IOPlus, including the Xilinx, MPC-107s, Universe II, and PCI bridge chips. After completing, it jumps to the run-time code
S	Run-time code	iop_runtime	The run-time code for the IOPlus command servicing.
S	Production parameters	prod_params	These parameters are set when the board is initially built, or when it is returned for a RMA. This sector should be electrically write-protected.
S	Configuration parameters	config_params	Various parameters which control the operation of the board, including VME parameters and PCI parameters
S	FPGA 1 program	xilinx	Data stream used to program the Xilinx
S	User programs	spea, speb, spec, sped, speab, speac, spead, spebc, spebd, speabc, speabd, spebcd, speabcd	User programs can be automatically loaded into the SPEs by the IOPlus upon startup
S	User global memory load	Gmemdata	This may be code or data that will be loaded into global memory on startup
M	User data		This is data that the user wants to store in non-volatile memory

A directory is stored in flash, which describes the location, size, and type of all items currently stored in flash. The directory is located at the flash base address, and shall contain 64 entries. An example directory is shown in Table A.6.

Table A.6 - Example Flash Directory

Entry Offset (4 bytes)	Entry Size (4 bytes)	Entry Name (32 bytes)
00000000	768	directory
00100100	1000	iop_copy
00101100	5000	iop_startup
00200000	2000	prod_params
00202000	2000	config_params

1) Note that data does not need to be stored contiguously in flash. Data is segmented so that maximum usage is made of the available flash memory space.
2) Entry names must be unique.
3) Reserved entry names (as shown above) have special meaning to the IOPlus software, and should not be used by application software.
4) User programs, which are intended to run on the SPEs, must be in S-Record format before they are written into flash memory. The name *spe[a][b][c][d]* you give to the file when writing it determines what SPE(s) it gets loaded to upon board reset. For instance, if you want the program to automatically load into SPEs A, C and D upon board reset, when writing the program into flash, name it spe*acd*. Chapter 8 has more information on the procedure for writing programs into flash memory.
5) User global memory load, can be a program or data that will be automatically written to global memory at board reset. The contents must be burned into flash memory from an S record file so that address information is obtainable. The contents can be written to any location in global memory with the exception of the addresses between 0x4000 – 0x40000. A configuration flag can be set to start execution of the contents at the completion of the copy to global memory. See Chapter 8 for more information on using this option.

Index

B

board configuration utility 8-1
board information structure 4-5, 5
board jumpers 2-2
board layout 2-1
Board Resource Manager 3-4

C

command channels 5-4
Common Boot Code 7-1
conventions used in this manual 1-1
COP interface 2-16
CPE 1-2

D

diagnostics 2-11

E

Emulator 2-16
Ethernet
  boot parameters 2-17
  PMC-ENET1 2-6
  Transsition Module 2-5

F

flash memory 3-6, 13
  updating 2-21
  writing to 5-35, 8-1
flash parameters 2-4

G

Geographical Address
  register 4-18
global memory 4-1

H

host list table 4

I

interrupt mask registers 4-8
interrupt protocol 13
interrupts
  generating 5-24
  status registers 4-10
  waiting for 5-33
IOPlus 3-2
  command format 5-2, 1
  commands 5-17–6-30
  host process IDs 4
  linked command list 5
  memory map 4-1
  Overview 1-1
  packet routing 2
  processor IDs 2
IOPlusAPI 6-9
IXA7
  board architecture 3-1
  Overview 1-1
IXAbsp 7-1
IXAtools 1-3
  host software 9-1
  installing 2-20
  porting 9-1–9-9
  programming with 7-1
  uninstalling 2-21

J

JTAG 2-1
  connectors 2-16

L

LEDs 2-1, 2-11, 5-30

M

master list address table
  address of 4-5

P

PCI bus
  configuration 8-8
  data moves 5-26
  snooping 3-6
PCI buses 3-4
PMC 1-3
PMC sites 3-6
  installing 2-4
  power specs 2-7, 3-12
PMC-ENET1 2-6
PPMC support 3-11

S

SBSRAM 3-5
SDRAM 3-5
SPE-PCI Bridge 3-3
SPEs 3-2
  memory map 4-3

T

Transition Module 2-5

U

Universe II 3-3

V

VMEbus....1-3
base address config. 2-4
configuration....8-5
connector defs....3-12
data moves 5-26
rmw operations 5-32
slot one operation....2-4
VMElib
porting.... 9-1–9-9
VxWorks
boot parameters....2-17
on IOPlus 6-1
on SPEs....2-21
shared memory routing 2-19

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