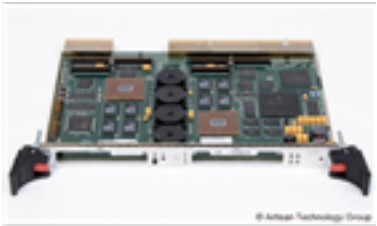
DY-4 Compact CHAMP-AV

High Performance Digital Signal Processing (DSP)



In Stock

Used and in Excellent Condition

Open Web Page

https://www.artisantg.com/61739-1

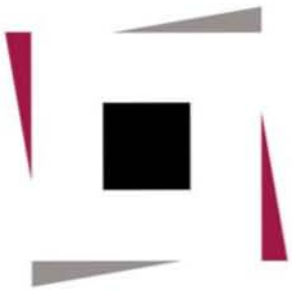
ARTISAN TECHNOLOGY GROUP

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

Artisan Technology Group

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

- Critical and expedited services
- In stock / Ready-to-ship
- We buy your excess, underutilized, and idle equipment
- Full-service, independent repair center



Dy 4 Systems

The future of high-integrity, embedded technology

Manual for the IXA4 Quad PowerPC Board

Version 2.8
September 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**

Customer Support

Technical Support Hotline

If you have any questions about the IXA4, 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

List of Figures

List of Tables

Chapter 1: Getting Started

1.1 Introduction

This manual describes the IXA4 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 IXA4 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 cPCI and PCI bus
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 IXA4 is a 6U cPCI board featuring four Motorola PowerPC 7400 or 7410 processors, a PowerPC referred to as the IOPlus, and two PMC sites. Figure 1.1 depicts the IXA4 architecture.

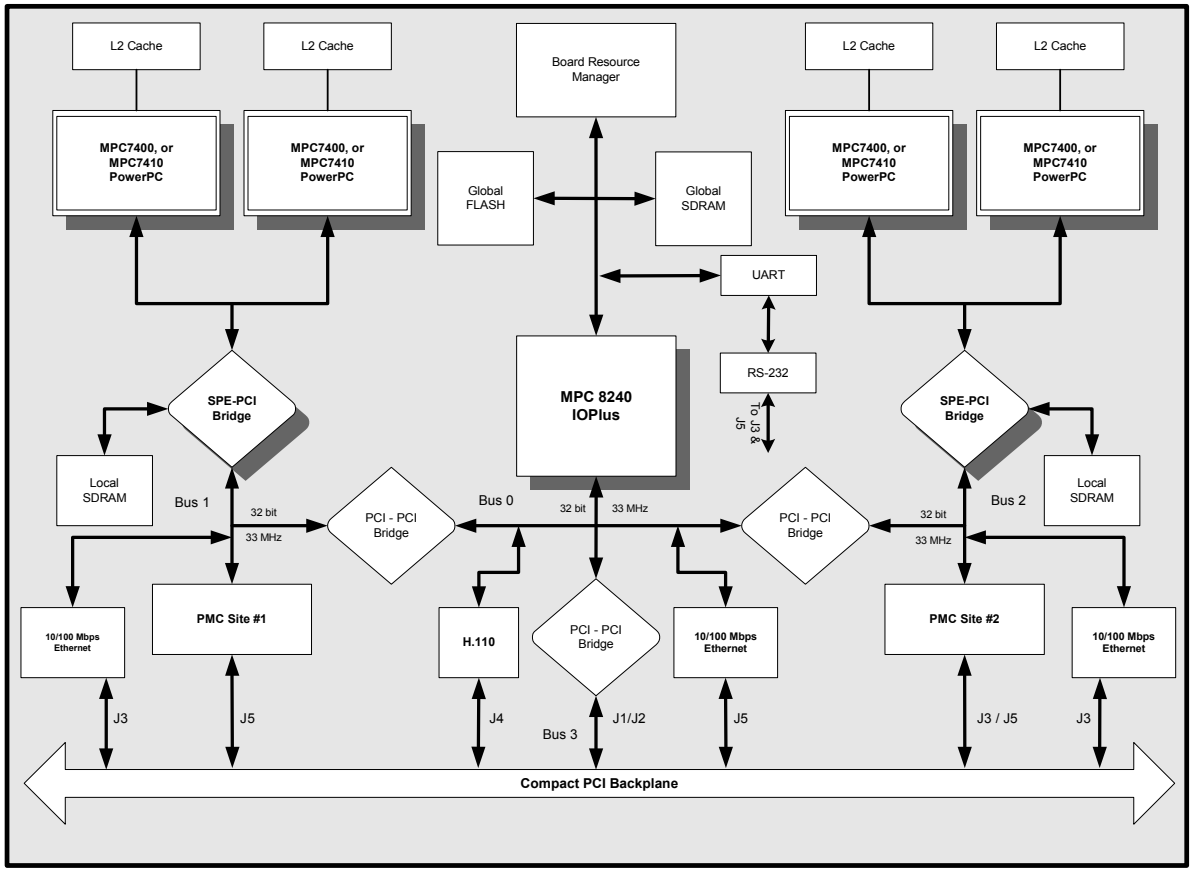


Figure 1.1 - Block diagram of the IXA4

The IXA4 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 - PCI bridges, Dual UART (DUART) 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 an 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, cPCI bus 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 IXA4 board. The local memory can be populated with up to 256 MB of SDRAM (100 MHz) per SPE. Global memory is

accessible by the IOPlus and the four PowerPC processors, and is populated with 64MB of SDRAM (100 MHz).

The internal 33MHz, 32-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 cPCI bus accesses global memory simultaneously.

The IXA4 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 (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 IXA4 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 cPCI bus. 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. Two serial interfaces into the board are also provided.

The user of the IXA4 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:

Draft Standard Physical and Environmental Layers for PCI Mezzanine Cards: PMC; IEEE P1386.1-2001

Draft Standard for a Common Mezzanine Card Family: CMC; IEEE P1386-2001

Draft Processor PMC Standard (VITA 32-199x) Draft 0.5, May 9, 2002.

PCI Local Bus Specification Revision 2.3, 29 March 2002

PowerPC™ Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, Motorola

CompactPCI Specification, PICMG 2.0 R3.0, October 1, 1999

CompactPCI Hot Swap Specification, PICMG 2.1 R2.0, January 17, 2001.

ComapactPCI Computer Telephony Specification, PICMG 2.5 R1.0, April 3, 1998

Chapter 2: Installation

2.1 Some Cautions

The IXA4 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 / COP pins together.

> ! WARNING: Both PMC sites only support 5V signaling bus levels. 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 IXA4 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 switches, and status LEDs.

The Product Information Block contains part number, Assembly/Revision number and serial number information. The second line is the Assembly/Revision number. The first alpha-numeric character is the revision on the baseboard. Different revisions of the baseboard have different cPCI pinouts on J3 and J5 and different PMC to backplane pin mappings. Tables in the relevant sections later in this document contain the different mappings. Locate the Assembly/Revision number and use this to locate the proper table for you revision on IXA4 board.

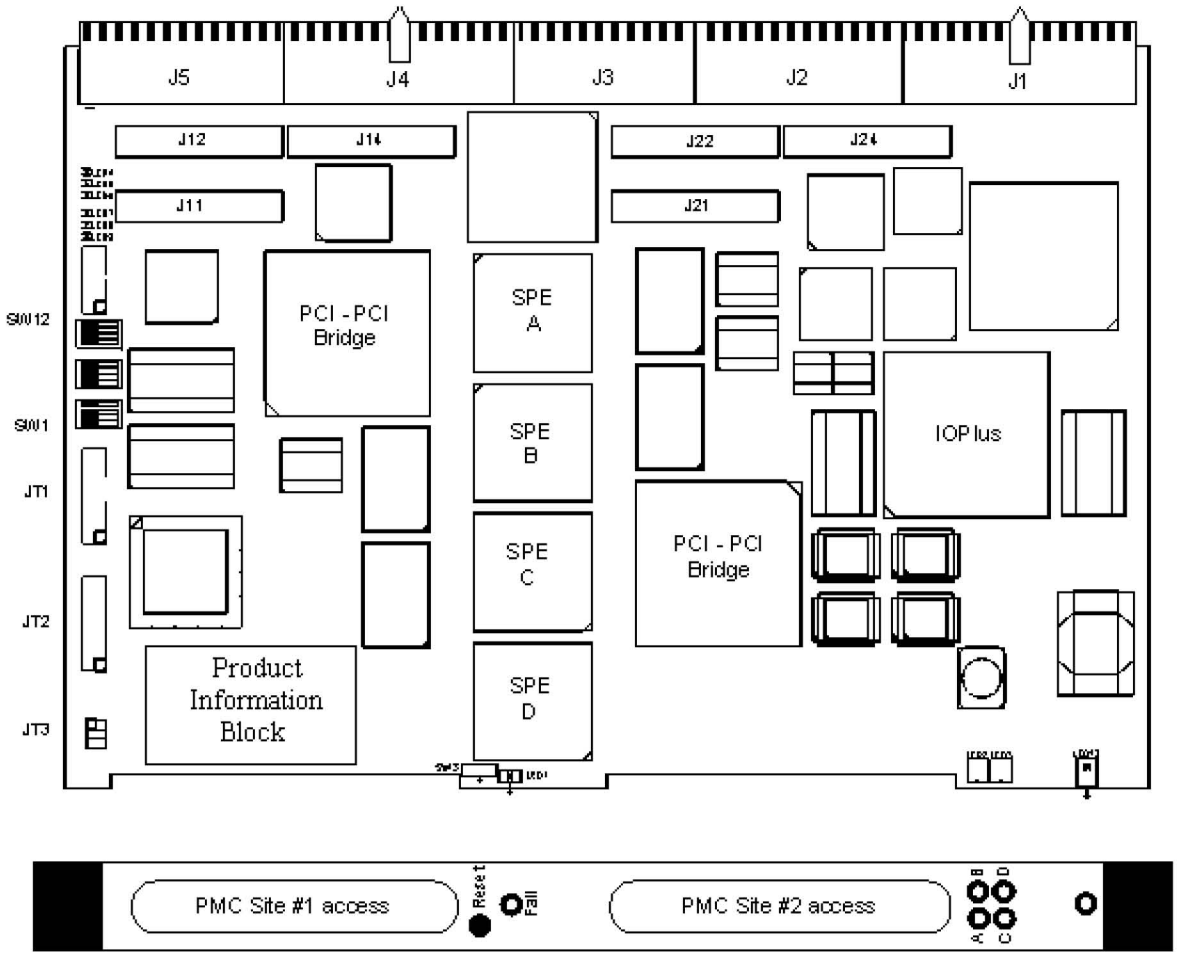


Figure 2.1 - IXA4 board layout, top and front panel views

2.3 Configuring the Board's DIP Switches

Table 2.1 defines the DIP switches (SW1 – SW12) on the IXA4 that reside at top middle of the board. These are the only switches on the board. The "On" and "Off" positions are indicated in silkscreen on the board, near SW1. Switches are available for setting the various board operational modes. Switches 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 – DIP Switch definition

Switch	On	Off	Default
SW1	Reserved		Off
SW2	Reserved		On
SW3	Reserved		On
SW4	Reserved		On
SW5	Standard Startup/Runtime code will execute	Minimal boot recovery code will execute, allowing FLASH to be re-burned or reconfigured[1]	On
SW6[2]	Reserved but **must be left in ON position** for board to work properly in recovery mode.		On
SW7	Load common boot code and SPE applications	Inhibit loading of common boot code and SPE applications on reset	On
SW8	Enable VxWorks Boot ROM[3]	Do not load VxWorks Boot ROM program	On
SW 9 & SW10	These two switches are used to select which SPE the JTAG header corresponds to. They have the following meaning: SW9 / SW10 / SPE Off / Off / D Off / On / C On / Off / B On / On / A		
SW11	Reserved. **Must be left in Off position**		Off
SW12	Reserved		Off

2.4 CompactPCI Interface

The IXA4 utilizes an Intel 21554 PCI-PCI bridge. This bridge is called non-transparent, meaning that the addresses on one side are completely independent of addresses on the other. All communication between the two sides makes use of address translation registers which take addresses from one domain and map them to addresses in the other. This addresses scheme allows the address map on each side of the bridge to be set up independently of the other.

[1] Turning off SW5 boots the board into a minimal recovery state, so that firmware can be re-burned, or FLASH parameters can be re-configured. This is used when FLASH firmware or configuration parameters have become corrupted.

[2] This switch has no meaning unless SW5 is also OFF.

[3] This switch has no effect unless VxWorks is loaded into FLASH memory and enabled using FLASH configuration parameters.

The IXA4 Board functions as a Compact PCI Peripheral Slot board only, it cannot be used in the System Slot. Among other things this means that it does not generate the cPCI locking and arbitration, and only transmits interrupts on the backplane.

It is the responsibility of the system controller (the CPU board plugged into the system slot) to configure the CompactPCI addresses. It will determine the memory map for each board in the system and assign a base address for each IXA4 board in the system. The startup code on the IXA4 board will set up its internal memory map and will program the internal (on board) side of the address translation registers so that when a CompactPCI address destined for the IXA4 is seen it goes to the appropriate on board address. Chapter 4 contains more information on configuring the internal side of the address translation registers.

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 filler panel that is attached to the IXA4 by Dy 4 Systems before shipment.

2.6 Installing the Rear Panel Module

If you have purchased the Rear Panel module, IXA-RP, you must configure the module's jumpers prior to use. Figure 2.2 depicts the location of jumpers and connectors on the IXA-RP. The connectors are designed to plug into the backside of the cPCI backplane, and must be plugged into the same slot as the IXA4 board. The Rear Panel module provides three Ethernet connections and two serial port connections.

The IXA-RP has jumper blocks that are used for configuring the serial ports. 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 IXA4 configures the Ethernet ports. Ethernet port configuration and the serial port pin out are documented later in this chapter.

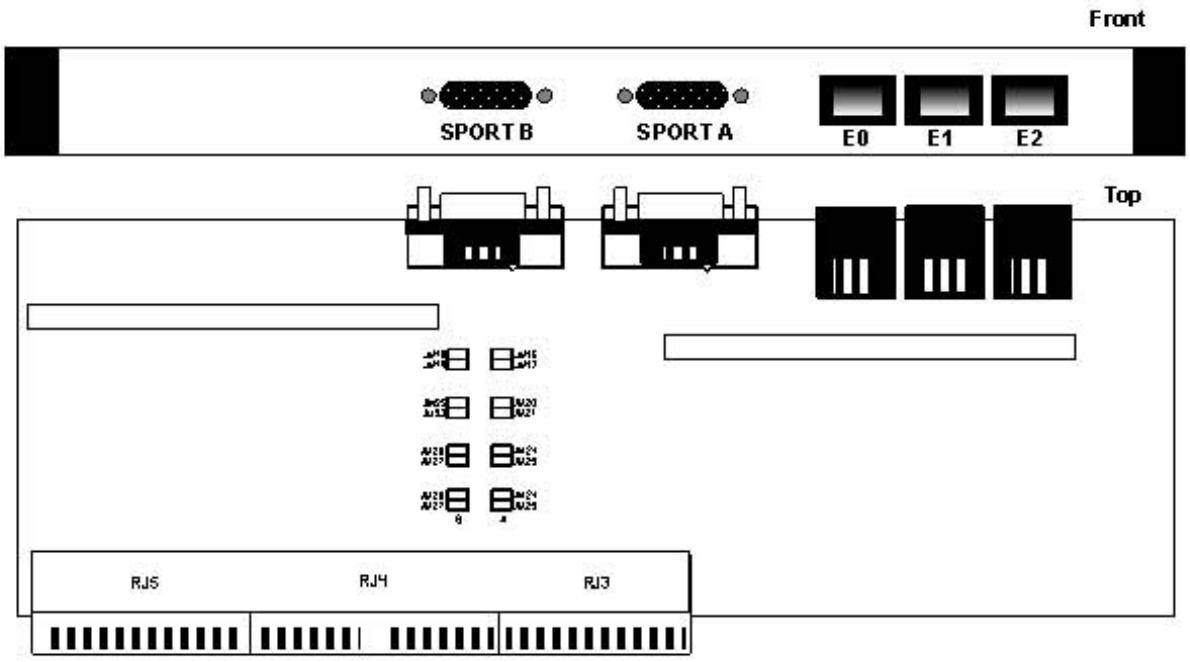


Figure 2.2 – Rear Panel Module layout

Table 2.2 – Rear Panel Module Jumper Definition

Mode	Settings
Sport A set for DTE (default setting)	Shunt JM16 Shunt JM17 Shunt JM20 Shunt JM21 Shunt JM24 Shunt JM25 Shunt JM28 Shunt JM29
Sport A set for DCE	Shunt JM16 to JM17 Shunt JM20 to JM21 Shunt JM24 to JM25 Shunt JM28 to JM29
Sport B set for DTE (default setting)	Shunt JM18 Shunt JM19 Shunt JM22 Shunt JM23 Shunt JM26 Shunt JM27 Shunt JM30 Shunt JM31
Sport B set for DCE	Shunt JM18 to JM19 Shunt JM22 to JM23 Shunt JM26 to JM27 Shunt JM30 to JM31

2.7 Verifying cPCI Backplane Power

The IXA4 uses 5 V and 3.3 V power sources from the cPCI bus backplane. +12 V and –12 V from the backplane go directly to the PMCs and are not used on the IXA4 board. 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 - IXA4 Power Supply Tolerances

Power Supply	Required Tolerance	Maximum Ripple (pk – pk)
+5 V	4.875 V to 5.25 V	50 mV
+3.3 V	3.20 V to 3.45 V	50 mV
+12 V	11.4 V to 12.6 V	240 mV
-12 V	-11.4 V to -12.6 V	240 mV
V(I/O)	Same as 5 V or 3.3 V tolerance	50 mV

2.8 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 CPE starts execution from FLASH memory, and the SPE processors stall, waiting for their bridges to be enabled. The red Fail LED illuminates.

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

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

4. The CPE 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 Fail 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 edit the FLASH boot parameters.

10. After finishing the count down, the boot ROM 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 slot ID, as determined by the slot ID read from the backplane.

%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.

2.9 Power-up Diagnostics

The front panel LEDs are used to indicate the health and status of the IXA4 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 and +5 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	Reserved
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 failed test
On	on	on	on	on	Board held in reset

2.10 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 VxWorks command bootChange when run from windShell is asynchronous to burning the FLASH. That is, when the command is done, the command prompt immediately comes back and does not wait for the FLASH to complete the burn. Because of this, a user may reset the card before FLASH has been completely burned, which will corrupt the FLASH memory. For further information, please contact Dy4 Systems Technical Support.

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: switch SW7 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. Restore SW7 to the on position.
7. Reapply power to the board.

Recovering from VxWorks Boot ROM or Startup Firmware Errors

A faulty VxWorks 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: switch SW5 and SW6 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. Restore SW5 and SW6 to the on position.

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: set SW5 off, SW6 on, and install the card in a slot fitted with a rear panel module.

3. Connect the serial port from the 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. Restore SW5 to the on position.

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. Set SW5 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. Restore SW5 to the on position.

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

2.11 JTAG/COP Connections

There are two JTAG/COP connectors on the IXA4 board: JT1 for the IOPlus and JT2 for the SPEs.

> ! WARNING: Improper connection of the emulator can damage the IXA4 board and/or the emulator. Emulator connector should be keyed.

2.12 Configuring an Emulator for an IXA4

An emulator may be used to interact with each processor and the board hardware on the IXA4 card. Two JTAG/COP connections are provided on the IXA4 for this purpose. Connector JT1 is used to connect an 8240 emulator to the IOPlus processor while connector JT2 is used to connect an 74xx 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 2.1 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 IXA4 with WindRiver VisionProbe emulators. Therefore, the information in this section is based on the use of the WindRiver 74xx and WindRiver 82xx emulators. Both VisionICE and VisionProbe emulators have been used.

> ! WARNING: When emulating the IOPlus, apply power to the emulator after the IXA4 has had power applied to prevent driving signals to un-powered components on the IXA4 card. Ignoring this warning, in some instances, may result in card latch-up.

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 to support the SPE processors, this connector is multiplexed between the four SPE processors. The target processor is chosen via IXA baseboard switches SW9 and SW10 (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 SW9 and SW10 to the processor that you wish to target.
3. Connect the emulator to the JT2 connector on the IXA4
4. Power up the IXA4 board.
5. Open the Vision Click software and follow the procedures outlined for opening communication with the emulator.
6. Type “inn” to get into background mode (do not type “in” as this command will attempt to initialize the SPE-PCI bridge).
7. Type “go” to put the emulator and card into the proper state.
8. 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 WindRiver 74xx emulator, this means that each time the “inn” command is executed, the IXA4 board will reset. Once the board has been reset, it is important to run the emulator (step 7). 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 reset 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 WindRiver emulator) will reset the entire IXA4 board.

2.13 Configuring the VxWorks Boot Parameters

The SPORTA serial port on the IXA-RP module is used to configure the VxWorks boot parameters. When the IXA4 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 the boot loader 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 compatible with the jumper settings on the IXA-RP (See Table 2.2). SPORTA by default is RS-232, 9600 baud, 8 bit, no parity, and 1 stop bit and the driver expects a software handshake protocol (XON/XOFF). The DB-9 connector is configured with the jumpers on the IXA-RP. See Table 2.6 for the pin definition of the DB-9 for DTE and DCE operation.

Table 2.6 - Serial Port Connector Definition

Pin	Signal	Description	DCE Definition	DTE Definition
1	DCD	Data Carrier Detect	output	input
2	RD	Receive Data	output	input
3	TD	Transmit Data	input	output
4	DTR	Data Terminal Ready	input	output
5	GND	Signal Ground		
6	DSR	Data Set Ready	output	input
7	RTS	Request to Send	input	output
8	CTS	Clear to Send	output	input
9	RI	Ring Indicator	output	input

Once you have the boot 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          : fei
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 “fei” to specify the INTEL 82559 device. The updated boot parameters are stored in FLASH on the IXA4 and preserved there when the card is powered off. Therefore, subsequent boots of the card can auto-boot without intervention.

The boot parameters are also parsed by each SPE to automatically set the SPE boot parameters for your site. The “other” field must be filled in with the full path of the SPE VxWorks image (vxWorks or vxWorks5_2) in order to successfully boot the SPEs. Make sure the version of the image file corresponds to the file you intend to burn into FLASH; spworks.s or spworks5_2.s.

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 IXA4 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.

So, for example, if the IOPlus boot parameters are as shown above, the route add command to add the IOPlus to the route table would be:

```
route add 192.99.22.100 207.96.24.235
```

And for the SPE's:

```
route add 192.99.22.101 207.96.24.235
route add 192.99.22.102 207.96.24.235
route add 192.99.22.103 207.96.24.235
route add 192.99.22.104 207.96.24.235
```

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

When using more than one IXA4, be sure to assign different IP addresses for each board. Table 2.7 shows an example of a three-board system configuration.

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
3	IOPlus	192.99.22.300	207.96.24.237
3	SPE A	192.99.22.301	None
3	SPE B	192.99.22.302	None
3	SPE C	192.99.22.303	None
3	SPE D	192.99.22.304	None

2.14 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 which 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:

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 IXA4. 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 IXA4's FLASH memory. This utility runs under Windows 95/98/2000/NT and requires an Ethernet connection between the IXA4 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 cPCI chassis with the IXA4. Software libraries are provided with IXAtools to support applications running on this computer to load, start, and exchange information with the IXA4 over the cPCI bus. 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 IXA4 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 7 describes the SPE BSP and its use.

Chapter 3: Hardware Architecture

3.1 Introduction

The IXA4 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.

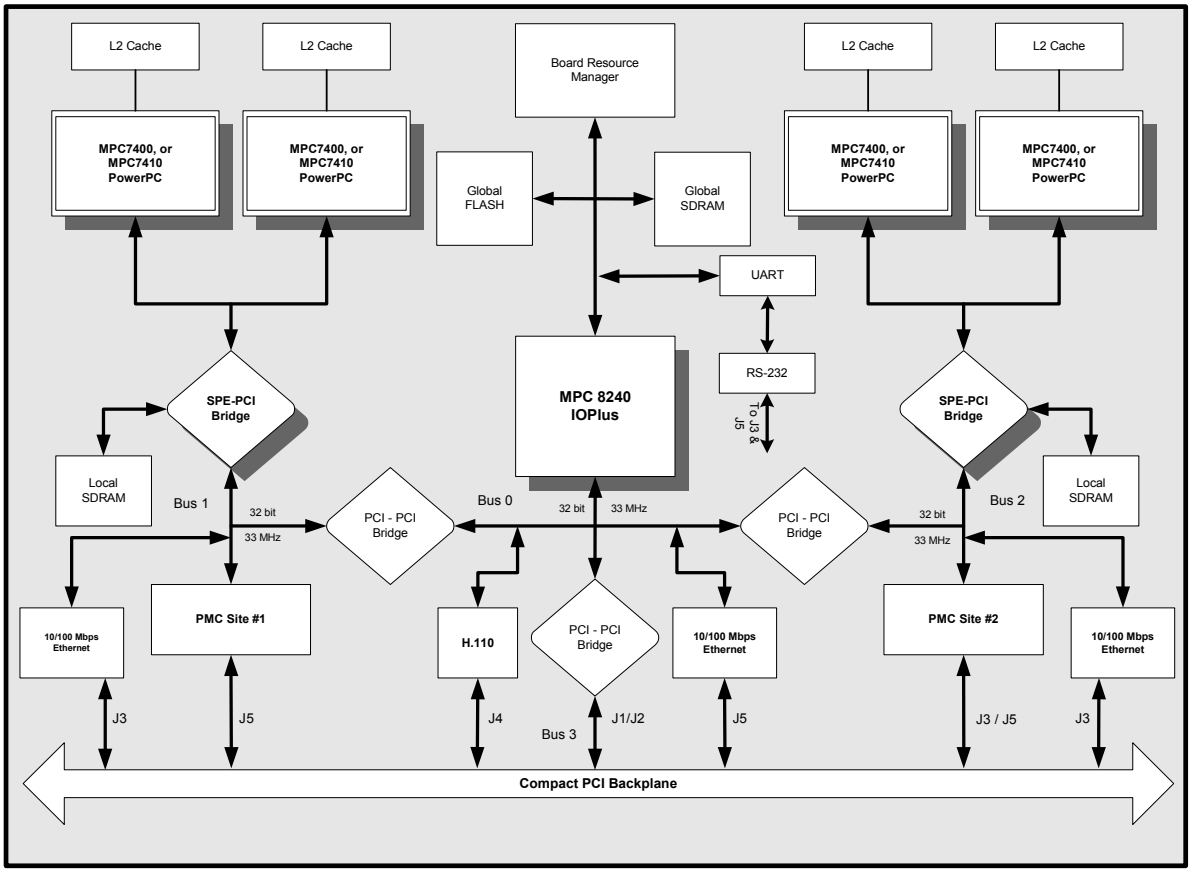


Figure 3.1 - IXA4 Board Architecture

3.2 IOPlus

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

33 MHz 32-bit PCI bus interface, including PCI arbitration
100 MHz 64-bit SDRAM controller, supporting up to 1 GByte of external memory
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
Support for 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 cPCI bus interface
Initialization of all PCI bridges
Initialization of PMC modules
Loading and configuration of firmware devices
Command and status for cPCI bus 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 IXA4 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 IXA4 comes with MPC7400/7410 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 7 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 IXA4, 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 33 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 PCI Local Bus

The IXA4 contains four local PCI buses each isolated by PCI bridges. The Primary PCI bus, PCI bus #0 in Figure 3.1, is a 32 bit, 33MHz 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 cPCI bus. 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 IXA4 exist in PCI space, the PCI memory map of the board plays an important role in the programming and data movement of the IXA4. This map is presented in Chapter 4.

Major features of the PCI-PCI bridges:

- Three 33 MHz, 32-bit PCI-PCI bridges providing bus isolation and transaction forwarding
- Accepts Type 0 and Type 1 configuration accesses
- Supports a uniform linear system address map (with the exception of the 21554 as discussed earlier)
- Performs clocking, reset, and arbitration for the secondary bus

3.6 Board Resource Manager

The Board Resource Manager is an FPGA that provides interrupt and LED access for the IXA4. 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 following are the capabilities provided by the Board Resource Manager:

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

The Board Resource Manager is controlled by a group of addressable registers that are defined in Chapter 4.

Interrupts

One function of the board resource manager is to multiplex interrupts from the many sources supported by the IXA4 board and selectively route them to one or more of the processors on the board. The interrupt multiplexor function is controlled by the mask registers defined in Chapter 4. Each processor (all four SPEs and the IOPlus) has an interrupt mask register. On power up the mask register masks out all interrupts so that no interrupts will be seen at the processors. The application code sets up the mask register so that when a particular interrupt becomes active it will interrupt one or more processors.

The interrupts, including the user interrupts on J3 and J5, are not latched inside the Board Resource Manager. Therefore they must be either latched at the source (as is the case with all on board interrupts), or active long enough to guarantee that the interrupted processor can enter its Interrupt Service Routine (ISR) and read the register within the Board Resource Manager that identifies the source of the interrupt. If the interrupt goes away before this happens, the processor will not be able to identify any interrupt source. The time required for the processor to do this is very dependent upon the application software.

One interrupt is provided from the cPCI backplane bridge (intel 21554). This interrupt can be asserted through software by writing to the secondary IRQ or I2O registers in the 21554 (see 21554 documentation for further details). This is **the only way** for another board in the system to generate an interrupt to the IXA4 over the backplane.

User I/O interrupts from the backplane are active low.

3.7 Board Semaphores

Board semaphores are provided via secure locations in global memory. Semaphores can only be set when they are zero. A non-zero value in a semaphore is cleared by writing the value to the semaphore. Logic compares the data being written to the current value and if they are the same clears the semaphore. 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.

Dy 4 Systems does not provide library functions to gain control of and release hardware semaphores, but instead supplies test and set functions that can be used for the same purpose.

3.8 Global Memory

The IXA4 has 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, cPCI bus, 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.

3.9 SPE Local Memory

The IXA4 can be populated with up to 256 Mbytes of SDRAM per SPE cluster. The SDRAM bus runs at 100MHz. IXA4 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 accesses 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 transactions to local memory may be continually retried because the PCI transaction cannot break in. This is especially true if both processors are polling on locations in separate pages of the SDRAM. If the polling cannot be avoided, PCI bus snooping may be turned on to elevate the priority of the PCI transaction. However, with snooping on, an extra address cycle only occurs on the SPE local 60x bus for every PCI transaction which 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.

3.10 FLASH Memory

The IXA4 provides up to 32 Mbytes of 90 ns FLASH memory that is connected to the IOPlus. It is used to store initialization code, IXA4 configuration data, and user application code. FLASH memory can be accessed from the IOPlus, the SPEs, and from a 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.

3.11 PMC Sites

The IXA4 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 2.3, 29 March 2002.
Draft Processor PMC Standard (VITA 32-199x) Draft 0.5, May 9, 2002.
Draft Standard Physical and Environmental Layers for PCI Mezzanine Cards: PMC; IEEE P1386.1-2001.
Draft Standard for a Common Mezzanine Card Family: CMC; IEEE P1386-2001.

PMCs on the IXA4 board use 5 V signaling only. A 5 V keying pin is provided to prevent you from inadvertently plugging in the wrong type of PMC.

When a PMC manufacturer does not provide the required keying hole, the keying pin on the IXA4 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 IXA4 baseboard as well as the PMC.

! WARNING: Each PMC site is configured for 5 V signaling levels only! Do not install a 3.3 V PMC on a 5 V site. Installing incompatible PMCs can permanently damage the IXA4 baseboard and/or the PMC.

The IXA4 supports front panel user I/O and backplane user I/O through the J3 and J5 connectors. The PMC routing to J3 and J5 was changed on Revision C and higher IXA4 boards. On Revision C and higher, backplane I/O for the PMC Site 1 is routed out the J5 connector and backplane I/O for the PMC Site 2 is routed out the J3 and J5 connectors.

On Revisions A and B, backplane I/O for the PMC Site 1 is routed out the J3 connector zand backplane I/O for the PMC Site 2 is routed out the J5 connector.

Table 3.1

Table 3.1 - PMC to cPCI connector mapping

	Revisions A and B		Revisions C and Higher	
PMC Jn4 Connector	PMC Site 1 (cPCI bus Pin)	PMC Site 2 (cPCI bus Pin)	PMC Site 1 (cPCI bus Pin)	PMC Site 2 (cPCI bus Pin)
1	J3E-13	J5E-13	J5D-20	J5D-4
3	J3D-13	J5D-13	J5E-20	J5E-4
2	J3C-13	J5C-13	J5A-20	J5A-4
4	J3B-13	J5B-13	J5B-20	J5B-4
5	J3A-13	J5A-13	J5D-19	J5D-3
7	J3E-12	J5E-12	J5E-19	J5E-3
6	J3D-12	J5D-12	J5A-19	J5A-3
8	J3C-12	J5C-12	J5B-19	J5B-3
9	J3B-12	J5B-12	J5D-18	J5D-2
11	J3A-12	J5A-12	J5E-18	J5E-2
10	J3E-11	J5E-11	J5A-18	J5A-2
12	J3D-11	J5D-11	J5B-18	J5B-2
13	J3C-11	J5C-11	J5D-17	J5D-1
15	J3B-11	J5B-11	J5E-17	J5E-1
14	J3A-11	J5A-11	J5A-17	J5A-1
16	J3E-10	J5E-10	J5B-17	J5B-1
17	J3D-10	J5D-10	J5D-16	J3D-12
19	J3C-10	J5C-10	J5E-16	J3E-12
18	J3B-10	J5B-10	J5A-16	J3A-12
20	J3A-10	J5A-10	J5B-16	J3B-12
21	J3E-9	J5E-9	J5D-15	J3D-11
23	J3D-9	J5D-9	J5E-15	J3E-11
22	J3C-9	J5C-9	J5A-15	J3A-11
24	J3B-9	J5B-9	J5B-15	J3B-11
25	J3A-9	J5A-9	J5D-14	J3D-10
27	J3E-8	J5E-8	J5E-14	J3E-10
26	J3D-8	J5D-8	J5A-14	J3A-10
28	J3C-8	J5C-8	J5B-14	J3B-10
29	J3B-8	J5B-8	J5D-13	J3D-9
31	J3A-8	J5A-8	J5E-13	J3E-9
30	J3E-7	J5E-7	J5A-13	J3A-9
32	J3D-7	J5D-7	J5B-13	J3B-9
33	J3C-7	J5C-7	J5D-12	J3D-8
35	J3B-7	J5B-7	J5E-12	J3E-8
34	J3A-7	J5A-7	J5A-12	J3A-8
36	J3E-6	J5E-6	J5B-12	J3B-8
37	J3D-6	J5D-6	J5D-11	J3D-7
39	J3C-6	J5C-6	J5E-11	J3E-7
38	J3B-6	J5B-6	J5A-11	J3A-7
40	J3A-6	J5A-6	J5B-11	J3B-7

Table 3.1 - PMC to cPCI connector mapping (cont.)

	Revisions A and B		Revisions C and Higher	
PMC Jn4 Connector	PMC Site 1 (cPCI bus Pin)	PMC Site 2 (cPCI bus Pin)	PMC Site 1 (cPCI bus Pin)	PMC Site 2 (cPCI bus Pin)
41	J3E-5	J5E-5	J5D-10	J3D-6
43	J3D-5	J5D-5	J5E-10	J3E-6
42	J3C-5	J5C-5	J5A-10	J3A-6
44	J3B-5	J5B-5	J5B-10	J3B-6
45	J3A-5	J5A-5	J5D-9	J3D-5
47	J3E-4	J5E-4	J5E-9	J3E-5
46	J3D-4	J5D-4	J5A-9	J3A-5
48	J3C-4	J5C-4	J5B-9	J3B-5
49	J3B-4	J5B-4	J5D-8	J3D-4
51	J3A-4	J5A-4	J5E-8	J3E-4
50	J3E-3	J5E-3	J5A-8	J3A-4
52	J3D-3	J5D-3	J5B-8	J3B-4
53	J3C-3	J5C-3	J5D-7	J3D-3
55	J3B-3	J5B-3	J5E-7	J3E-3
54	J3A-3	J5A-3	J5A-7	J3A-3
56	J3E-2	J5E-2	J5B-7	J3B-3
57	J3D-2	J5D-2	J5D-6	J3D-2
59	J3C-2	J5C-2	J5E-6	J3E-2
58	J3B-2	J5B-2	J5A-6	J3A-2
60	J3A-2	J5A-2	J5B-6	J3B-2
61	J3E-1	J5E-1	J5D-5	J3D-1
63	J3D-1	J5D-1	J5E-5	J3E-1
62	J3C-1	J5C-1	J5A-5	J3A-1
64	J3B-1	J5B-1	J5B-5	J3B-1

The PMC BUSMODE[4:2] signals are hardwired on the IXA4 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 IXA4 receives the BUSMODE1 signals from each PMC but does not check its value.

The IXA4 provides support for the PPMC standard. The PPMC support includes: frequency signaling capability, 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 IXA4 board:

The MONARCH# signal is on Jn2 pin 64. The IXA4 baseboard is the Monarch. The MONARCH# signals on each PMC site are not connected and allowed to float high

M66EN signal is on Jn2 pin 47. This pin is grounded on the IXA4 board indicating 33 MHz PCI bus operation.

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

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

The GNTB# signal is 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 on Jn2 pin 60. This signal is an open drain output from the PMC. When asserted by the PMC, the IXA4 will perform a board reset (same as pushing the reset switch on the IXA4 board).

The EREADY signal is 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 IXA4 configuration software will not perform enumeration on the respective PMC until the PMC releases this signal.

The IXA4 provides support for an Optional Second PCI Agent. Non-monarch PPMCs may include an optional second PCI agent

Power Requirements

The IXA4 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 is configured for 5 V signaling. The maximum power dissipation allowed for each PMC site is 10 Watts.

Table 3.2 - PMC Max. Current loads

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

3.12 cPCI bus Interface

The IXA4 is fully compliant with the cPCI specification. The board can only function as a cPCI peripheral board (i.e., it cannot act as the system controller). A major change was made to the pinout of J3 and J5 on Rev C of the IXA4 board. Tables 3.3, 3.4, and 3.67 give the signal mappings for J1, J2 and J4 respectively, for all revisions on the IXA4. Tables 3.5 and 3.7 give the pinout for J3 and J5 respectively, for revision C and higher IXA4 boards. Tables 3.8 and 3.9 show the pinouts of J3 and J5 for Revisions A and B of the IXA4 board.

Table 3.3 – J1 Connector Definitions

Pin	Z	A	B	C	D	E	F
25	GND	5V	RSV	ENUM#	3.3V	5V	GND
24	GND	AD[1]	5V	V(I/O)	AD[0]	RSV	GND
23	GND	3.3V	AD[4]	AD[3]	5V	AD[2]	GND
22	GND	AD[7]	GND	3.3V	AD[6]	AD[5]	GND
21	GND	3.3V	AD[9]	AD[8]	M66EN	C/BE[0]#	GND
20	GND	AD[12]	GND	V(I/O)	AD[11]	AD[10]	GND
19	GND	3.3V	AD[15]	AD[14]	GND	AD[13]	GND
18	GND	SERR#	GND	3.3V	PAR	C/BE[1]#	GND
17	GND	3.3V	RSV	RSV	GND	PERR#	GND
16	GND	DEVSEL#	GND	V(I/O)	STOP#	RSV	GND
15	GND	3.3V	FRAME#	IRDY#	BD_SEL#	TRDY#	GND
12-14	Key Area						
11	GND	AD[18]	AD[17]	AD[16]	GND	C/BE[2]#	GND
10	GND	AD[21]	GND	3.3V	AD[20]	AD[19]	GND
9	GND	C/BE[3]#	IDSEL	AD[23]	GND	AD[22]	GND
8	GND	AD[26]	GND	V(I/O)	AD[25]	AD[24]	GND
7	GND	AD[30]	AD[29]	AD[28]	GND	AD[27]	GND
6	GND	REQ#	PCI_PRES	3.3V	CLK	AD[31]	GND
5	GND	RSV	RSV	RST#	GND	GNT#	GND
4	GND	RSV	HEALTHY#	V(I/O)	INTP	RSV	GND
3	GND	INTA#	RSV	RSV	5V	RSV	GND
2	GND	RSV	5V	RSV	RSV	RSV	GND
1	GND	5V	-12V	RSV	+12V	5V	GND

Table 3.4 – J2 Connector Definitions

Pin	Z	A	B	C	D	E	F
22	GND	GA4	GA3	GA2	GA1	GA0	GND
21	GND	RSV	RSV	RSV	RSV	RSV	GND
20	GND	RSV	RSV	RSV	GND	RSV	GND
19	GND	RSV	RSV	RSV	RSV	RSV	GND
18	GND	RSV	RSV	RSV	GND	RSV	GND
17	GND	RSV	GND	RSV	RSV	RSV	GND
16	GND	RSV	RSV	RSV	GND	RSV	GND
15	GND	RSV	GND	RSV	RSV	RSV	GND
14	GND	RSV	RSV	RSV	GND	RSV	GND
13	GND	RSV	GND	V(I/O)	RSV	RSV	GND
12	GND	RSV	RSV	RSV	GND	RSV	GND
11	GND	RSV	GND	V(I/O)	RSV	RSV	GND
10	GND	RSV	RSV	RSV	GND	RSV	GND
9	GND	RSV	GND	V(I/O)	RSV	RSV	GND
8	GND	RSV	RSV	RSV	GND	RSV	GND
7	GND	RSV	GND	V(I/O)	RSV	RSV	GND
6	GND	RSV	RSV	RSV	GND	RSV	GND
5	GND	RSV	GND	V(I/O)	RSV	RSV	GND
4	GND	V(I/O)	RSV	RSV	GND	RSV	GND
3	GND	RSV	GND	RSV	RSV	RSV	GND
2	GND	RSV	RSV	RSV	RSV	RSV	GND
1	GND	RSV	GND	RSV	RSV	RSV	GND

Table 3.5 – J3 Connector Definitions

Pin	Z	A	B	C	D	E	F
19	GND	SGA4	SGA3	SGA2	SGA1	SGA0	GND
18	GND	1TX+	1TX-	GND			GND
17	GND	1RX+	1RX-	GND			GND
16	GND	2TX+	2TX-	GND			GND
15	GND	2RX+	2RX-	GND			GND
14	GND	1TRMPLN	2TRMPLN	USER_INT0	USER_INT1	RPMPRES	GND
13	GND	RTSB_232	CTSB_232	RXB_232	TXB_232	CDB_232	GND
12	GND	PMC2-16	PMC2-20	GND	PMC2-17	PMC2-19	GND
11	GND	PMC2-22	PMC2-24	GND	PMC2-21	PMC2-23	GND
10	GND	PMC2-26	PMC2-28	GND	PMC2-25	PMC2-27	GND
9	GND	PMC2-30	PMC2-32	GND	PMC2-29	PMC2-31	GND
8	GND	PMC2-34	PMC2-36	GND	PMC2-33	PMC2-35	GND
7	GND	PMC2-38	PMC2-40	GND	PMC2-37	PMC2-39	GND
6	GND	PMC2-42	PMC2-44	GND	PMC2-41	PMC2-43	GND
5	GND	PMC2-46	PMC2-48	GND	PMC2-45	PMC2-47	GND
4	GND	PMC2-50	PMC2-52	GND	PMC2-49	PMC2-51	GND
3	GND	PMC2-54	PMC2-56	GND	PMC2-53	PMC2-55	GND
2	GND	PMC2-58	PMC2-60	GND	PMC2-57	PMC2-59	GND
1	GND	PMC2-62	PMC2-64	GND	PMC2-61	PMC2-63	GND

Table 3.6 – J4 Connector Definitions

Pin	Z	A	B	C	D	E	F
25	NP	SGA4	SGA3	SGA2	SGA1	SGA0	FP
24	NP	GA4	GA3	GA2	GA1	GA0	FP
23	NP	+12V	RSV	CT_EN	-12V	CT_MC	FP
22	NP	RSV	RSV	RSV	RSV	RSV	FP
21	NP	RSV	RSV	RSV	RSV	RSV	FP
20	NP	NP	NP	NP	NP	NP	NP
19	NP	NP	NP	NP	NP	NP	NP
18	NP	RSV	RSV	RSV	RSV	RSV	NP
17	NP	NP	NP	NP	NP	NP	NP
16	NP	NP	NP	NP	NP	NP	NP
15	NP	RSV	RSV	RSV	RSV	RSV	NP
12-14	Key Area						
11	NP	CT_D29	CT_D30	CT_D31	V(I/O)	/CT_FRAME_A	GND
10	NP	CT_D27	3.3V	CT_D28	5V	/CT_FRAME_B	GND
9	NP	CT_D24	CT_D25	CT_D26	GND	FR_COMP#	GND
8	NP	CT_D21	CT_D22	CT_D23	5V	CT_C8_A	GND
7	NP	CT_D19	5V	CT_D20	GND	CT_C8_B	GND
6	NP	CT_D16	CT_D17	CT_D18	GND	CT_NETREF_2	GND
5	NP	CT_D13	CT_D14	CT_D15	3.3V	CT_NETREF_1	GND
4	NP	CT_D11	5V	CT_D12	3.3V	SCLK	GND
3	NP	CT_D8	CT_D9	CT_D10	GND	SCLK-D	GND
2	NP	CT_D4	CT_D5	CT_D6	CT_D7	GND	GND
1	NP	CT_D0	3.3V	CT_D1	CT_D2	CT_D3	GND

Table 3.7 – J5 Connector Definitions

Pin	Z	A	B	C	D	E	F
22	GND	RTSA_232	CTSA_232	RXA_232	TXA_232	CDA_232	GND
21	GND	0TX+	0TX-	0TRMPLN	0RX+	0RX-	GND
20	GND	PMC1-2	PMC1-4	GND	PMC1-1	PMC1-3	GND
19	GND	PMC1-6	PMC1-8	GND	PMC1-5	PMC1-7	GND
18	GND	PMC1-10	PMC1-12	GND	PMC1-9	PMC1-11	GND
17	GND	PMC1-14	PMC1-16	GND	PMC1-13	PMC1-15	GND
16	GND	PMC1-18	PMC1-20	GND	PMC1-17	PMC1-19	GND
15	GND	PMC1-22	PMC1-24	GND	PMC1-21	PMC1-23	GND
14	GND	PMC1-26	PMC1-28	GND	PMC1-25	PMC1-27	GND
13	GND	PMC1-30	PMC1-32	GND	PMC1-29	PMC1-31	GND
12	GND	PMC1-34	PMC1-36	GND	PMC1-33	PMC1-35	GND
11	GND	PMC1-38	PMC1-40	GND	PMC1-37	PMC1-39	GND
10	GND	PMC1-42	PMC1-44	GND	PMC1-41	PMC1-43	GND
9	GND	PMC1-46	PMC1-48	GND	PMC1-45	PMC1-47	GND
8	GND	PMC1-50	PMC1-52	GND	PMC1-49	PMC1-51	GND
7	GND	PMC1-54	PMC1-56	GND	PMC1-53	PMC1-55	GND
6	GND	PMC1-58	PMC1-60	GND	PMC1-57	PMC1-59	GND
5	GND	PMC1-62	PMC1-64	GND	PMC1-61	PMC1-63	GND
4	GND	PMC2-2	PMC2-4	GND	PMC2-1	PMC2-3	GND
3	GND	PMC2-6	PMC2-8	GND	PMC2-5	PMC2-7	GND
2	GND	PMC2-10	PMC2-12	GND	PMC2-9	PMC2-11	GND
1	GND	PMC2-14	PMC2-16	GND	PMC2-13	PMC2-15	GND

Table 3.8 – J3 Connector Definitions for Revisions A and B

Pin	Z	A	B	C	D	E	F
19	GND	CDA		CDB		2RD-	GND
18	GND	CTSA		CTSB		2RD+	GND
17	GND	RTSA		RTSB		2TD-	GND
16	GND	RDA		RDB		2TD+	GND
15	GND	TDA		TDB		USER_INT1	GND
14	GND	3.3V	3.3V	3.3V	5V	5V	GND
13	GND	PMC1-5	PMC1-4	PMC1-3	PMC1-2	PMC1-1	GND
12	GND	PMC1-10	PMC1-9	PMC1-8	PMC1-7	PMC1-6	GND
11	GND	PMC1-15	PMC1-14	PMC1-13	PMC1-12	PMC1-11	GND
10	GND	PMC1-20	PMC1-19	PMC1-18	PMC1-17	PMC1-16	GND
9	GND	PMC1-25	PMC1-24	PMC1-23	PMC1-22	PMC1-21	GND
8	GND	PMC1-30	PMC1-29	PMC1-28	PMC1-27	PMC1-26	GND
7	GND	PMC1-35	PMC1-34	PMC1-33	PMC1-32	PMC1-31	GND
6	GND	PMC1-40	PMC1-39	PMC1-38	PMC1-37	PMC1-36	GND
5	GND	PMC1-45	PMC1-44	PMC1-43	PMC1-42	PMC1-41	GND
4	GND	PMC1-50	PMC1-49	PMC1-48	PMC1-47	PMC1-46	GND
3	GND	PMC1-55	PMC1-54	PMC1-53	PMC1-52	PMC1-51	GND
2	GND	PMC1-60	PMC1-59	PMC1-58	PMC1-57	PMC1-56	GND
1	GND	VIO	PMC1-64	PMC1-63	PMC1-62	PMC1-61	GND

Table 3.9 – J5 Connector Definitions for Revisions A and B

Pin	Z	A	B	C	D	E	F
22	GND	GTCK	KTCK	PTCK	PCHKSTPO	1TD+	GND
21	GND	GTMS	KTMS	PTMS	PCHKSTPI	1TD-	GND
20	GND	GTDI	KTDI	PTDI	PQCK	1RD+	GND
19	GND	GTDO	KTDO	PTDO	PQREQ	1RD-	GND
18	GND	GTDO2	KTRST	PTRST	PBRST	0TD+	GND
17	GND	GTCK_R	JSRESET	PSRESET	USER_INT0	0TD-	GND
16	GND	GTCK_C	JHRESET	PHRESET	XTCK	0RD+	GND
15	GND	GTRST	XTDO	XTDI	XTMS	0RD-	GND
14	GND	IX RSVD	IX RSVD	IX RSVD	IX RSVD	IX RSVD	GND
13	GND	PMC2-5	PMC2-4	PMC2-3	PMC2-2	PMC2-1	GND
12	GND	PMC2-10	PMC2-9	PMC2-8	PMC2-7	PMC2-6	GND
11	GND	PMC2-15	PMC2-14	PMC2-13	PMC2-12	PMC2-11	GND
10	GND	PMC2-20	PMC2-19	PMC2-18	PMC2-17	PMC2-16	GND
9	GND	PMC2-25	PMC2-24	PMC2-23	PMC2-22	PMC2-21	GND
8	GND	PMC2-30	PMC2-29	PMC2-28	PMC2-27	PMC2-26	GND
7	GND	PMC2-35	PMC2-34	PMC2-33	PMC2-32	PMC2-31	GND
6	GND	PMC2-40	PMC2-39	PMC2-38	PMC2-37	PMC2-36	GND
5	GND	PMC2-45	PMC2-44	PMC2-43	PMC2-42	PMC2-41	GND
4	GND	PMC2-50	PMC2-49	PMC2-48	PMC2-47	PMC2-46	GND
3	GND	PMC2-55	PMC2-54	PMC2-53	PMC2-52	PMC2-51	GND
2	GND	PMC2-60	PMC2-59	PMC2-58	PMC2-57	PMC2-56	GND
1	GND	RPMPRES	PMC2-64	PMC2-63	PMC2-62	PMC2-61	GND

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. 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 IXA4 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 referring 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 IXA4 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 IXA4 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	PCI Address Generated
00000000 – 07FFFFFF	128 MB	Global Memory	System Memory	None
08000000 – 3FFFFFFF	896 MB	Reserved		None
40000000 – 7FFFFFFF	1 GB	IOPlus Reserved		None
80000000 – 9FFFFFFF	512 MB	PMC 1 and PCI A/B Devices	PCI Memory Space	80000000 – 9FFFFFFF
A0000000 – AFFFFFFF	256 MB	SPE AB SDRAM		A0000000 – AFFFFFFF
B0000000 – B5EFFFFF	95 MB	Reserved		B0000000 – B5EFFFFF
B5F00000 – B5FFFFFF	1 MB	SPE AB Embedded Utilities Memory Block		B5F00000 – B5FFFFFF
B6000000 – BFFFFFFF	160 MB	Dy 4 Systems Reserved		00000000 – 07FFFFFF
C0000000 – DFFFFFFF	512 MB	PMC 2 and PCI C/D Devices		C0000000 – DFFFFFFF
E0000000 – EFFFFFFF	256 MB	SPE CD SDRAM		E0000000 – EFFFFFFF
F0000000 – F5EFFFFF	95 MB	Reserved		F0000000 – F5EFFFFF
F5F00000 – F5FFFFFF	1 MB	SPE CD Embedded Utilities Memory Block		F5F00000 – F5FFFFFF
F6000000 – F6FFFFFF	16 MB	Reserved		F6000000 – F6FFFFFF
F7000000 – F7FFFFFF	16 MB	IOPlus Ethernet		F7000000 – F7FFFFFF
F8000000 – FBFFFFFF	64 MB	CPCI Upstream Memory Window		F8000000 – FBFFFFFF
FC000000 – FCFFFFFF	16 MB	cPCI Control/Status Registers		FC000000 – FCFFFFFF
FD000000 – FD0FFFFF	1 MB	IOPlus Embedded Utilities Memory Block	Available ONLY to the IOPlus	
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	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	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	PCI Address Generated
00000000 – 0FFFFFFF	256 MB	SPE AB SDRAM	Local System Memory	None
10000000 – 3FFFFFFF	768 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 and PCI A/B Devices	PCI Memory Space	80000000 – 9FFFFFFF
A0000000 – A7FFFFFF	128 MB	Illegal Block		None
A8000000 – B5EFFFFF	223 MB	Reserved		A8000000 – B5EFFFFF
B5F00000 – B7FFFFFF	33 MB	Illegal Block		None
B8000000 – BFFFFFFF	128 MB	Global SDRAM (must use SPE_PCI Bridge Outbound Translation Register to translate address to global memory)		00000000 – 07FFFFFF (After Boot and Initialization)
C0000000 – DFFFFFFF	512 MB	PMC Site 2 and PCI C/D Devices		C0000000 – DFFFFFFF
E0000000 – EFFFFFFF	256 MB	SPE CD SDRAM		E0000000 – EFFFFFFF
F0000000 – F5EFFFFF	95 MB	Reserved		F0000000 – F5EFFFFF
F5F00000 – F5FFFFFF	1 MB	SPE CD Embedded Utilities Memory Block		F5F00000 – F5FFFFFF
F6000000 – F6FFFFFF	16 MB	Reserved		F6000000 – F6FFFFFF
F7000000 – F7FFFFFF	16 MB	IOPlus Ethernet		F7000000 – F7FFFFFF
F8000000 – FBFFFFFF	64 MB	cPCI Upstream Memory Window		F8000000 – FBFFFFFF
FC000000 – FCFFFFFF	16 MB	cPCI Control/Status 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	FD100000 – FDFFFFFF
FE000000 – FEBFFFFF	12 MB	PCI I/O Space	PCI I/O Space	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	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	40000 – 41FFF (At Boot Time Only; This range is Programmable in the SPE-PCI Bridge)
FFF02000 – FFFFFFFF	1016 KB	Not Used		FFF02000 – FFFFFFFF

Table 4.3 - SPE C/D Cluster Memory Map

SPE Local Address	Size	Description	Local Description	PCI Address Generated
00000000 – 0FFFFFFF	256 MB	SPE CD SDRAM	Local System Memory	None
10000000 – 3FFFFFFF	768 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 and PCI A/B Devices	PCI Memory Space	80000000 – 9FFFFFFF
A0000000 – AFFFFFFF	256 MB	SPE AB SDRAM		A0000000 – AFFFFFFF
B0000000 – B5EFFFFF	95 MB	Reserved		B0000000 – B5EFFFFF
B5F00000 – B5FFFFFF	1 MB	SPE AB Embedded Utilities Memory Block		B5F00000 – B5FFFFFF
B6000000 – B7FFFFFF	32 MB	Reserved		B6000000 – B7FFFFFF
B8000000 – BFFFFFFF	128 MB	Global SDRAM (must use SPE-PCI Bridge Outbound Translation Register to translate address to global memory)		00000000 – 07FFFFFF (After Boot and Initialization)
C0000000 – DFFFFFFF	512 MB	PMC Site 2 and PCI C/D Devices		C0000000 – DFFFFFFF
E0000000 – E7FFFFFF	128 MB	Illegal Block		
E8000000 – F5EFFFFF	223 MB	Reserved		E8000000 – F5EFFFFF
F5F00000 – F5FFFFFF	1 MB	Illegal Block		
F6000000 – F6FFFFFF	16 MB	Reserved		F6000000 – F6FFFFFF
F7000000 – F7FFFFFF	16 MB	IOPlus Ethernet		F7000000 – F7FFFFFF
F8000000 – FBFFFFFF	64 MB	cPCI Upstream Memory Window		F8000000 – FBFFFFFF
FC000000 – FCFFFFFF	16 MB	cPCI Control/Status 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	FD100000 – FDFFFFFF
FE000000 – FEBFFFFF	12 MB	PCI I/O Space	PCI I/O Space	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	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	40000 – 41FFF (At Boot Time Only; This Range is Programmable in the SPE-PCI Bridge)
FFF02000 – FFFFFFFF	1016 KB	Not Used		FFF02000 – FFFFFFFF

Table 4.4 - PCI Memory Map

PCI Address Range	Size	Description
00000000 - 07FFFFFF	128 MB	Global Memory
08000000 - 7FFFFFFF	1920 MB	Not Used
80000000 - 9FFFFFFF	512 MB	SPE AB PMC
A0000000 - AFFFFFF	256 MB	SPE AB SDRAM
B0000000 - B5EFFFFF	95 MB	Reserved
B5F00000 - B5FFFFFF	1 MB	SPE AB Embedded Utilities Memory Block
B6000000 - BFFFFFFF	160 MB	Not Used
C0000000 - DFFFFFFF	512 MB	SPE CD PMC
E0000000 - EFFFFFF	256 MB	SPE CD SDRAM
F0000000 - F5EFFFFF	95 MB	Reserved
F5F00000 - F5FFFFFF	1 MB	SPE CD Embedded Utilities Memory Block
F6000000 - F6FFFFFF	16 MB	Reserved
F7000000 - F7FFFFFF	16 MB	IOPlus us Ethernet
F8000000 - FBFFFFFF	64 MB	cPCI Upstream Memory Window
FC000000 - FCFFFFFF	16 MB	cPCI Control/Status 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 cPCI bus. The IOPlus does not control the allocation of the global memory area available for the SPE and cPCI bus 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 Reserved	
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 cPCI Memory

Inbound Address Translation

There are three Inbound (downstream) Address Translation Windows. These are used to translate cPCI addresses to IXA4 addresses. These are set up by the IXA4 boot code and should not be modified by the application. These windows are used to map all of global memory and the upper 32 MB of each SPE local memory space. Table 4.6 shows the windows.

The two windows for the SPE clusters show the upper 32 MB of memory for that cluster. Therefore the offset from the board base address changes depending on whether each SPE has 64 MB, 128 MB, or 256 MB of memory.

Table 4.6 – cPCI Inbound Memory Map

Offset from cPCI base address	Window Size	Address size	Privileged / Non-Privileged	Program / Data	Description
0x00000000 – 0x3FFFFFFF	64 MB	A32	Both	Both	Global Memory
0xA2000000 – 0xA3FFFFFF	32 MB	A32	Both	Both	SPE A/B (if cluster has 64 MB of memory)
0xA6000000 – 0xA7FFFFFF	32 MB	A32	Both	Both	SPE A/B (if cluster has 128 MB of memory)
0xAE000000 – 0xAFFFFFFF	32 MB	A32	Both	Both	SPE A/B (if cluster has 256 MB of memory)
0xE2000000 – 0xE3FFFFFF	32 MB	A32	Both	Both	SPE C/D (if cluster has 64 MB of memory)
0xE6000000 – 0xE7FFFFFF	32 MB	A32	Both	Both	SPE C/D (if cluster has 128 MB of memory)
0xEE000000 – 0xEFFFFFFF	32 MB	A32	Both	Both	SPE C/D (if cluster has 256 MB of memory)

Outbound Address Translation

There are also three Outbound (upstream) memory windows. These windows map addresses on the cPCI bus into IXA4 memory space. Two of these windows must be set up by the application code, the third window is reserved and must not be used by application code. Table 4.7 shows the location of these windows.

These windows are 32 MB each and are accessible from any device on the IXA4 PCI address space (IOPlus, SPEs, PMCs, etc.)

Table 4.7 – cPCI Outbound Memory Map

IXA4 PCI Address	Window Size	Address size	Privileged / Non-Privileged	Program / Data	Description
0xF8000000 – 0xF9FFFFFF	32 MB	A32	Both	Both	Outbound window 1
0xFA000000 – 0xFBFFFFFF	32 MB	A32	Both	Both	Outbound window 1

4.6 Board Resource Manager Register Map

The Board Resource Manager occupies 8 MB of the IOPlus and SPE-PCI bridge memory maps in the range FF00_0000 and FF7F_FFFF. Tables 4.8 and 4.9 present the registers available in that address range from the AB cluster's and CD cluster's respective window into that space. Table 4.10 presents the registers available to the IOPlus in that space.

Table 4.8 - Board Resource Manager 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	Reserved
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	Reserved
FF000070	Read/Write	Reserved

Table 4.9 - Board Resource Manager 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	Reserved
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	Reserved
FF000070	Read/Write	Reserved

Table 4.10 - Board Resource Manager Memory Map for the IOPlus

IOPlus Address	Register Type	Description
FF400000	Read Only	IOPlus General Interrupt Status Register
FF400008	Read Only	Reserved
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	Reserved
FF400068	Read/Write	Reserved
FF400070	Read/Write	Reserved
FF400078	Read/Write	Miscellaneous Register
FF400080	Read Only	CPCI 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. Although the Board Resource Manager provides nine interrupt outputs, only five of these are affected by the masks (the other four interrupt outputs are not masked and are connected directly to the IOPlus). Four of the masked outputs go to the SPEs (one per processor) while one of the outputs connects into the IOPlus. Each SPE processor is provided with two INT interrupt mask registers which are accessible locally (i.e. no PCI bus transactions are necessary). The IOPlus processor is also provided with two mask registers.

The interrupt mask registers are shown in Figure 4.1 through Figure 4.3. Figure 4. 1 shows the IOPlus controlled mask registers (which are accessible only to the IOPlus) while Figure 4.2 and Figure 4.3 show the INT interrupt mask registers that are 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.

High Order Register

Bit	15	14	13	12	11	10	09	08
	Reserved	Reserved	0INTA_EN	Ethernet0	UART Channel_B	UART Channel_A	CCD_ INT_EN	CCD_PCI_ EN
Bit	07	06	05	04	03	02	01	00
	CAB_ INT_EN	CAB_PCI_ 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
	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

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

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

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

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

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

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

Figure 4.2 - Interrupt Mask Register 1 Format for SPE processors

Processor X INT 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 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 4.3 - Interrupt Mask Register 2 Format for SPE processors

Table 4.11 - 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 Cluster AB MPC107
CAB_INT_EN	Primary Interrupt from AB SPE-PCI bridge
CCD_PCI_EN	PCI Interrupt from Cluster CD MPC107
CCD_INT_EN	Primary Interrupt from CD SPE-PCI bridge
USER_INT0_EN	User Interrupt 0
USER_INT1_EN	User Interrupt 1
UART Channel A	UART Interrupt on Channel A (IOPlus only)
UART Channel B	UART Interrupt on Channel B (IOPlus only)
Ethernet0	Ethernet on MPC8240 bus (IOPlus only)
EthernetAB	Ethernet on Cluster AB side
EthernetCD	Ethernet on Cluster CD side
0INTA_EN	21554 Doorbell interrupt (IOPlus only)

The definitions of Mask Register bit fields are provided in Table 4.11. 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 and 2 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 originated the interrupt is not evident by the value in the Interrupt Status and Clear Register. See Figure 4.4 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 4.4.

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 4.4 - 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
   - SPE CD:
     - CD General Interrupt Status Register
   - IOPlus:
     - IOPlus General 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 4.12.

Table 4.12 - Relationship of General Interrupt Status Register Fields to Other Status Registers

General Interrupt Status Register Bit Field	Corresponding Status Register to Check
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 read-only.

The registers defined to convey interrupt status are illustrated in the Figures 4.5, 4.6, and 4.7. 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	Reserved

where:

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 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	EthernetAB	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 4.5 - 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	Reserved

where:

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 Status 1/LED C Control Register

Bit	07	06	05	04	03	02	01	00
	LED_C	Reserved	EthernetCD	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
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 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	Reserved

where:

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 Status 1 Register

Bit	07	06	05	04	03	02	01	00
	Reserved	0INTA	Ethernet0	USER INT0	PMC1B/D	PMC1A/C	CAB_PCI	CAB_INT

IOPlus Status 2 Register

Bit	07	06	05	04	03	02	01	00
	Reserved	UART Channel A	UART Channel B	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 4.7 - 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 4.5 and Figure 4.6. 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.

cPCI Geographical Address Register

This 16 bit read only register allows the software to read the COMPACT PCI Geographical Address pins.

The format of the COMPACT PCI Geographical Address Register is shown in Figure 4.8. This register may be accessed only by the IOPlus.

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

Figure 4.8 COMPACT PCI Geographical Address Register Format

GA(4:0) COMPACT PCI Geographical Address pins.
SGA(4:0) COMPACT PCI Shelf Geographical Address pins.

Miscellaneous Register

This 16 bit read only register allows the software to read miscellaneous status information.

The format the Miscellaneous Register is shown in Figure 4.9. This register may be accessed only by the IOPlus.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Not Used														CFGAUX	RPMPRES

Figure 4.9 Miscellaneous Register Format

Chapter 5: Using the IOPlus

5.1 Introduction

This chapter contains information on the generic IOPlus capabilities. Some of these capabilities pertain specifically to a VME bus backplane. Such capabilities are not applicable to the IXA-4 product.

This chapter describes the communication protocols used by the IOPlus to communicate with host software and SPE software (also referred to as external software components). When the IXA4 board boots, an IOPlus runtime kernel is loaded into the IOPlus processor. This code is of minimal size and is not active until it is "awakened" by an interrupt. The IOPlus kernel is used to process commands that it receives from the SPE processors and/or the host. Routines are provided in IXAtools that can be linked into a SPE user application to command the IOPlus kernel. An understanding of the command protocol details discussed in sections 5.2 – 5.8 is not required in order to use the routines provided by IXAtools. However, you will want to refer to the information provided in these sections if you are writing your own interface to the IOPlus.

External software components communicate with the IOPlus by placing one or more command packets into a linked command list. The IOPlus responds to these commands by placing a response packet into a linked list of responses.

5.2 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 Figure 5.1.

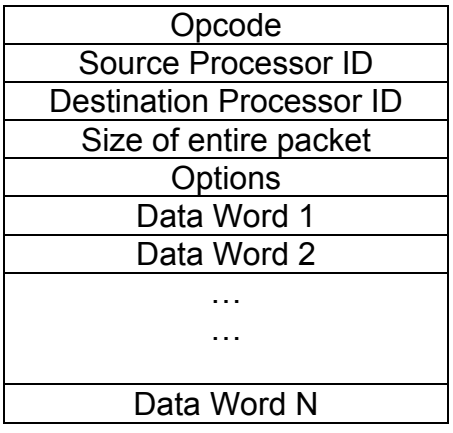


Figure 5.1 - Command Packet

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.

5.3 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 IXA4 board as shown in Table 5.1.

Table 5.1 - 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 5.2.

Absolute processor IDs could be used to discriminate a multi-board IXA4 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 5.2 - 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 5.1, the host processor IDs are assigned immediately after the last SPE processor ID.

5.4 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 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).

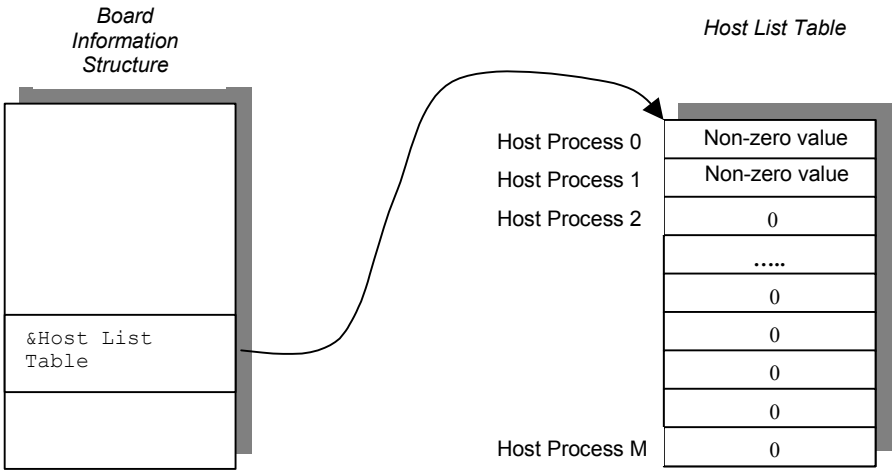


Figure 5.3 - Host Process ID Assignment with Two Host Processes Already Attached

5.5 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 cPCI bus), 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.

5.6 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 5.4 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 5.4 illustrates a "list address table" with all the linked command list entries in global memory. Figure 5.5 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.

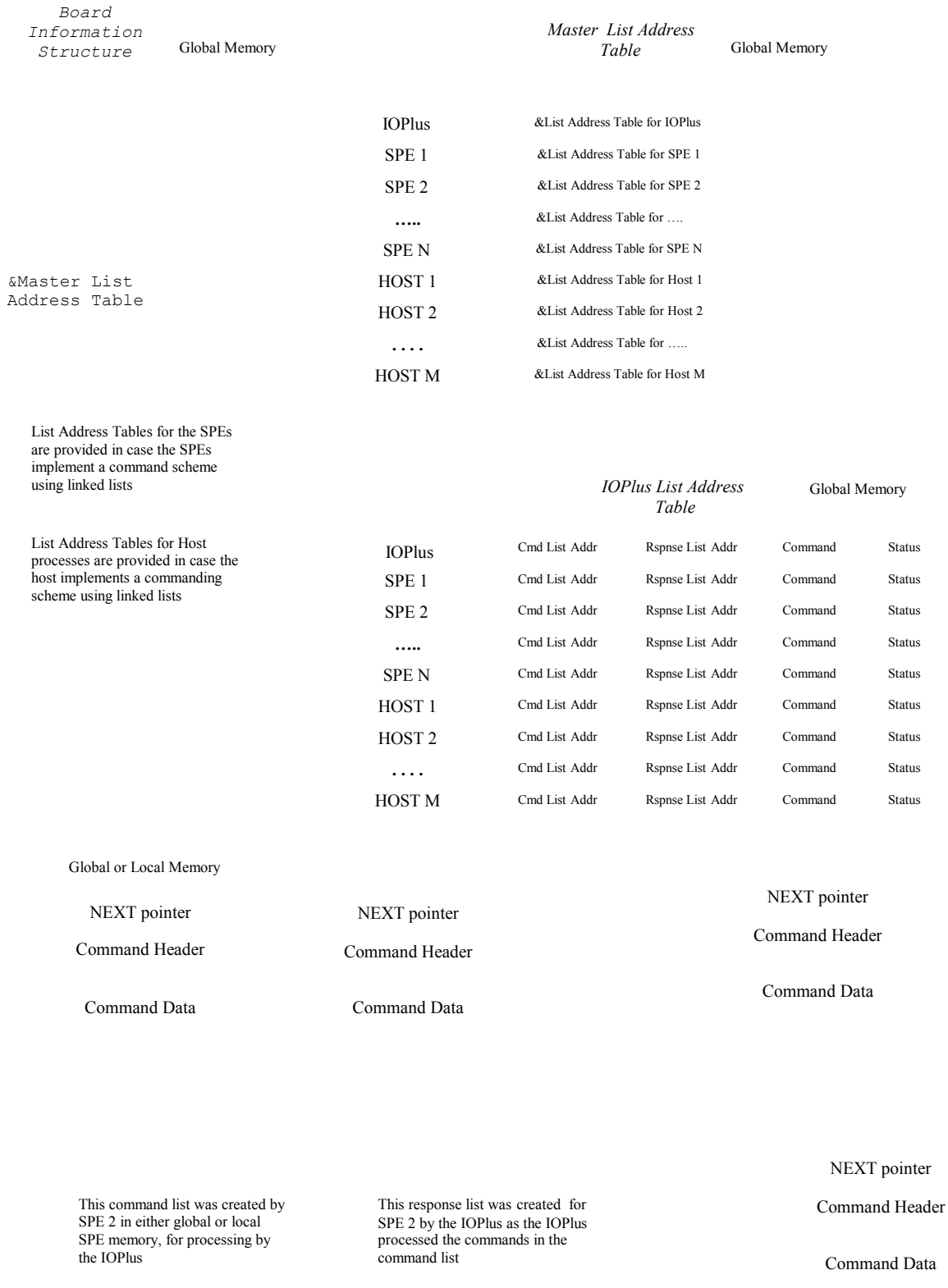


Figure 5.4 - All List Address Table Entries in Global Memory

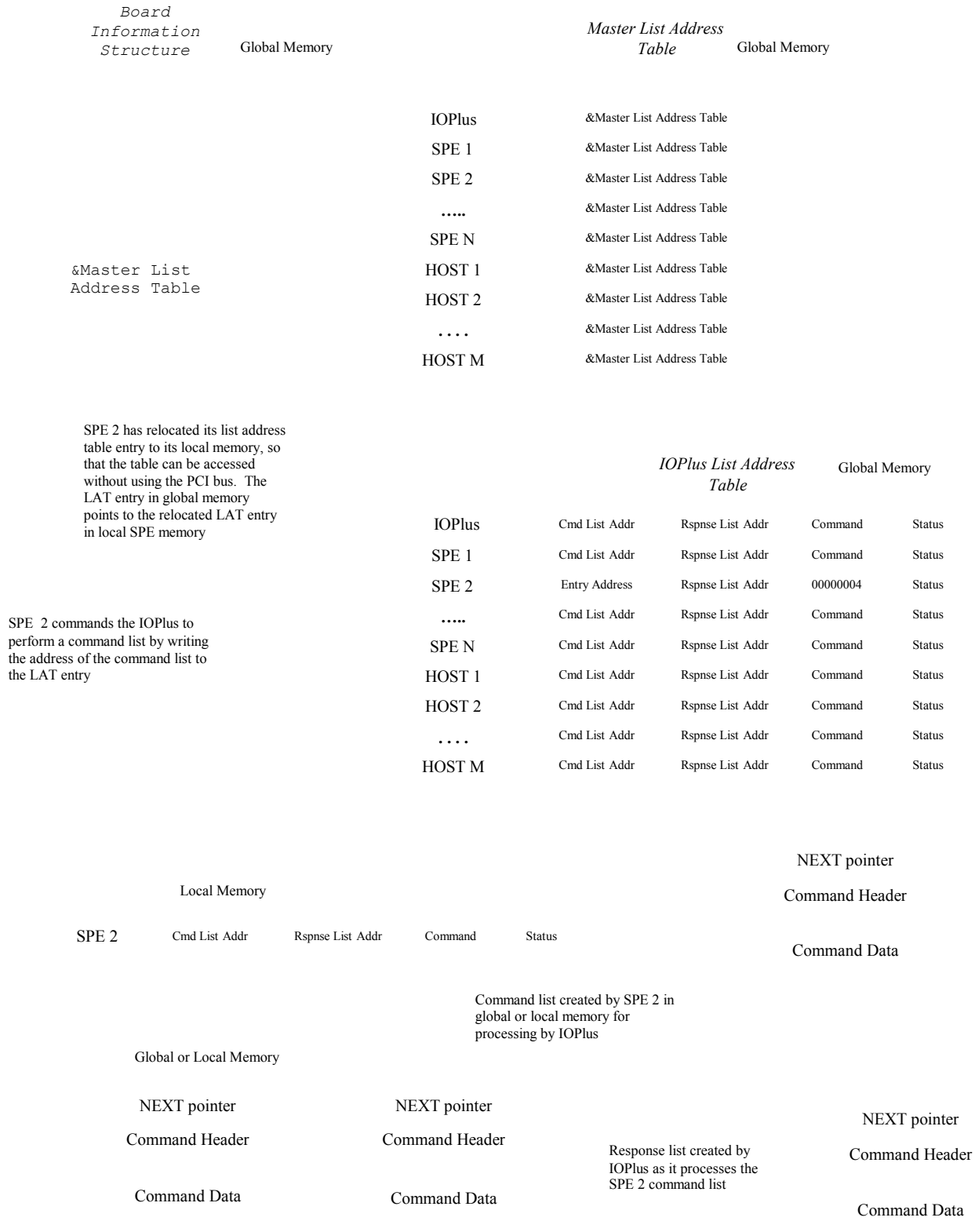


Figure 5.5 - List Address Table Entry Relocated to Local Memory

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 5.4. 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 (e.g. 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 5.5 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.

5.7 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).

5.8 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 Table 5.2.

Table 5.2 - 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 5.3 - 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

5.9 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.

5.10 Semaphore Protocol

The IXA4 hardware provides four 7-bit wide semaphore registers, which may be used for sharing system resources. The algorithm for requesting and releasing a semaphore is as follows:

Requesting a semaphore:

1) Write your processor ID + 1 to the semaphore register, setting the upper bit to 1.
2) Read the semaphore register
3) If you read back the value that you wrote, then you have been granted the semaphore -- utilize the shared resource, then release the semaphore

4) If you read back a value different than the value you wrote, then someone else has been granted the semaphore – you must request the semaphore again (jump to step 1)

Releasing a semaphore:

1) Write zero to the semaphore register with the upper bit set to 1 (i.e. write an 0x80 to the semaphore register).

Semaphore values between 32 and 255 refer to off-board semaphores (note that no protocol is currently defined by the IOPlus for using off-board semaphores).

Table 5.4 shows the uses for each semaphore.

Table 5.4 - Semaphore Assignments

Semaphore ID	Use
0	Protects queue data structures
1	Protects mailbox registers
2	Reserved by Dy 4 Systems
3	Reserved by Dy 4 Systems
4	Reserved by Dy 4 Systems
5	Reserved by Dy 4 Systems
6	Reserved by Dy 4 Systems
7	Reserved by Dy 4 Systems
8	For customer use
. . . .	
15	For customer use

5.11 FLASH Memory Management Protocol

The IXA4 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 5.5.

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 5.5 - 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, 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 the 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 5.6.

Table 5.6 - 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.

5.12 IOPlus Command List

This section describes all commands processed by the IOPlus. 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 the Command/Response Packet Format paragraph 5.3.

These commands can be issued by SPE applications, by using functions in the SPE library. These functions are defined in Chapter 7.

Several commands use a memory section ID as a parameter. Table 5.7 lists the valid IDs for that parameter. Tables 5.8, 5.9 and 5.10 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 5.7 - 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	Reserved		
9	Reserved		
10	SPE A SDRAM memory	X	
11	Reserved		
12	Reserved		
13	Reserved		
14	SPE B SDRAM memory	X	
15	Reserved		
16	Reserved		
17	Reserved		
18	SPE C SDRAM memory	X	
19	Reserved		
20	Reserved		
21	Reserved		
22	SPE D SDRAM memory	X	
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 5.8 - 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 5.9 - 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 5.10 - Format of Board Information Table (Table ID 26)

Field offset	Value
0	Board Type
1	Trace status
2	Board health
3	Product version
4	Initialization 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	Reserved
37 – 40	PMC (0 – 3) Ids
41 – 59	Reserved
60	Reserved
61	Reserved
62	Reserved
63	Reserved
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 5.10 - 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 contents
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: 0xACEF 0001
Source: source ID
Destination: destination ID
Size: 9
Options: see Command/Response Packet Format paragraph 5.3

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	Opcode 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: 0xACEF 0007
Source: source ID
Destination: destination ID
Size: 8
Options: see Command/Response Packet Format paragraph 5.3

Parameters:

Parm1: Interrupt type
- cPCI 1
- Mailbox: 2
- reserved: 3

Parm2: Info1
- When generating cPCI interrupts:
  - Interrupt level
- When generating Mailbox interrupts:
  - Processor ID

Parm3: Info2
- Reserved (must be 0)

Description:

This command generates interrupts on the cPCI bus. It also will generate mailbox 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_LOOPBACK

Header:

Opcode:	0xACEF 0005
Source:	source ID
Destination:	destination ID
Size:	5 + size of data
Options:	see Command/Response Packet Format paragraph 5.3

Parameters:

Data to be looped back

Description:

This command is used for debugging a communications path between processors. When the command is received, the IOPlus will echo the command along with any attached data fields, back to the sender.

Response:

CMD_LOOPBACK is sent to the processor initiating the command.

CMD_MOVE_DATA

Header:

Opcode: 0xACEF 0006
Source: source ID
Destination: destination ID
Size: 17
Options: see Command/Response Packet Format paragraph 5.3

Parameters:

Parm1: Source address
Parm2: Offset in bytes
Parm3: Stride in words
Parm4: Option flag

Bit Fields									
31	----------	10	9	-	6	5	----------	0	
Reserved (must be 0)			Bus Type			Memory Address Type			

Memory address type:

0x00000000: PCI address
0x00000001: cPCI address
0x00000002 IOP 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
0x0000000B -
0x0000003F: reserved

Bus type:

0x00000000: cPCI
0x00000040: PCI
0x00000080: reserved
0x000000C0: FLASH

Parm5: Destination address
Parm6: Offset in bytes
Parm7: Stride in words
Parm8: Option flag (see definition of option flag above)

Parm9: Number of words
Parm10: Operation code

Operation Code	Operation performed	Info 1	Info 2
0	Copy (no operation)	0	0
1	Byte swap	0	0
2	Word swap	0	0
3	Fixed to float	0	0
4	Float to fixed	0	0
5	Increment	Increment amount	0
6	Mask & shift	Mask	Shift amount minus is left positive is right
7	Mask, shift and convert to float		

Parm11: Info1
Parm12: Info2

Description:

This command moves data between memory located on the cPCI 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, cPCI or local bus. 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.

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: 0xACEF 000C
Source: source ID
Destination: destination ID
Size: 9
Options: see Command/Response Packet Format paragraph 5.3

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: 0xACEF 0003
Source: source ID
Destination: destination ID
Size: 7
Options: see Command/Response Packet Format paragraph 5.3

Parameters:

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

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

Parm2: Processor ID or Cluster ID

0x00000001	SPE A
0x00000002	SPE B
0x00000003	SPE C
0x00000004	SPE D
0x00000010	A/B cluster
0x00000020	C/D cluster

Description:

This command is used to reset/release SPE processors or reset the board. Validity checks for Reset type and Processor ID.

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:	0xACEF 0002
Source:	source ID
Destination:	destination ID
Size:	7
Options:	see Command/Response Packet Format paragraph 5.3

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 a 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:	0xACEF 0006
Source:	source ID
Destination:	destination ID
Size:	8
Options:	see Command/Response Packet Format paragraph 5.3

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 necessary.

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:	0xACEF 000F
Source:	source ID
Destination:	destination ID
Size:	4
Next:	0 or pointer
Options:	see Command/Response Packet Format paragraph 5.3

Parameters:

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

Description:

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

Response:

The response to this command is user-defined.

CMD_WAIT_INT

Header:

Opcode: 0xACEF 000E
Source: source ID
Destination: destination ID
Size: 8
Options: see Command/Response Packet Format paragraph 5.3

Parameters:

Parm1: Interrupt type

- Wait for cPCI0: 0x000000001
- Wait for Mailbox: 0x000000003
- Trap Mailbox: 0xFFFFFFFD
- Trap VME: 0xFFFFFFFF

Parm2: Info1

When waiting for cPCI 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 cPCI 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 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 list (note that command lists 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: 0xACEF 000B
Source: source ID
Destination: destination ID
Size: 9 + size of data
Options: see Command/Response Packet Format paragraph 5.3

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, This 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 use 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 RP 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 IXA4'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.

Table 6.1 - IOPlus VxWorks BSP Features

Feature	Description
Boot ROM Images	bootrom.hex
Boot Devices	Ethernet:fei
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 0 of EPIC
Serial	using 16C2550 dual UART
Timestamp	using timer 1 of EPIC
Ethernet	END driver INTEL 82559

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 interrupt
sysBusTas()	test and set a location across the cPCI bus
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
sysCpuCheck()	check CPU type
sysLocalToBusAdrs()	convert local address to bus address
sysIntDisable()	disable interrupts
sysIntEnable()	enable interrupts
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
sys557Init	initialize Intel 82559 END driver
sys557IntDisable	disable PMC interrupt for RP Ethernet
sys557IntEnable	enable PMC interrupt for RP Ethernet

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
0x00037000	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	PMC 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 entry allows the processors to access control registers within the Ethernet controller, and must not be modified.

Table 6.5 - Default PTE Configuration

Address Range	Cache Attributes
LOCAL_MEM_LOCAL_ADRS .. (RAM_LOW_ADRS-1)	Write-able, cache inhibited
RAM_LOW_ADDRS .. (LOCAL_MEM_SIZE–USER_RESERVED_MEM-1)	Write-able, cacheable
(LOCAL_MEM_SIZE-USER_RESERVED_MEM)..(LOCAL_MEM_SIZE-1)	Write-able, cache inhibited
ENET_IXA4_DEVICE_ADDR .. (ENET_DEVICE_ADDR+4k)	Write-able, 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
fei82557End.o	Ethernet driver for the INTEL 82559
xr2550.c	serial driver for the dual UART

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 IXA4 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.

Table 6.7 - Interrupt Vectors and Priorities

Name	Vector	Priority	Description
INT_VECTOR_TIMER0	00	08	Epic timer 0 (timestamp timer)
INT_VECTOR_TIMER1	01	08	Epic timer 1 (auxiliary timer)
INT_VECTOR_TIMER2	02	08	Epic timer 2 (unused)
INT_VECTOR_TIMER3	03	08	Epic timer 3 (unused)
INT_VECTOR_EXT0	04	06	interrupt (SPE A)
INT_VECTOR_EXT1	05	06	interrupt (SPE B)
INT_VECTOR_EXT2	06	06	interrupt (SPE C)
INT_VECTOR_EXT3	07	06	interrupt (SPE D)
INT_VECTOR_EXT4	08	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 (unused on IXA4)
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 an Ethernet interface for the INTEL 82559. The driver interface is based on the 4.4 BSD network stack (END). 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 IXA4 is programmed in FLASH memory at Dy 4 Systems and is not user alterable.

Serial Interface

The serial interface is provided by a 16C2550 dual channel UART.

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

The IOPlus initialization software, which runs before VxWorks, performs all PCI 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 Interface, 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_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 de-allocate 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	cPCI
	2	Mailbox
	3	reserved
Interrupt ID	When generating cPCI interrupts, specifies interrupt level, otherwise, specifies processor ID.	
Interrupt Vector	When generating cPCI 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_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 cPCI bus memory (i.e. other boards on the cPCI bus). The routine determines the correct bus to use to perform the transfer (including PCI, and cPCI bus). 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 cPCI bus memory.

Address Type is:

0x00000000: PCI address
0x00000001: cPCI 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	----	10	9	-	6	5	----------	0
Reserved (must be 0)			Bus Type			Set to address type (see above)		

Bus type:

0x00000000: cPCI
0x00000040: PCI
0x000000C0: FLASH

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 cPCI address.

When the return indicates that a DMA is in progress, use the routine *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 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_write_data( unsigned long section_id,
                                      unsigned long offset,
                                      unsigned long stride,
                                      unsigned long option,
                                      unsigned long num_words,
                                      unsigned long *ptr,
                                      unsigned long *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.

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.

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	Reset SPE
	0x00000002	Reset Cluster
	0x00000004	Reset board
	0x80000001	Release SPE from reset
	0x80000002	Release cluster from reset
Reset Items are:	0x00000001	SPE A
	0x00000002	SPE B
	0x00000003	SPE C
	0x00000004	SPE D
	0x00000010	A/B cluster
	0x00000020	C/D cluster

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_write_data

CALLING SEQUENCE:

```
#include <ioplus_api.h>

IOPLUS_API_STATUS  ioplus_write_data( unsigned long section_id,
                                      unsigned long offset,
                                      unsigned long stride,
                                      unsigned long option,
                                      unsigned long num_words,
                                      unsigned long *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.

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 IXA4 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 IXA4.

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 cPCI bus 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, an Ethernet boot loader capability for the IOPlus, a stdio library, and a common boot code component. The common boot code is a key element in running applications on an SPE pair which 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 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 defines an exception vector table (EVT) at physical locations zero through 0x2fff. The EVT is organized as 48 sections of 0x100 bytes each. Each section is associated with an event, such as an external interrupt, an instruction exception, or a floating-point exception.

Because the EVT lives in physical memory starting at location zero, and 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. Each processor has its own interrupt input line and interrupt sources. Therefore, each processor will need to take different actions to service the external interrupt.

The common boot code was designed to establish separate environments for each processor before the processors boot and run the 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 or 64 megabytes in length. The only reserved area is the EVT, which occurs 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 as shown in Table 7.1.

Table 7.1 - SPE Local Map with no CBC

Begin	End	Processor 1	Processor 2
0x00000000	0x00002FFF	physical EVT	physical EVT
0x00003000	0x01FFFFFF	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 at some other physical address. In order to do this, the CBC must establish a page translation table, and enable the memory management unit within the PowerPC. Physical memory then looks like that shown in Table 7.2.

Table 7.2 - SPE Local memory with CBC

Begin	End	Processor 1	Processor 2
0x00000000	0x00002FFF	physical EVT	physical EVT
0x00008000	0x0000FFFF	common boot code	common boot code
0x00040000	0x00FFFFFF	processor 1 memory	
0x01000000	0x01FABFFF		processor 2 memory
0x01FAC000	0x01FB1FFF	proc 1 initial stack	
0x01FB2000	0x01FB5FFF		proc 2 initial stack
0x01FB6000	0x01FBAFFF	proc 1 EVT	
0x01FBB000	0x01FBFFFF		proc 2 EVT
0x01FC0000	0x01FDFFFF	proc 1 page table	
0x01FE0000	0x01FFFFFF		proc 2 page table

Notice that 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 starting at location 0. This EVT is not mapped into either processor's logical address space, so it is protected from unauthorized alteration. However, each processor includes a logical EVT, which starts at logical address zero.

When an exception occurs, the PPC automatically disables the memory management unit, 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 completely transparent to the application. In fact, this method has been successfully tested with the VxWorks operating system.

Each processor uses a page table to contain MMU data. 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 128k page table 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 physical memory. The page tables are not mapped into either processor's address space: they are therefore protected from unauthorized alteration. The contents of the page tables may be interrogated and altered through IXA board support library function calls.

The configuration shown in Table 7.2 is an example. All sizes and boundaries (except for the start address and size of the physical EVT, and the start and size of the common boot code) are user-configurable.

> The logical EVT is actually 0x5000 bytes in length. Using this length remaps the EVT, and some 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.

Version Identification.

By displaying the contents of the common boot code in memory, 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 using FLASH Memory

After loading and starting the CBC, the CBC initializes memory data structures, and then waits for a download.

The IOPlus will examine its FLASH memory to determine if an application is to be loaded into the processor. If so, the IOPlus waits until the CBC has completed its initialization. The IOPlus then downloads the application, and informs the CBC to begin processing.

Loading Applications using an Emulator

If no application exists in FLASH memory, the CBC begins a polling operation, waiting to receive the starting address of the application. The CBC polls physical location 0x2000 (for processor 1, or 0x2008 for processor 2), for a nonzero value. This polling is accomplished once every several milliseconds, with cache enabled, so that a polling processor is not consuming bus bandwidth. When the processor detects a nonzero address at the poll location, it assumes the address 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 jumps to the application.

Executable images can be loaded using an emulator. The download should be performed only after the CBC has completed its initialization, and the download must not disturb the physical EVT, page tables, or common boot code. A good practice is to begin execution at location 0x8000 (the start of the common boot code), wait for the LED flash pattern (indicating initialization is complete), and then download the application. After completing the download, set the 32-bit word at location 0x2000 (or 0x2008 for processor 2) to the entry point of the application, and then resume execution. The CBC will detect the nonzero address, and call the entry point.

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 protections afforded 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: The VxWorks BSP 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 most likely destroy the processor 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.

Circumventing the CBC

An application need not use the CBC. However, the applications programmers must understand that the EVT will be shared by both processors, if the CBC is circumvented. The CBC can be disabled by an application simply 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. Edit cbconfig.c to reflect the new configuration.
2. Recompile cbconfig.c
3. Relink this file with cbcmain.o and libixa.a.
4. Save the old cbc.hex S-record file for recovery. Burn the new cbc.hex file into FLASH memory.

Adding Regions

The file cbconfig.c contains data structures that describe regions of memory for each processor. During initialization, the CBC reads these data structures and populates the page table. A region data structure entry contains the physical and logical start addresses, the size of the region, and the region attributes.

The addresses and lengths must be a multiple of the page size (4096). As the CBC reads the data structure and establishes map entries, it checks for conflicts. A conflict will occur if a region includes one or more logical page addresses that have already been mapped. Logical addresses must be unique within a processor's address space. Physical addresses can be repeated: a processor may view the same physical address through more than one logical address.

For a region of length N, the CBC adds N/4096 page table entries to the page table during initialization. It is possible for a page table overflow to occur. In this case, the size of the page table must be increased. Page table resizing guidelines are provided latter in this chapter.

Attributes govern how the processor accesses the region. 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.

Additional attributes can be specified by OR'ing or adding one or more of the following attribute 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 addresses and large blocks of shared memory. These attributes treat the access as an I/O space. The guarded attribute prevents the processor from re-ordering accesses to the memory area.

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

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 interprocessor communication should be set to this mode.

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.

Adding Blocks

The PowerPC includes two separate MMU mapping facilities. The page table maps memory in 4k pages. This requires a large data structure (a page table) that describes the mapping and attributes for each page. The CBC builds a page table from the region data structures as describe above.

It is often desirable to map a large contiguous block of memory. A large page table is not recommended. Performance would be somewhat degraded, because the probability of a memory access requiring a page table access is increased. As an alternative to using the page table, the PowerPC provides a block address translation mechanism.

Up to four blocks can be defined for mapping instruction accesses, and four blocks can be defined for mapping data accesses. A block is a contiguous region of logical and physical memory, having a length that is a power of two between 128k bytes to 256 megabytes (inclusive). A block must start at an address that is a multiple of its size, and be mapped to an address that is a multiple of its size. Regions of memory that do not meet this criteria must be mapped using the page table.

cbconfig.c includes data structures for each processor which load the Block Address Translation (BAT) registers. These registers can be altered by the user. Macros translate the logical address, physical address, length, and attribute flags into BAT register images.

The CBC loads these images into the BAT registers during setup. The flag values used are the same as those used to describe region attributes.

BAT registers must not describe overlapping logical address spaces. Conflicts among BAT registers cause a machine fault. BAT registers may conflict with page table entries. In this case, the processor uses the BAT register mapping, not the page table mapping.

Altering the Page Table Size

The page table size is described in cbconfig.c. The page table size must be a power of two, between 64k and 32 megabytes in length. Furthermore, the page table must start at a memory address that is a multiple of the page table size. That is, a page table of length 256k bytes must start at a 256k byte boundary.

When a page table overflows (because more or larger regions have been added), the page table size must be increased. Because the page table size must be a power of two, the next larger page table size is twice the last page table size.

The page table is organized as a hash table, with each entry requiring 8 bytes. In order to leave adequate room for hashing collisions, Motorola recommends the page table be sized using the following formula:

Page Table Size = [(total # pages) * 16] rounded to the next highest power of two.

This is the minimum recommended size. Certain combinations of logical addresses may generate more collisions, and require a larger page table.

The CBC meets the page table alignment requirements by placing the page tables at the end of physical memory. Since the size of memory is a power of two (32 megabytes or 64 megabytes), the memory size minus the page table size is always aligned to a multiple of the page table size. Two page tables must live within each physical memory (one page table for each processor). The source file cbconfig.c places the largest page table at the end of physical memory, followed by the smaller page table. This insures alignment of both page tables.

Note that page tables are not mapped into either processor's logical address space. Page tables are accessed by physical address. The processor has no need to map the tables into its logical space. The processor updates and accesses the page tables using physical accesses. Excluding the page tables from the logical mapping prevents the application from inadvertently altering the page table contents.

Altering Memory Allocation

The file cbconfig.c allocates approximately half of the physical memory to each processor within the pair. This allocation can easily be changed by altering cbconfig.c.

Creating Shared Regions

Some regions are mapped into the logical address space of more than one processor. Such regions are said to be shared regions. Both processors can access such a region.

One example of a shared region is the common boot code, starting at physical (and logical) address 0x8000. Both processors see (and execute) the exact same code from this region. The attributes for this region are set to ATTR_RO. If either processor tries to write to the common boot code region, an exception will result.

Data regions can be shared as well. Such a region will have attributes of ATTR_RWX. Note that creating a shared region is a necessary, but not sufficient, step in sharing data in a way that provides coherency and integrity. Further steps must be taken to insure data integrity. These steps depend on the level of visibility that other processors (or I/O devices) require.

If a region includes data to be shared only between the two processors sharing a common bridge, then the data pages can be marked with the attribute ATTR_M. This causes the caching mechanism to enforce memory coherency. This coherency requires additional bus bandwidth and coordination, so it must be used judiciously.

Regions shared between processors and I/O devices (or between processors serviced by different bridge chips) require a different approach. This is because the signals used to provide memory coherency are not extended across the PCI bus. One approach is to mark such regions as cache inhibited and guarded. This causes the processors to bypass cache and to perform reads and writes in the order generated by the software.

If a region of memory is used for output data from the processor to an I/O device (or to other processors), then the region can be marked as cache enabled and write-through. This forces the processor to update memory and cache as writes to the output area are performed.

Finally, a shared region can be fully cached if the application software assumes responsibility for flushing and invalidating the cache. The processor should flush cache immediately after writing, and invalidating cache before reading. Functions within the board support library can accomplish this.

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 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 (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.

Memory Configurations

The memory configuration for each processor is described in cbconfig.c. The CBC establishes memory management data structures at power-up time from the data structures in this source file. This file may be edited, recompiled, and re-burned into FLASH to customize the memory configuration.

A processor's memory configuration is described using the following primary data structure:

```
struct CBC_CONFIG
{
      unsigned long stack_ptr;      /* initial stack ptr       */
      unsigned long *bats;          /* -> bat register values  */
      struct REGION *r;             /* -> region information   */
      struct REGION *shared;        /* -> shared regions       */
};
```

Because each processor must have a separate stack, the initial stack pointer value must be specified here. The remaining fields point to other data structures used during initialization. "bats" contains block address translation (BAT) register images. The "r" region information points to a table of private regions, while the "shared" region pointer points to a list of regions which will be mapped in for all processors. "shared" is intended for commonly accessible board resources, such as PCI devices. An example instance of this data structure for processor A follows:

```
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      */
};
```

Region Data Structure

A region describes an area of contiguous memory having a length that is a multiple of the page size. The exception vector table is mapped to a region. The initial stack can occupy

its own region. During initialization, the CBC adds a page table entry for each page represented by a region. A region has the following format:

```
struct REGION
{
        unsigned long logical;      /* logical start addr   */
        unsigned long physical;     /* mapped to this addr  */
        unsigned long size;         /* region size (bytes)  */
        unsigned long attr;         /* cache attributes     */
};
```

The address and size fields must be multiples of the page size. Attributes are the sum or logical OR of one or more of the following values:

```
#define ATTR_CLEAR      0x80        /* zero out during init */

#define ATTR_W          0x40        /* write through mode   */
#define ATTR_I          0x20        /* inhibit cache mode   */
#define ATTR_M          0x10        /* memory coherency reqd*/
#define ATTR_G          0x08        /* guarded              */

#define ATTR_RWX        0x02        /* rd, wr, and execute  */
#define ATTR_RO         0x01        /* rd, execute only     */
```

A region attribute consists of exactly one protection attribute (ATTR_RWX or ATTR_RO), optionally OR'ed with one or more caching attributes (ATTR_W, ATTR_I, ATTR_M, ATTR_G), optionally OR'ed with an initialization attribute (ATTR_CLEAR). The protection and cache attributes correspond to the PPC's memory access control bits. The ATTR_CLEAR bit is an artificial extension used by the common boot code during initialization. Example:

ATTR_RWX | ATTR_I | ATTR_G | ATTR_CLEAR

> ATTR_CLEAR must not be specified for a stack segment. This is because the stack is in use at the time the common boot code initializes the segments. Clearing out the stack segment results in an unrecoverable exception during initialization.

The values PAGE_TABLE and REGION_END have special meaning within the logical address field. PAGE_TABLE indicates an area of memory with a valid length and physical address, but the logical addresses will not be mapped. As the name implies, this is used to specify a region used for a page table. Within an array of multiple regions, REGION_END follows the last segment. Example:

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

External Memory Configuration

External memory is on-board memory that is not attached to the local processor's bridge device. The IOPlus' memory (the on-board global memory), memory belonging to the "other" processor pair, and memory available on PMC cards are considered to be external memory from a single processor's point of view.

The global memory and the other processor pair's memory may be mapped in using BAT registers. Because these memories are large and meet alignment requirements for block translation, BAT registers are ideal for this purpose. BAT registers for each processor are described in a data structure having eight entries corresponding to the eight PowerPC BAT registers. The first four map instruction accesses, the last four map data accesses:

```
static unsigned long proca_batregs[] =
{
    BATREG_NULL,    /* instruction BATS */
    BATREG_NULL,
    BATREG_NULL,
    BATREG_NULL,
    BATREG_GLOBAL, /* data BATS */
    BATREG_NULL,
    BATREG_NULL,
    BATREG_NULL
};
```

BAT register contents consist of two 32-bit words. These words are constructed from the following macros:

```
BATREG_S(log,phys,len,attr) generates BAT register values for a block
which is accessible in supervisor mode only.

BATREG_P(log,phys,len,attr) generates BAT register values for a block
which is accessible in problem (user) state only.

BATREG_SP(log,phys,len,attr) generates BAT register values for a block
accessible in both problem and user states.

BATREG_NULL must be specified for unused BAT registers.
```

The len parameter specifies the block length. The length must be a power of two, between 128k and 256M, inclusive. Attributes can be one or more of the following values, logically Ored together:

```
#define BATREG_G  0x08        /* guarded                    */
#define BATREG_M  0x10        /* memory coherence required  */
#define BATREG_I  0x20        /* cache inhibited            */
#define BATREG_W  0x40        /* write through              */
#define BATREG_RW 0x02        /* read/write protection      */
```

Example: The following BAT register specification maps the global memory on the board into the address space of one of the PPCs. The global memory resides at a PCI starting address of 0xb8000000, and has a length of 32 megabytes. This region of memory will be readable and writable. We specify cache inhibited and guarded attributes, since this memory space is used for communication between processors across PCI bus:

```
/*
 *  BATREG value for global memory.  Global (CPE) memory starts
 *  at address 0xb800_0000 on the PCI bus, and is 32 meg long.
 *  It is accessible from supervisor state and problem state.
 */

#define BATREG_GLOBAL \
      BATREG_SP(0xb8000000, 0xb8000000, 0x02000000, \
      BATREG_G+BATREG_I+BATREG_RW)
```

PMC cards can be mapped-in using BAT registers or pages. Page table entries are best suited for PMC cards requiring a small address space (e.g., for control registers). BAT registers may be used for PMC cards requiring a large address space, provided the address space meets block alignment criteria. Recall that there are only four BAT registers available for data mapping. Using BAT registers for PMC access may necessitate trading a BAT register entry configured for some other purpose.

Local Memory Configuration Strategies

Separate Logical and Physical Address Spaces

In this model, each processor within a pair may access to up to approximately half of the available memory. Logical EVTs, stacks, and page tables are located at the end of physical memory. Each processor maps in the common boot code and data areas (addresses 0x8000 through 0xffff). Each processor then maps in a “main” region of memory as follows:

Processor A (C):
    Logical 0x0001_0000 to (memory size)/2
    mapped to physical 0x0001_0000 to (memory size)/2
Processor B (D):
    Logical (memory size)/2 + 0x0001_0000 to END_MEMORY
    mapped to physical (memory size)/2 + 0x0001_0000 to END_MEMORY

where
    END_MEMORY = size of memory - sizeof(page tables) - sizeof (stacks)
                 - sizeof (EVTs)

Note that applications built to run on processor A and B must be built to run in these separate address spaces. Note also that logical addresses within the main memory region correspond to physical addresses.

Region tables for processors A and B having this memory configuration might look like this. Note that MAIN_BEGIN_A and MAIN_BEGIN_B are mapped so that the logical addresses within these regions equal the physical addresses. This example presumes that the MAIN regions do not overlap.

```
static struct REGION proca_regions[] =
{
    {0x00000000,   EVT_A,       EVT_SIZE,   ATTR_CLEAR+ATTR_RWX},
    {MAIN_BEGIN_A, MAIN_BEGIN_A,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}
};
static struct REGION procb_regions[] =
{
    {0x00000000,   EVT_B,       EVT_SIZE,   ATTR_CLEAR+ATTR_RWX},
    {MAIN_BEGIN_B, MAIN_BEGIN_B,MAIN_SIZE_B,ATTR_CLEAR+ATTR_RWX},
    {STACK_B,      STACK_B,     STACK_SIZE_B, ATTR_RWX},
    {PAGE_TABLE,   PGTBL_B,     PGTBL_SIZE_B, 0},
    {REGION_END,   REGION_END,  0,            0}
};
```

Congruent Logical/Separate Physical

As in the previous model, each processor within a pair may access up to approximately half of the available memory. Logical EVTs, stacks, and page tables are located at the end of physical memory. Each processor maps in the common boot code and data areas (addresses 0x8000 through 0xffff). Each processor then maps in a “main” region of memory as follows:

Processor A (C):

Logical 0x0001_0000 to (memory size)/2
mapped to physical 0x0001_0000 to (memory size)/2

Processor B (D):

Logical 0x0001_0000 to (END_MEMORY - (memory size)/2)
mapped to physical (memory size)/2 + 0x0001_0000 to END_MEMORY

where

END_MEMORY = size of memory - sizeof(page tables) - sizeof (stacks)
- sizeof (EVTs)

In this case, both processors see a region of memory beginning at logical address 0x0001_0000 covering approximately half of memory. This means that any application built to run in the logical space starting at 0x0001_0000 will run in processors A or B. Each processor will have a separate copy of the application in its physical memory, but they can run the same application.

```
static struct REGION proca_regions[] =
{
    {0x00000000,   EVT_A,       EVT_SIZE,   ATTR_CLEAR+ATTR_RWX},
    {MAIN_BEGIN_A, MAIN_BEGIN_A,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}
};
static struct REGION procb_regions[] =
{
    {0x00000000,   EVT_B,       EVT_SIZE,   ATTR_CLEAR+ATTR_RWX},
    {MAIN_BEGIN_A, MAIN_BEGIN_B,MAIN_SIZE_B,ATTR_CLEAR+ATTR_RWX},
    {STACK_B,      STACK_B,     STACK_SIZE_B, ATTR_RWX},
    {PAGE_TABLE,   PGTBL_B,     PGTBL_SIZE_B, 0},
    {REGION_END,   REGION_END,  0,            0}
};
```

Notice that processor B's MAIN logical address space begins at the same location as processor A's address space. Both processors have a similar logical address space, but the MAIN portion of each processor's memory occupies a different physical region of memory.

Combined

This is the factory configuration of cbconfig.c. This strategy combines the above two strategies. A single physical page can be mapped to multiple logical addresses. In this case, the B (D) processor maps its main memory region into the logical space starting at 0x0001_0000, and the logical address space starting at (memory size)/2 + 0x0001_0000.

An advantage of this method is that the B (D) processor can accept and execute software built to run in processor A, or it can accept and execute software built to run in the high spaces available to processor B (D) only.

A disadvantage of this approach is that a larger page table is required.

In the example below, note that the B processor maps its MAIN memory space into two logical address spaces: the logical address space beginning at MAIN_BEGIN_A, and also the space beginning at MAIN_BEGIN_B.

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

```
static struct REGION procb_regions[] =
{
    {0x00000000,   EVT_B,       EVT_SIZE,   ATTR_CLEAR+ATTR_RWX},
    {MAIN_BEGIN_A, MAIN_BEGIN_B,MAIN_SIZE_B,ATTR_CLEAR+ATTR_RWX},
    {MAIN_BEGIN_B, MAIN_BEGIN_B,MAIN_SIZE_B,ATTR_CLEAR+ATTR_RWX},
    {STACK_B,      STACK_B,     STACK_SIZE_B, ATTR_RWX},
    {PAGE_TABLE,   PGTBL_B,     PGTBL_SIZE_B, 0},
    {REGION_END,   REGION_END,  0,            0}
};
```

Leaving Space for Global Buffers

It is sometimes desirable to leave a portion of the memory available for buffer space. External devices (PMC devices or other processors on the board) can use this space for data movement.

Because the space will be available for access by other devices, caching attributes must be carefully considered. Enabling cache for such a region is not recommended. While the bridge device is capable of "snooping" PCI accesses to local memory, the snooping activity will compete for the local processor bus, compromising the bus bandwidth of both processors, whether they are interested in the data buffer or not.

Inhibiting cache is the simplest option, but this may not provide the best performance. It may be more effective to leave cache enabled, and then to use the cache flush (or cache invalidate) functions after writing output data (or before reading input data) to externally visible memory.

In order to carve out a space for data buffers, reduce the size of the main memory region for one or both processors. If the size of the A (C) processor main memory region was reduced, then adjust the starting address of the B (D) memory region as well. Adjusting the sizes will open up a "gap" in the physical address space between the end of the B (D) processor main memory region, and the start of the stack/EVT/page table area. Recall that these areas are always allocated at the end of physical memory.

This new region can be mapped into one, or both, processors by adding an appropriate region to the processor's region table. The shared region table is for regions common to all four processors on a board. If each pair of processors has a buffer space with the same size and starting address, a region entry may be added to the shared region table. Otherwise, the new region should be added to the private region table for all processors requiring access to the region.

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 to access to the internal 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.

Table 7.3 - SPE VxWorks BSP Features

Feature	Description
Boot ROM Images	Bootrom.hex
Boot Devices	Ethernet:sm
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 & L2 cache in copyback cache mode
Sysclock	using PPC Decrementer
AuxClock	using timer 0 of EPIC
Serial	using shared memory protocol
Timestamp	using timer 1 of EPIC
Ethernet	shared memory network using PCI bus

7.5 Commanding the IOPlus from a SPE

The IXAbsp library includes functions to create and manage linked lists of command packets to be processed by the IOPlus. From the IXAbsp perspective, the mechanism for controlling IOPlus command processing is called a command channel. The IOPlus defines a fixed number of command channels that can be allocated by the SPEs or by host processes. The IXAbsp library provides functions to open and close command channels, to create command lists associated with a command channel, to start and stop command list processing for a specific command channel, and to check the status of command channels.

Associated with a command channel is a command list where command packets for IOPlus processing are located. A command list consists of one or more command packets and can be located either in SPE memory space or in PCI memory space. Also associated with a command channel is a response list where the IOPlus places response packets created during the processing of the commands in the command list. Like the command list, the response list consists of one or more response packets and can be located either in SPE memory space or in PCI memory space. The locations of the command and response lists are specified by the IXAbsp function call that opens a command channel. The command and response lists are located independently of each other, i.e. both can reside in PCI memory space, both in SPE memory space, or one in PCI memory space and the other in SPE memory space.

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 decrease in command processing performance as the IOPlus must access the channel control structure through the PCI bridges.

Table 7.4 - SPE 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 interrupt
sysBusTas()	test and set a location across the cPCI bus
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
sysCpuCheck()	check CPU type
sysLocalToBusAdrs()	convert local address to bus address
sysIntDisable()	disable interrupts
sysIntEnable()	enable interrupts
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

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.

Using Command Channels

A 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 lists for the channel are located, whether they exist in SPE or PCI memory space, and if the command channel is to be relocated to SPE memory space.

For example, the following code segment opens a command channel, placing the command list in a global memory block allocated using the *ixa_malloc* function, the response list local memory allocated using *malloc*, and relocating the command channel to local memory:

```
CMD_CHANNEL   cmd_chan;
unsigned int *cmd;
unsigned int *response;
unsigned int  src_opt;
unsigned int  dest_opt;
IXABSP_STATUS stat;
unsigned int  cmd_chan_option;
CMD_ERROR     cmd_err;

cmd = (unsigned int *) ixa_malloc((CMD_MOVE_DATA_SZ +
                                   CMD_GENERERATE_INT_SZ) *
                                   sizeof(unsigned int));

response = (unsigned int *) malloc(2 * CMD_ACK_SZ
                                     * sizeof(unsigned int));

cmd_chan_option =  CMD_CHANNEL_OPTION_RESP_LOCAL |
                   CMD_CHANNEL_OPTION_RELOCATE;

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

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

Once a command channel is opened, the command list 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 list, care must be taken to create the list in the proper order, starting with the first command packet in the list and ending with the last. The next pointer field in a command packet is used to determine where the subsequent command in the list 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 list and could result in corrupted memory. The next field of the last command packet in the list should be set to zero. 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 a command list consisting of a move data command packet which transfers data from the cPCI bus to global memory followed by a generate interrupt command packet which will trigger a mailbox interrupt:

```
src_opt = CMD_MOVE_DATA_PCI_ADDR;

dest_opt = CMD_MOVE_DATA_PCI_ADDR;

ixa_cmd_set_opcode(&cmd_chan, 0, CMD_MOVE_DATA);
ixa_cmd_set_option(&cmd_chan, 0, CMD_OPTION_ACK);
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
*/
ixa_cmd_set_next(  &cmd_chan, 0, cmd+CMD_MOVE_DATA_SZ);

ixa_cmd_set_opcode(&cmd_chan, 1, CMD_GENERATE_INT);
ixa_cmd_set_option(&cmd_chan, 1, CMD_OPTION_ACK);
ixa_cmd_set_param(&cmd_chan, 1, 0, CMD_GENERATE_INT_MBOX);/* int type
*/
ixa_cmd_set_param(&cmd_chan, 1, 1, 4); /* mailbox 4 */
ixa_cmd_set_param(&cmd_chan, 1, 2, 0); /* not used */
ixa_cmd_set_next( &cmd_chan, 1, 0);
```

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 list has been created, the processing of the commands by the IOPlus 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. When called in asynchronous mode, the function returns as soon as the command processing has been started. Upon completion of the command list 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 list 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 list 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);
```

7.6 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_PCI_**** pulls all the PCI bus 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:

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:

ixa_cache_disable flushes, 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:

ixa_cache_flush will flush *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:

ixa_cache_invalidate will invalidate *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:

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:

ixa_cache_sync will synchronize 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_temp_read.

ixa_cache_throttle_write

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_cache_throttle_write ( int delay );
```

DESCRIPTION:

The function, *ixa_cache_throttle_write* 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 a range 0 to 255. If the new delay value is negative 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.

ixa_cmd_close

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_close (CMD_CHANNEL *cmd_chan);
```

DESCRIPTION:

ixa_cmd_close frees an IOPlus 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 list */
  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* structure element indicates how many error responses exist in the command channel response list. The *cmd_op* parameter identifies the command operation that generated the first error in the response list. The *err_xxxx* values are the parameters of the first error CMD_ACK response packet in the response list. 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 section 5.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_CMD_CHANNEL_BUSY: command channel is 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 command channel to the IOPlus. 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 list. *response_addr* points to the response list. 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 command channel 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 – relocate the command channel to local memory

The IOPlus has a fixed number of available command channels. Once allocated by calling *ixa_cmd_open*, a command channel can be unallocated by calling *ixa_cmd_close*, thereby freeing the command channel for use by another processor.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_RESOURCE_UNAVAILABLE: no command channels are available.

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 creating a command list for a command channel. The *cmd_chan* parameter specifies the command channel for the command list, and must be a valid CMD_CHANNEL pointer returned by *ixa_cmd_open*. *cmd_idx* specifies which command in the list is being modified, with the first command in the list 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 packet.

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 command list

ixa_cmd_set_next – sets the next field of the command. The last command in a command list 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 list.

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 be a valid CMD_CHANNEL pointer returned 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 list associated with the command channel should have been created 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 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_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 be a valid CMD_CHANNEL pointer returned by the *ixa_cmd_open* function.

RETURN STATUS:

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

ixa_cmd_VME_error

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_VME_error (CMD_ERROR   *cmd_err);
```

DESCRIPTION:

The *ixa_cmd_VME_error* function 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.

NOTES:

ixa_cmd_VME_error is meaningful only on IXA VME boards.

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.

NOTES:

ixa_cmd_VME_read and *ixa_cmd_VME_write* are meaningful only on IXA VME boards.

ixa_cmd_VME _status

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_cmd_VME_status (unsigned int *status);
```

DESCRIPTION:

The *ixa_cmd_VME_status* function 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.

NOTES:

ixa_cmd_VME_status is meaningful only on IXA VME boards.

ixa_CPCI_close

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_CPCI_close (CPCI_DEVICE *dev);
```

DESCRIPTION:

ixa_CPCI_close closes a window to the compact PCI backplane. If the window was opened as a shared window, then this function takes no action and returns a status indicating success. If the window was not a shared window, the window is made available for reallocation, and a successful status is returned.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

NOTES:

ixa_CPCI_close is meaningful only on IXA Compact PCI boards.

ixa_CPCI_open

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_CPCI_open ( CPCI_DEVICE *dev,
                              void *cpci_addr,
                              unsigned char shared);
```

DESCRIPTION:

ixa_CPCI_open opens a window of memory mapped to the compact PCI backplane. The cPCI bridge is adjusted to map the desired address into the processor's local address space. The data structure at *dev is updated to reflect this mapping.

The bridge device on the IXA-4 supports two master windows. *ixa_CPCI_open* allocates an available window. If *shared* is zero, then the window is opened for exclusive use. If *shared* is nonzero, then the window is opened for shared use. Multiple processors opening the same window with a nonzero *shared* option will allocate only a single window. The *shared* mode is strongly recommended in cases where all four processors have to access the same general region of cPCI backplane space.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

NOTES:

The window size is 32 megabytes (0x02000000 bytes).

cpci_addr must be a multiple of the window size.

ixa_CPCI_open is meaningful only on IXA Compact PCI boards.

ixa_CPCI_read

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_CPCI_read ( CPCI_DEVICE *dev,
                              void *cpci_addr,
                              void *local_addr,
                              unsigned long count,
                              int swap);
```

DESCRIPTION:

ixa_CPCI_read moves data from the compact PCI backplane at address *cpci_addr* to a local buffer at *local_addr*. *Count* 32-bit words are moved. If *swap* is nonzero, then bytes are reordered during the copy. *dev* is a device structure initialized by *ixa_CPCI_open*.

ixa_CPCI_read is meaningful only on IXA Compact PCI boards.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

NOTES:

ixa_CPCI_read is meaningful only on IXA Compact PCI boards.

ixa_CPCI_to_local, ixa_local_to_CPCI

CALLING SEQUENCE:

```
#include <ixa.h>

void *ixa_CPCI_to_local (     CPCI_DEVICE *dev,
                              void *cpci_addr);

void *ixa_local_to_CPCI (     CPCI_DEVICE *dev,
                              void *local_addr);
```

DESCRIPTION:

Given a device descriptor populated by ixa_CPCI_open, these functions translate a CPCI backplane address to a local address or visa-versa. These functions return 0xffffffff if the translation cannot be performed.

RETURN STATUS:

Translated address, or 0xffffffff if error.

NOTES:

ixa_CPCI_to_local and *ixa_local_to_CPCI* are meaningful only on IXA Compact PCI boards.

ixa_CPCI_write

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_CPCI_write( CPCI_DEVICE *dev,
                              void *cpci_addr,
                              void *local_addr,
                              unsigned long count,
                              int swap);
```

DESCRIPTION:

ixa_CPCI_write moves data from the local buffer at local_addr to the compact PCI backplane at address *cpci_addr*. *count* 32-bit words are moved. If *swap* is nonzero, then bytes are reordered during the copy. *dev* is a device structure initialized by *ixa_CPCI_open*.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

NOTES:

ixa_CPCI_write is meaningful only on IXA Compact PCI boards.

ixa_delay

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_delay (unsigned long nticks);
```

DESCRIPTION:

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:

These 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:

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:

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 starting the output DMA operation.

ixa_evt_disable

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_evt_disable ( void );
```

DESCRIPTION:

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:

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:

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:

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:

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 device 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* will return 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 IXA 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* will get 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:

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:

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:

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:

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 *ixa_get_vector* function 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 *ixa_int_lock* function 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 as *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 *ixa_int_setpri* function will set 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:

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 *ixa_int_unlock* function 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_ack

CALLING SEQUENCE:

```
#include <ixa.h>

int ixa_ipi_ack ( int target_proc );
```

DESCRIPTION:

The function *ixa_ipi_ack* will acknowledge (clear) an inter-processor interrupt asserted by processor *target_proc* where *target_proc* is the board processor ID (as opposed to the system processor ID) of the interrupting processor.

RETURN STATUS:

A non-zero value will be returned if an invalid target processor ID is passed. Otherwise, the function will return a zero.

NOTES:

User applications should not call this function. The interrupt service shell within IXA tools automatically acknowledges inter-processor interrupts before calling the user-specified interrupt handler.

ixa_ipi_disable

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_ipi_disable ( void );
```

DESCRIPTION:

The function *ixa_ipi_disable* disables all inter-processor interrupts (IPI).

RETURN STATUS:

None

NOTES:

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_IPI0 + procNum`,), and then enable the interrupt (by calling *ixa_ipi_enable*).

The vector number for *ixa_int_setvect* is always INT_VECTOR_IPI0 + *procNum*, where *procNum* is the number of the processor receiving interrupts (0 for the IOP, 1 for the A processor, etc.)

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_enable

CALLING SEQUENCE:

```
#include <ixa.h>

void ixa_ipi_enable ( void );
```

DESCRIPTION:

The function *ixa_ipi_enable* enables all inter-processor interrupts (IPI).

RETURN STATUS:

None

NOTES:

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_IPI0 + procNum`,), and then enable the interrupt (by calling *ixa_ipi_enable*).

The vector number for *ixa_int_setvect* is always INT_VECTOR_IPI0 + *procNum*, where *procNum* is the number of the processor receiving interrupts (0 for the IOP, 1 for the A processor, etc.)

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* will blink 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* will blink 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* will turn 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* will turn 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* will return 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* will translate 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* will update 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* will create 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* will read a logical address, *addr*, using the MMU while function *ixa_mmu_poke_l* will write 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* will read a physical address, *addr*, bypassing the MMU while function *ixa_mmu_poke_p* will write 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* will update or replace 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* will create or replace 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* will create or replace *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* will disable 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* will invalidate the page table entry for that address. Function *ixa_mmu_unmap_pages* will invalidate *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 *local_adrs);
void *ixa_local_to_pci (void *pci_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.

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.

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 *ixa_PCI_read* transfers data from the PCI bus to local memory while the *ixa_PCI_write* function 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 an on-board shared memory semaphore residing in global memory semaphore to serialize access to the TAS location. All other processor 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 underlying semaphore is busy. 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 Thermal Assist Unit measures temperatures in 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 unknown.

RETURN STATUS:

Junction temperature, in degrees Centigrade, in range from 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:

ixa_ timer_cancel will cancel 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 *ixa_timer_create* function 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 *ixa_timer_get_ticks_per_second* will return 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 *ixa_timer_get_TBL* function 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 *ixa_timer_get_time* function will return 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 *ixa_timer_get_usec* function will return 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 *ixa_timer_get_timebase* will return 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 *ixa_timer_init* function 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:

These 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 *ixa_timer_set* function 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:

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 initialization 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:

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.

ixa_timern_get_time 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 s, int n);
unsigned long ixa_timern_sec_to_ticks (unsigned long ms, int n);
unsigned long ixa_timern_usec_to_ticks (unsigned long us, int n);
```

DESCRIPTION:

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 *msec*, *sec*, and *usec* convert milliseconds, 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:

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:

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:

Ixa_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:

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 C processors is meaningless.

ixa_VME_close, ixa_VME_open

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_VME_open (  VME_DEVICE     *vme_dev,
                              unsigned int    addr_space,
                              unsigned int    addr_mod,
                              unsigned int    data_size);


IXABSP_STATUS ixa_VME_close(  VME_DEVICE     *vme_dev);
```

DESCRIPTION:

ixa_VME_open allocates a PCI-to-VME image from the VME interface chip and initializes the VME access characteristics based on the following parameters:

vme_dev - pointer to a VME_DEVICE structure.
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.

Upon successful return from *ixa_VME_open*, the *vme_dev* parameter can then be passed as a parameter to the *ixa_VME_read* and *ixa_VME_write* functions. Eight images are available on the board. A single processor can perform multiple opens, each with different VME access characteristics. Note, however, that all the SPE processors and the IOPlus share these PCI-to-VME images. *ixa_VME_close* releases the PCI-to-VME image associated with *vme_dev* and make it available for reallocation.

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_RESOURCE_UNAVAILABLE: no PCI-to-VME images available.

NOTES:

ixa_VME_close and *ixa_VME_open* are meaningful only on IXA VME boards.

ixa_VME_dma

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_VME_dma (       VME_DEVICE     *vme_dev,
                                  void           *vme_addr,
                                  void           *local_addr,
                                  unsigned int    direction,
                                  unsigned int    count,
                                  unsigned int    flags);
```

DESCRIPTION:

The *ixa_VME_dma_init* function initializes the PCI DMA engine of the Board Resource Manager to transfer data between the VME address *vme_addr* and *local_addr*. *local_addr* must be located in SDRAM memory. Prior to calling *ixa_VME_dma_init, ixa_VME_open* must be called to setup a PCI-to-VME image within *vme_dev*. The VME access characteristics of the DMA operation are determined by the open call. The direction of the DMA is specified by *direction* (PCI_DMA_PCI_TO_LOCAL, PCI_DMA_LOCAL_TO_PCI), and the number of 4-byte values to transfer is specified by *count*. The *flag* parameter specifies the following additional DMA options which can be OR'd together:

PCI_DMA_INTR_ENABLE – a DMA done interrupt will occur at the completion of the transfer on the ISN_EXT_INT6 source. If this option is not specified, a DMA done interrupt will not occur.

Once a PCI DMA has been initialized, the application must call *ixa_VME_dma_start* for the DMA to commence.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPENED: *vme_dev* not opened.
IXABSP_RESOURCE_NOT_AVAILABLE: Board Resource Manager DMA channel is in use.
IXABSP_ADDRESS_OUT_OF_RANGE: the starting or ending *local_address* of the DMA is not a valid SDRAM address.

NOTES:

ixa_VME_dma is meaningful only on IXA VME boards.

ixa_VME_dma_start

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_VME_dma_start (void);
```

DESCRIPTION:

This function starts a Board Resource Manager PCI DMA that has been initialized previously by the *ixa_VME_dma_init* function.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

NOTES:

ixa_VME_dma_start is meaningful only on IXA VME boards.

ixa_VME_dma_status

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_VME_dma_status (*dma_status);
```

DESCRIPTION:

The *ixa_VME_dma_status* function returns in *dma_status* the status of the Board Resource Manager PCI DMA engine. Possible values are PCI_DMA_DONE, PCI_DMA_IN_PROGRESS, and PCI_DMA_ERROR.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

NOTES:

ixa_VME_dma_status is meaningful only on IXA VME boards.

ixa_VME_int_clear

CALLING SEQUENCE:

```
#include <ixa.h>

ixa_VME_int_clear ( unsigned int vme_intr);
```

DESCRIPTION:

ixa_VME_intr_clear function clears the pending VME interrupt specified by *vme_intr*. Valid values for *vme_intr* are as follows:

VERR – VME error interrupt
VIRQ1 – VME level 1 interrupt
VIRQ2 – VME level 2 interrupt
VIRQ3 – VME level 3 interrupt
VIRQ4 – VME level 4 interrupt
VIRQ5 – VME level 5 interrupt
VIRQ6 – VME level 6 interrupt
VIRQ7 – VME level 7 interrupt

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

NOTES:

ixa_VME_intr_clear is meaningful only on IXA VME boards.

ixa_VME_int_disable, ixa_VME_int_enable

CALLING SEQUENCE:

```
#include <ixa.h>

ixa_VME_int_enable (    unsigned int vme_intr,
                        unsigned int lintr_src,
                        unsigned int SPE_intr_src);

ixa_VME_int_disable (   unsigned int vme_intr,
                        unsigned int lintr_src,
                        unsigned int SPE_intr_src);
```

DESCRIPTION:

ixa_VME_intr_enable function enables the Universe VME bridge to monitor the VME interrupt specified by *vme_intr*. Valid values for *vme_intr* are as follows:

VERR – VME error interrupt
VIRQ1 – VME level 1 interrupt
VIRQ2 – VME level 2 interrupt
VIRQ3 – VME level 3 interrupt
VIRQ4 – VME level 4 interrupt
VIRQ5 – VME level 5 interrupt
VIRQ6 – VME level 6 interrupt
VIRQ7 – VME level 7 interrupt

lintr_src specifies which LINT# the VME bridge asserts when the VME interrupt occurs. Valid values for *lintr_src* are INT_L0, INT_L1, INT_L2, INT_L3, INT_L4, INT_L5, INT_L6, or INT_L7. As a convenience, this function calls *ixa_intr_map* to map the *lintr_src* to the specified *SPE_intr_src* in the interrupt mux.

The user application must call *ixa_int_setvect* to setup the interrupt service routine for the interrupt vector in initialization.

The *ixa_VME_intr_disable* function disables monitoring of the specified *vmd_intr*. As a convenience, *ixa_intr_unmap* is called to unmap *lintr_src* from the specified *SPE_intr_src* in the interrupt mux.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.

NOTES:

ixa_VME_int_disable and *ixa_VME_int_enable* are meaningful only on IXA VME boards.

ixa_VME_int_gen

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_VME_int_gen(unsigned int level,
                unsigned int vector);
```

DESCRIPTION:

ixa_VME_intr_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.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_VME_INVALID_INTR: invalid *level* parameter.

NOTES:

ixa_VME_int_generate is meaningful only on IXA VME boards.

ixa_VME_read, ixa_VME_write

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_VME_read (  VME_DEVICE      *vme_dev,
                              void            *vme_addr,
                              void            *local_addr,
                              unsigned int     count,
                              unsigned int     mode,
                              int              swap);


IXABSP_STATUS ixa_VME_write(  VME_DEVICE      *vme_dev,
                              void            *vme_addr,
                              void            *local_addr,
                              unsigned int     count,
                              unsigned int     mode,
                              int              swap);
```

DESCRIPTION:

ixa_VME_read and *ixa_VME_write* perform master VME transfers by directly accessing the board VME interface chip via the PCI bus. Prior to calling *ixa_VME_vme_read* and *ixa_VME_write*, the *ixa_VME_open* function must be called to open the VME device and set up the address modifier and data size of the transfer. The *vme_dev* parameter must be a valid VME_DEVICE pointer returned by *ixa_VME_open*. *vme_addr* is the starting VME address of the transfer and *local_addr* is an address local to the calling processor. The *count* parameter specifies the number of data elements to be transferred. The size of the data elements is determined by the *data_size* parameter used in the *ixa_VME_open* call to open the *vme_dev*. When *swap* is set to 1, data is byte-swapped before being written (or after being read). If *swap* is set to 0, data is not byte-swapped.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPENED: *vme_dev* not opened.

NOTES:

ixa_VME_read and *ixa_VME_write* are meaningful only on IXA VME boards.

ixa_VME_rmw

CALLING SEQUENCE:

```
#include <ixa.h>

IXABSP_STATUS ixa_VME_rmw ( VME_DEVICE    *vme_dev,
                            void          *vme_addr,
                            unsigned int   mask,
                            unsigned int   comp_val,
                            unsigned int   swap_val,
                            unsigned int  *return_val);
```

DESCRIPTION:

The *ixa_VME_rmw* function performs a VME read-modify-write (RMW) operation. RMW cycles on the VMEbus consist of a single read followed by a single write operation. The *vme_dev* parameter must be a valid VME_DEVICE pointer returned by *ixa_VME_open*. The VME attributes of the RMW operation are those specified when opening *vme_dev*. The *vme_addr* parameter specifies the VME address at which the RMW operation will occur. The *mask, comp_val* and *swap_val* parameters specify which bits in the read data are compared and modified in the RMW cycle. During a RMW, the VMEbus read data is bitwise compared with the *comp_val* and *mask* parameters. The valid compared and masked bits are then swapped using the *comp_val* parameter. Each masked bit that compares true is swapped with the corresponding bit in *swap_val*. A false comparison results in the original bit being written back. The data from the read portion of the RMW on the VMEbus is returned in *return_val*.

RETURN STATUS:

IXABSP_SUCCESS: successful completion.
IXABSP_DEVICE_NOT_OPENED: *vme_dev* not opened.

NOTES:

ixa_VME_rmw is meaningful only on IXA VME boards.

printf

CALLING SEQUENCE:

```
#include <ixa.h>

void printf ( char *format, ...);
```

DESCRIPTION:

printf will print 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:

Function *putchar* will transmit 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:

Function *puts* will output 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:

sprintf will print 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.

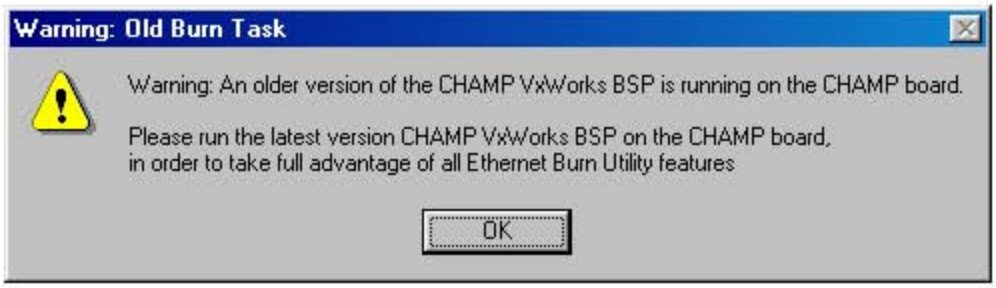


Figure 8.4 – Old Burn Task

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

VME configuration is not applicable for IXA4 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.

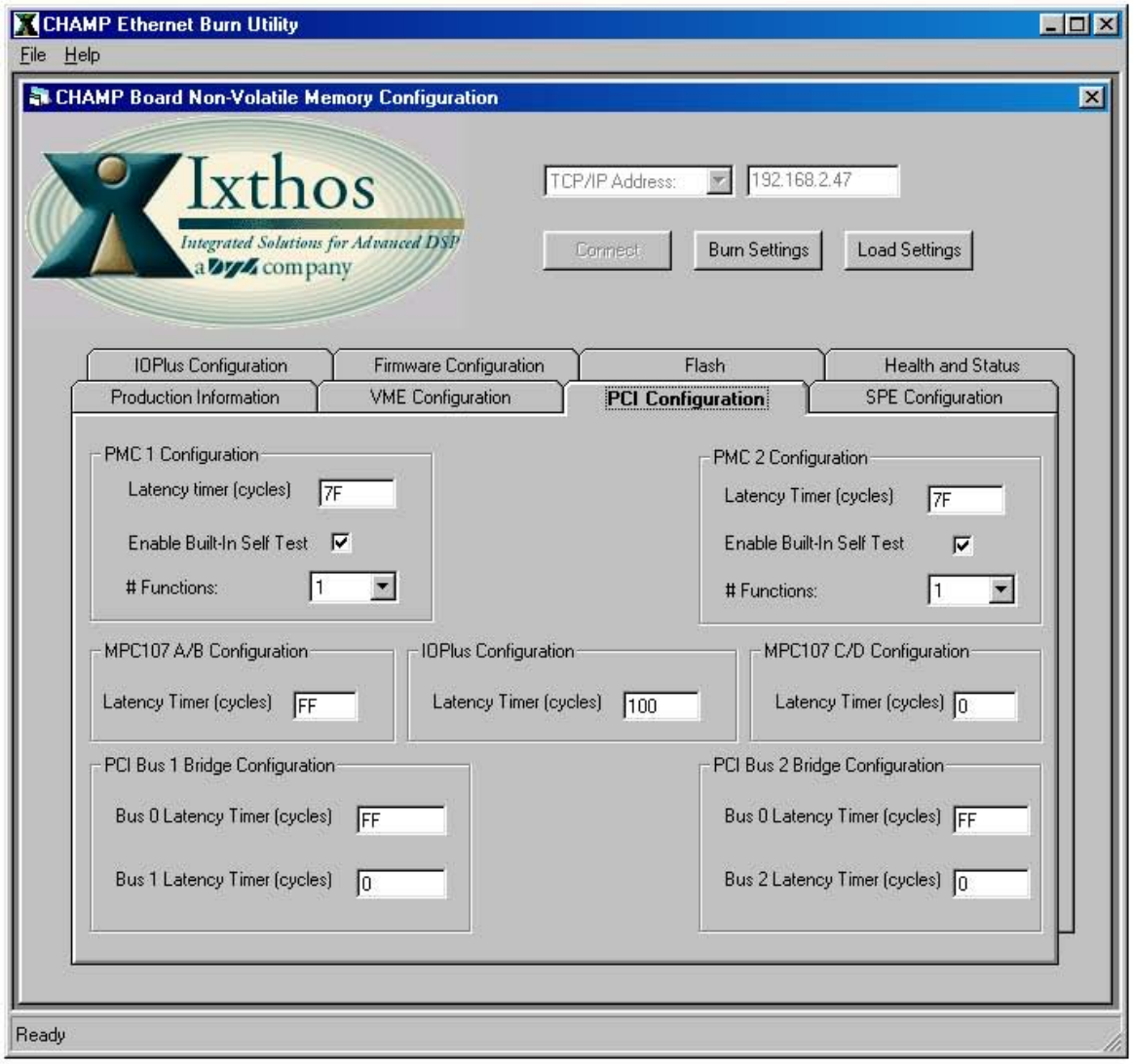


Figure 8.7 - PCI Configuration Page

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:

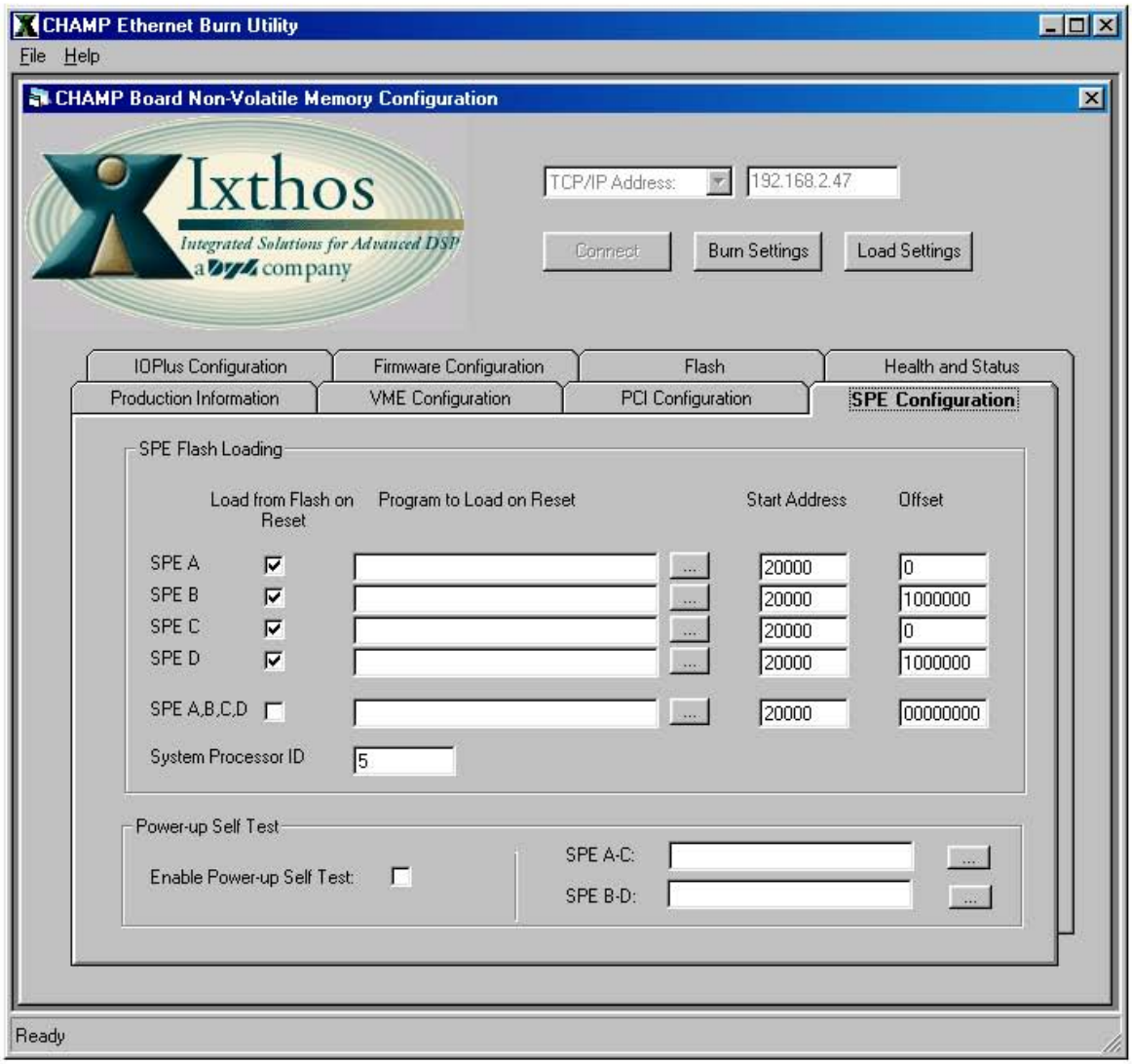


Figure 8.8 - SPE Configuration Page

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 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:

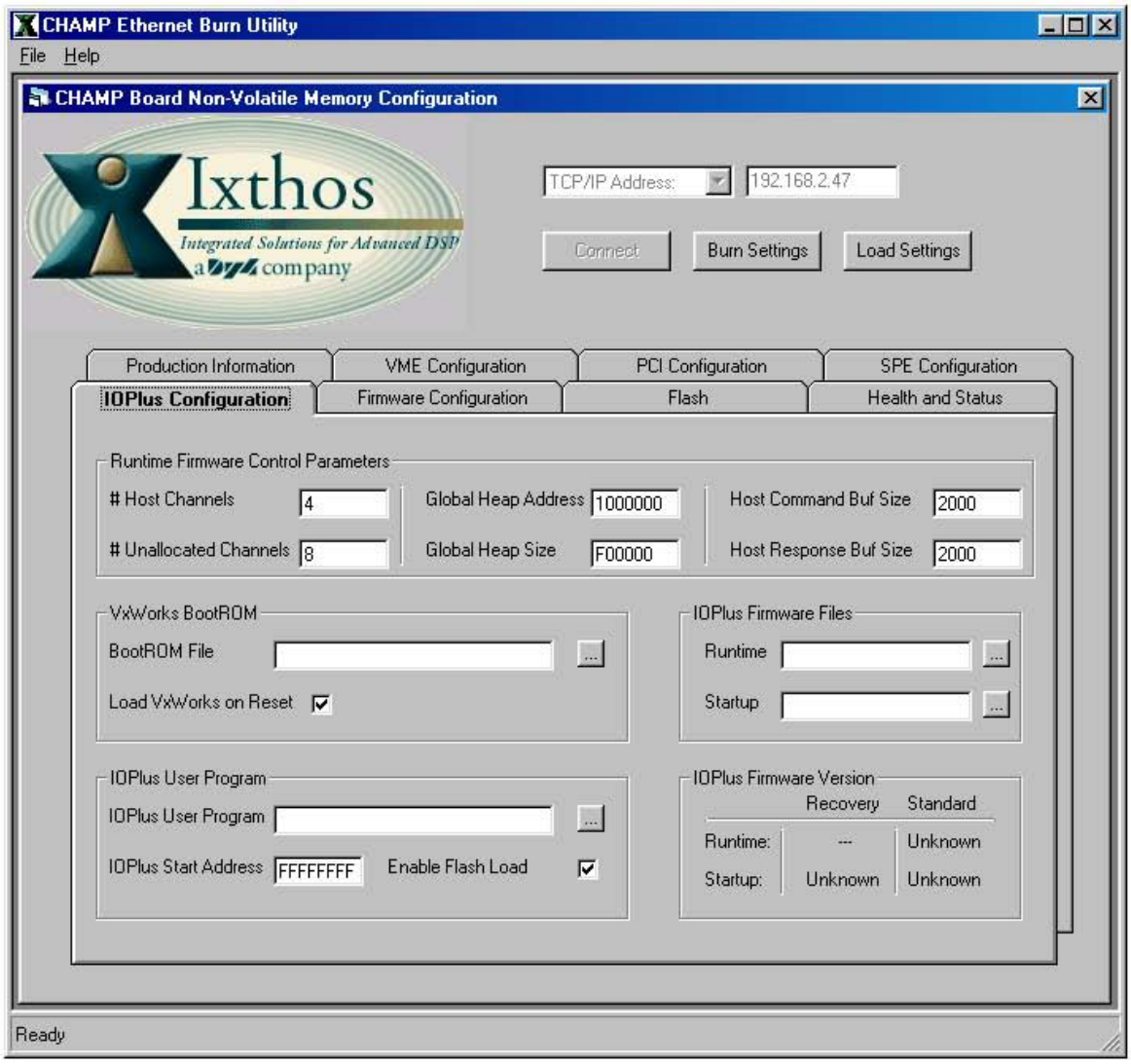


Figure 8.9 - IOPlus Configuration Page

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 information 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.

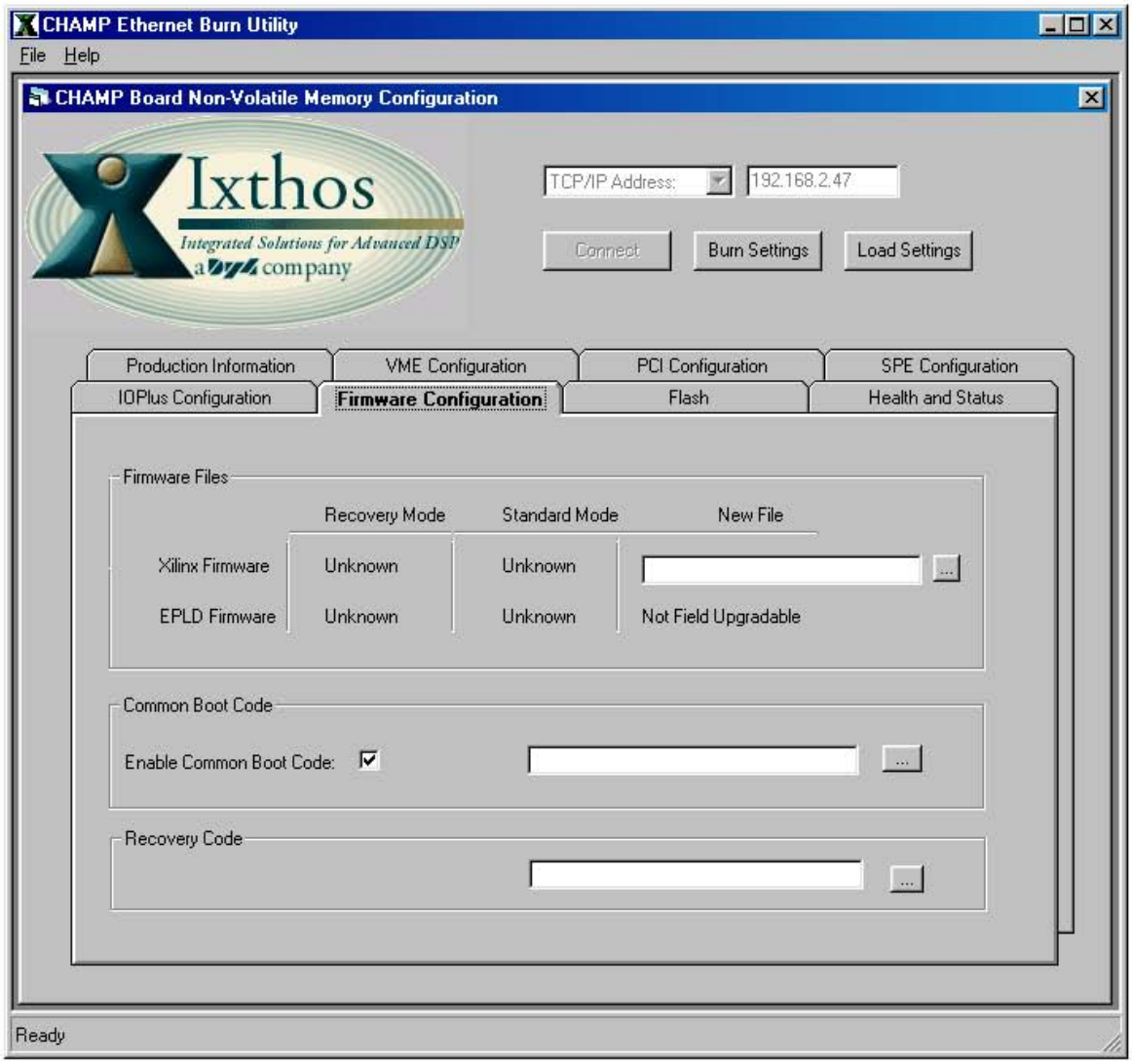


Figure 8.10 - Firmware Configuration Page

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 back plane.

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.

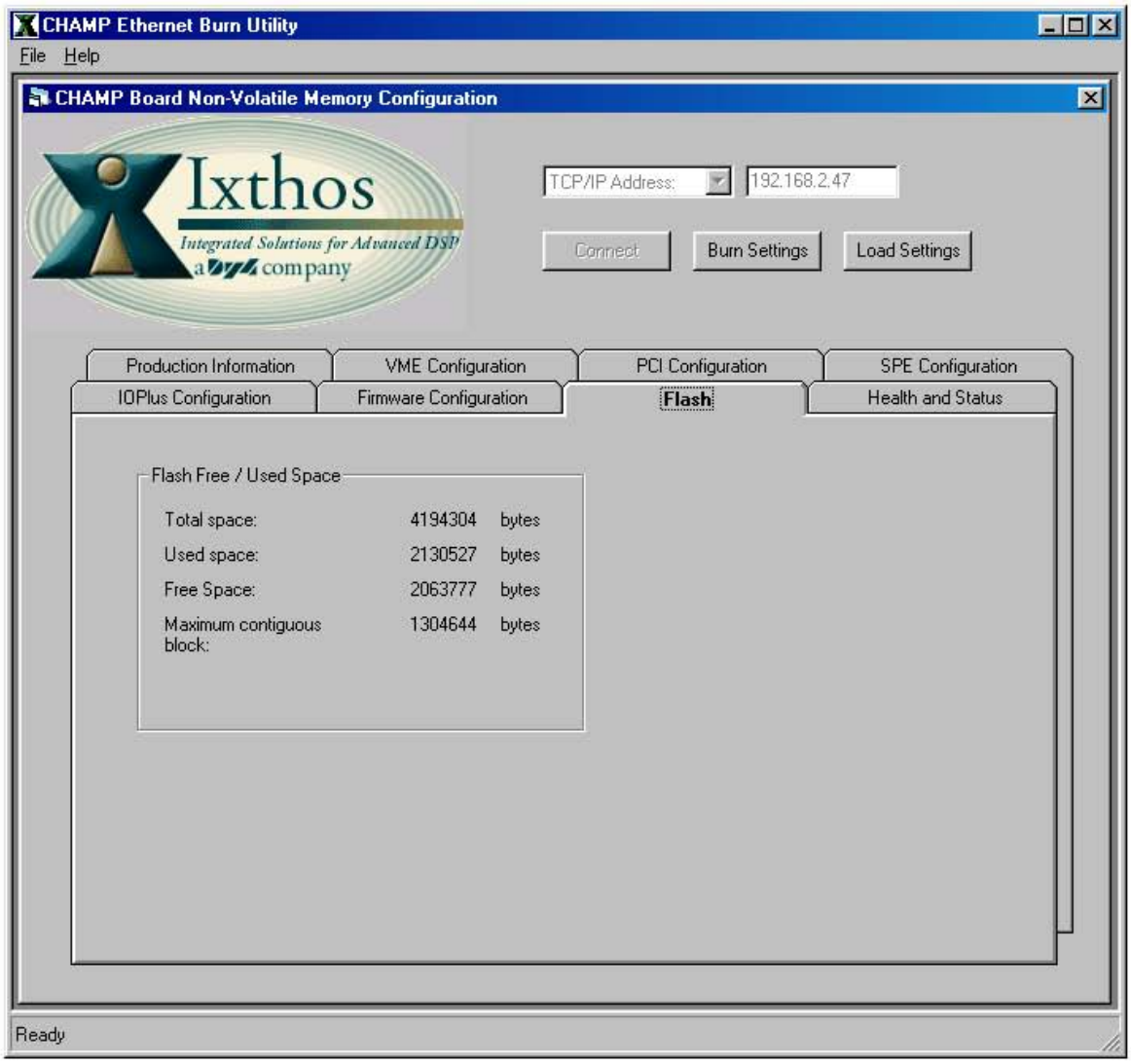


Figure 8.11 - FLASH Page

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.

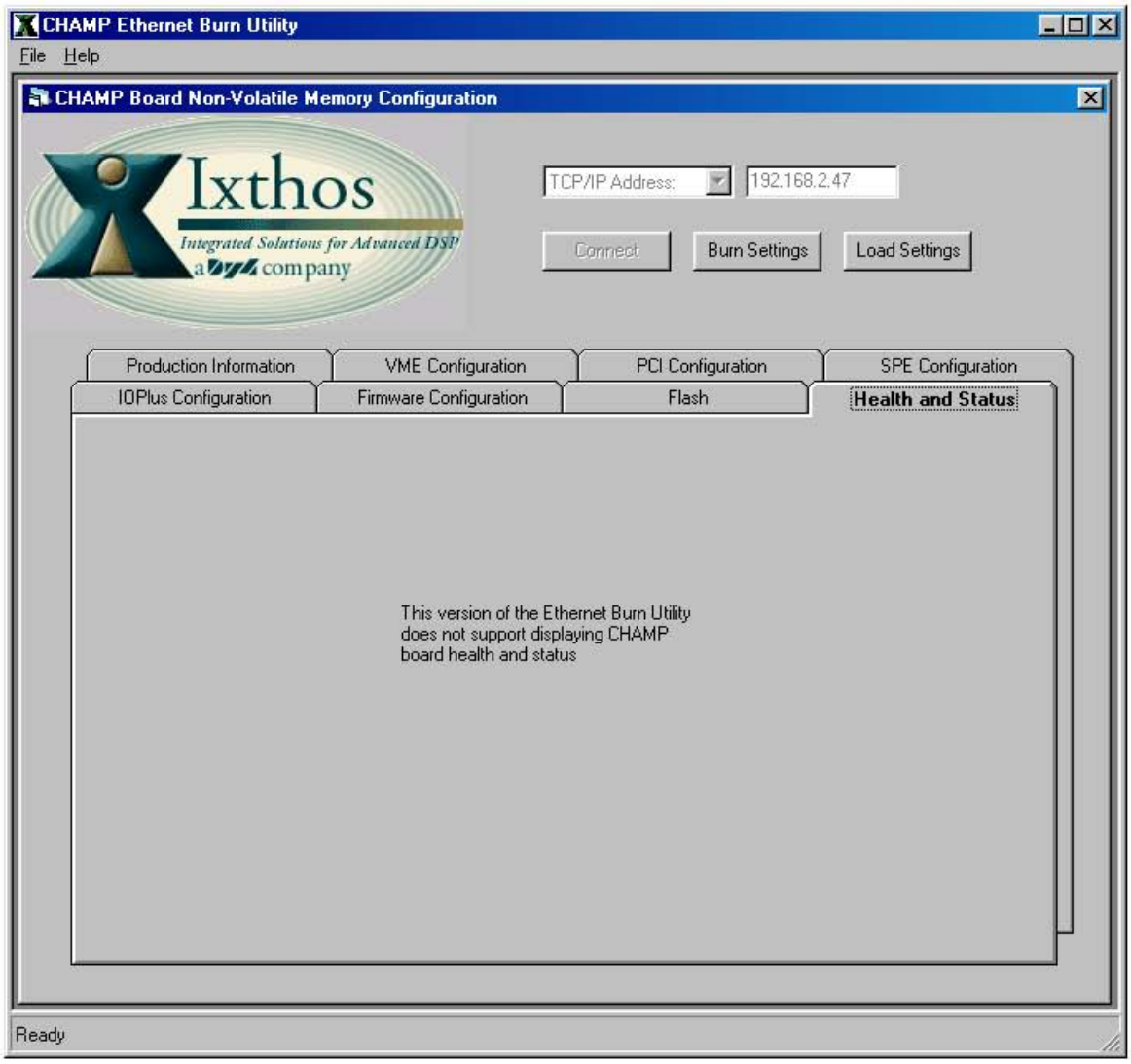


Figure 8.12 – Health and Status Page

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.

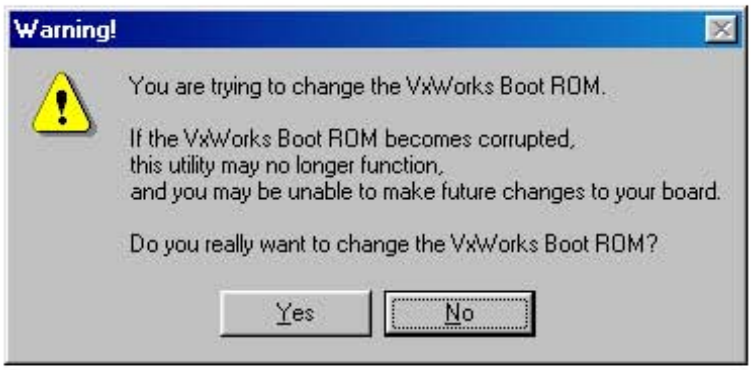


Figure 8.15 – Changing VxWorks Boot ROM

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.

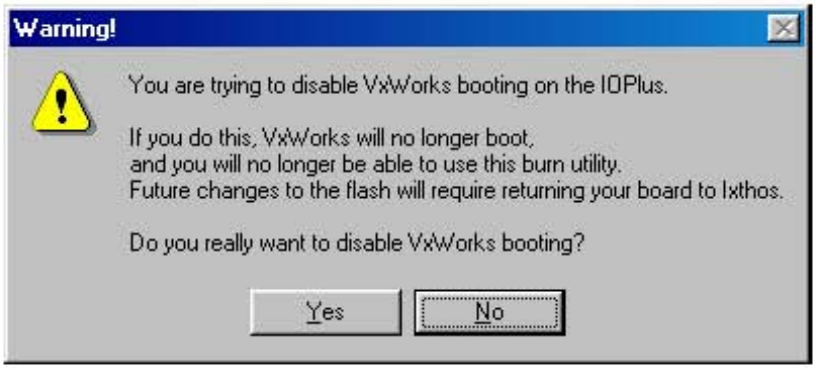


Figure 8.16 – Changing VxWorks Boot ROM

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).

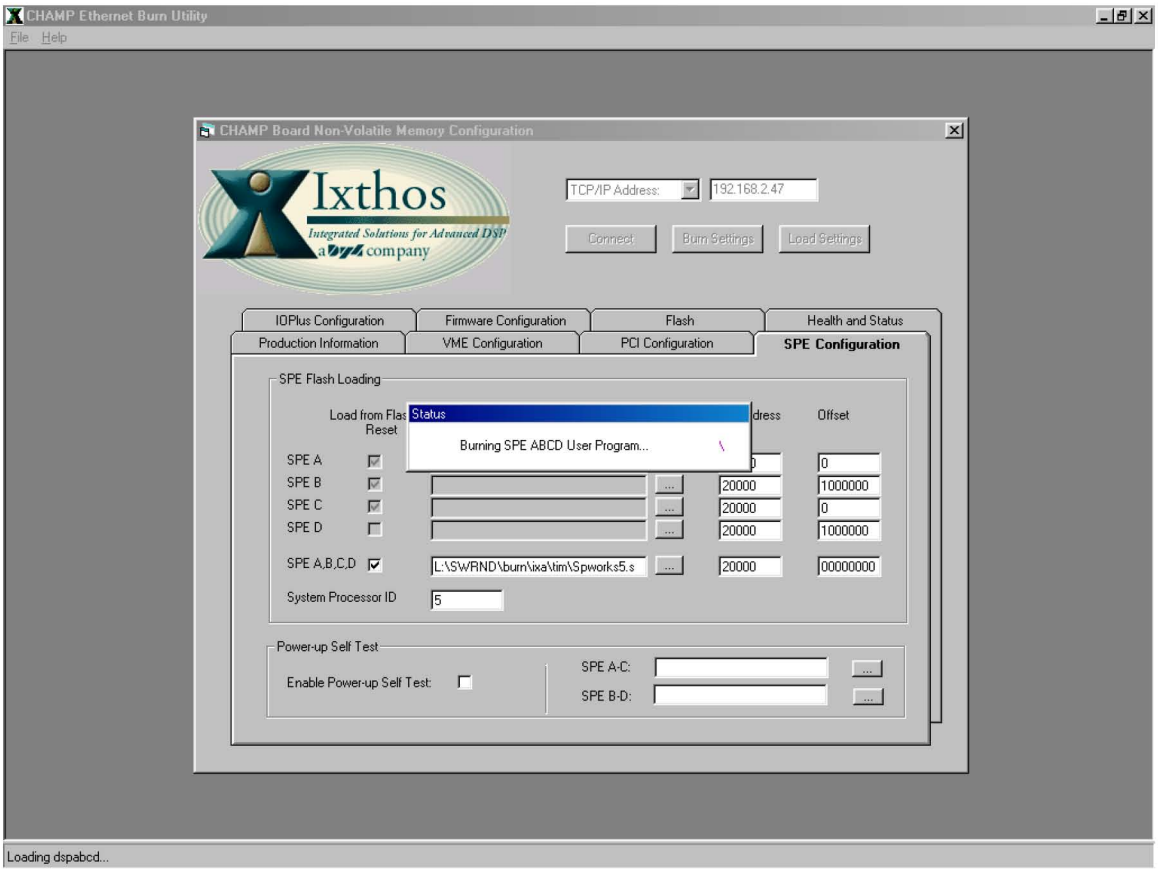


Figure 8.19 – Screen During Burn Operation

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 behavior 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 CPCI 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 *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 *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	Unused
1	Unused
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	Unused
12	Unused

Table 8.5 - FLASH configuration parameters (cont.)

Configuration Parameter Number	Description
13	Unused
17	Unused
18	Unused
19	Unused
20	Unused
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	Unused
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	Unused
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	Unused
65	Unused
66	Unused
67	Unused
.	…
95	Unused

Sample ldflash *Command File*

The sample *ldflash* command file shown below performs the following common tasks:

Attaches to a board at 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).

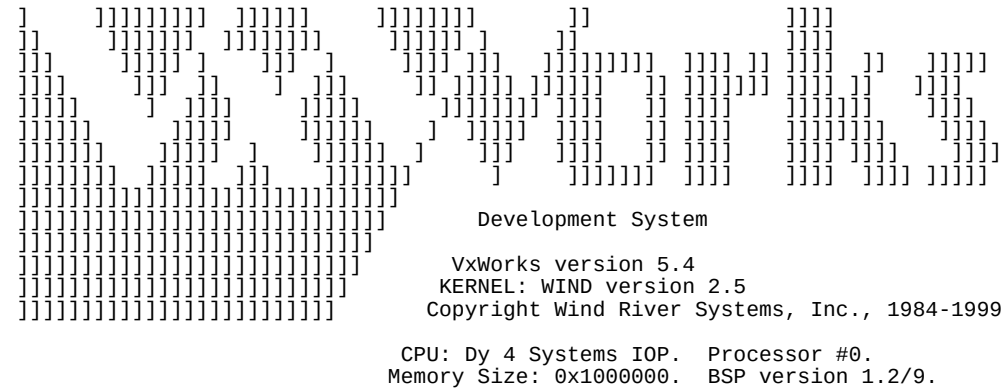
Sets configuration parameter 30 to global memory address 0x50000. After loading the global memory with the program stored in *iop.hex*, the IOPlus firmware will begin executing at global memory address 0x50000 (which is assumed to be the entry point for the program stored in *iop.hex*).

```
attach  80000000              ; Attach to board at 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 IXA4. 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 cPCI bus. To use HostAPI requires a cPCI bus interface driver that conforms to an Dy 4 Systems convention, shown in the Figure 9.1 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.

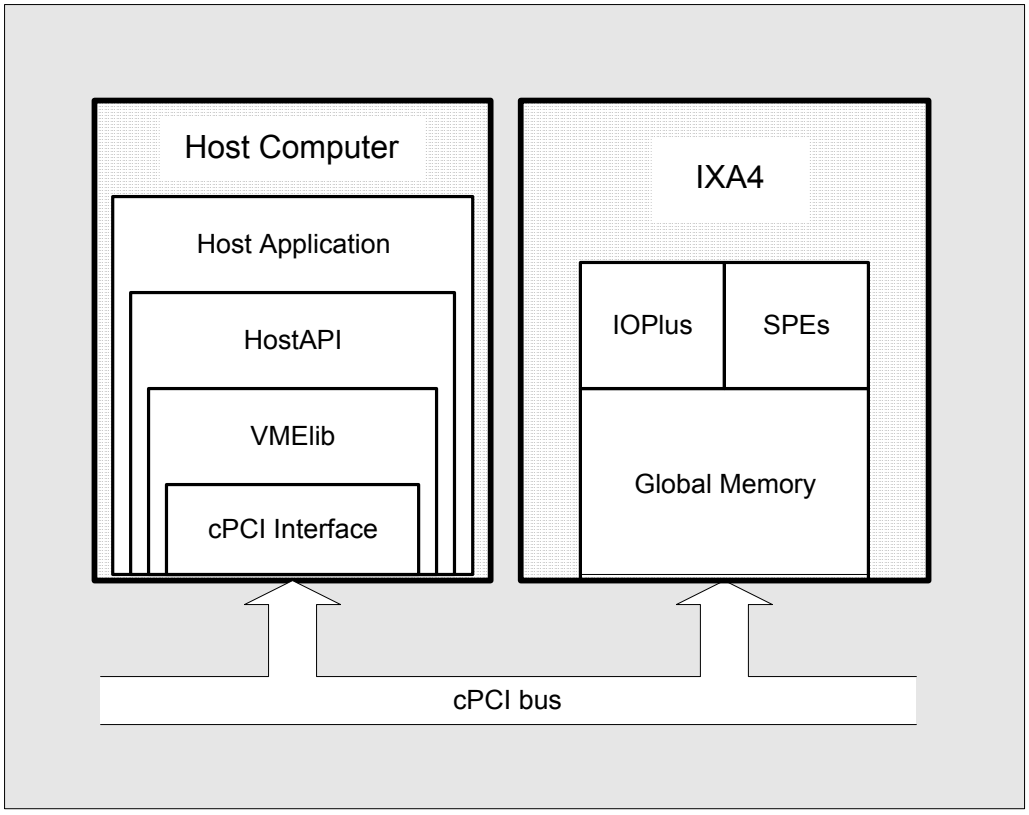


Figure 9.1 - HostAPI Relationship Diagram

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.

The name VMElib may look a little out of place in context with a cPCI board, but by maintaining the naming conventions used in the VME product base, Dy 4 Systems can extend the HostAPI library across its entire product base. The process of doing this does cause the VMElib to filter application request from a VME frame of reference to a PCI frame of reference. This will become more apparent in the function definitions that follow.

9.2 Porting HostAPI

HostAPI, communicates with the IXA4 using the cPCI bus. Dy 4 Systems has ported HostAPI to a number of cPCI 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 cPCI 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 cPCI bus interface driver.
2. Constructing a set of functions (VMElib) to interface the cPCI bus interface driver to the HostAPI software.
3. Compiling, linking, and testing the HostAPI software with the VMElib and cPCI bus 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 cPCI bus interface driver for the required operating system?

Does the operating system come with a cPCI bus 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 cPCI bus interface driver for your application. You should still review the next section to determine if the cPCI bus interface driver is capable of satisfying the requirements of a HostAPI port.

cPCI bus Interface Driver Requirements

A cPCI bus interface driver for HostAPI needs only support basic cPCI bus read/write capabilities. The host software does not require that interrupts be issued or serviced by the driver.

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 cPCI bus interface driver software from the HostAPI software. The interface between the HostAPI and the cPCI bus 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 cPCI bus 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 cPCI 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 for a VME address modifier. It is not applicable to cPCI implementations an can be ignored.

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 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 cPCI 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 cPCI 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 cPCI 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 vem_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 cPCI 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 cPCI device driver with the HostAPI library software, the cPCI 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 cPCI bus analyzer can be used to confirm proper address modifier and address mode values.

Test Equipment.

Testing will require a host chassis, the cPCI host computer, the test software, test driver software, and a slave cPCI card.

A software test harness should be written that accesses the slave card from the host. The test harness should perform a *vme_open*, 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 cPCI bus analyzer can be used to confirm the address, length, and address modifier bits for all read and write operations. If no cPCI bus 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 cPCI 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 IXA4 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 IXA4 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   cpci_address,
                                  void          **board_pointer );
```

DESCRIPTION:

host_board_open attaches to an IXA board with the specified cPCI 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 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 cPCI 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_BOARD,
HOST_RESET_PCI_BUS,
HOST_RESET_CLUSTER_A,
HOST_RESET_CLUSTER_B,
HOST_RESET_SPE_A,
HOST_RESET_SPE_B,
HOST_RESET_SPE_C,
HOST_RESET_SPE_D.
HOST_RELEASE_SPE_A,
HOST_RELEASE_SPE_B,
HOST_RELEASE_SPE_C,
HOST_RELEASE_SPE_D

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 IXA4. 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_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 local 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 local PCI address of the allocated memory, not the cPCI 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  cpci_address,
                                  char           *filename,
                                  char           *section_name );
```

DESCRIPTION:

host_load_flash burns a program or file in s-record format into the IXA4 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 local PCI address must be converted to the cPCI address of the allocated region before accessing the memory through the cPCI bus. The local PCI address can be converted to a cPCI address using the following formula:

cPCI_Address = cPCI_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 cPCI 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       cpci_address,
                                    unsigned long       length_in_bytes,
                                    HOST_ADDRESS_SPACE  address_space,
                                    void                **resource_ptr
                                    );
```

DESCRIPTION:

host_map_resource maps an IXA4 board resource onto the cPCI bus, so that the board resource can be accessed from other boards on the cPCI bus. The type of resource, cPCI address at which the resource should be mapped, and the length (in bytes) of the cPCI address window to map, must 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 IXA4 board can be created.

address_space is not needed for the IXA4.

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
```

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 cpci_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 cpci_address,
                                       void          *ptr,
                                       unsigned long max_words );
```

DESCRIPTION:

host_read_board_info reads the specified data item from the IXA4 board information structure. The function requires the cPCI base address of the IXA4 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 IXA4 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 cPCI-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 cPCI bus, starting at the specified cPCI address. The data read from the cPCI bus is placed into a buffer pointed to by *ptr.*

RETURN STATUS:

HOST_NO_ERROR:	completed successfully.
HOST_VME_READ_ERROR:	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 cPCI bus, starting at the specified cPCI address. The data written to the cPCI bus is placed into a buffer pointed to by *ptr*.

RETURN STATUS:

HOST_NO_ERROR:	completed successfully.
HOST_VME_WRITE_ERROR:	cPCI 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.

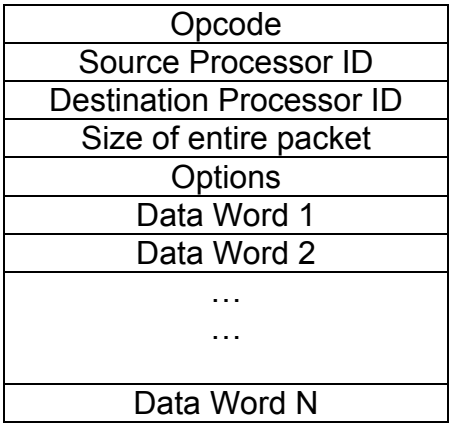


Figure A.1 - Command Packet

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:

6) Get address of host list table from board information structure (see section 5.5)
7) Get maximum number of allowed host processes from board information structure
8) Scan host list table for free entries (entries which are non-zero)
9) When a non-zero entry is found, perform a 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.
10) 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:

4) Get address of host list table from board information structure.
5) Determine offset in "host list table" using the following formula:
Offset = "processor ID" – "maximum # of SPEs on board" - 1
6) Write a zero to this offset in the "host list table" (note that this write does not need to be atomic).

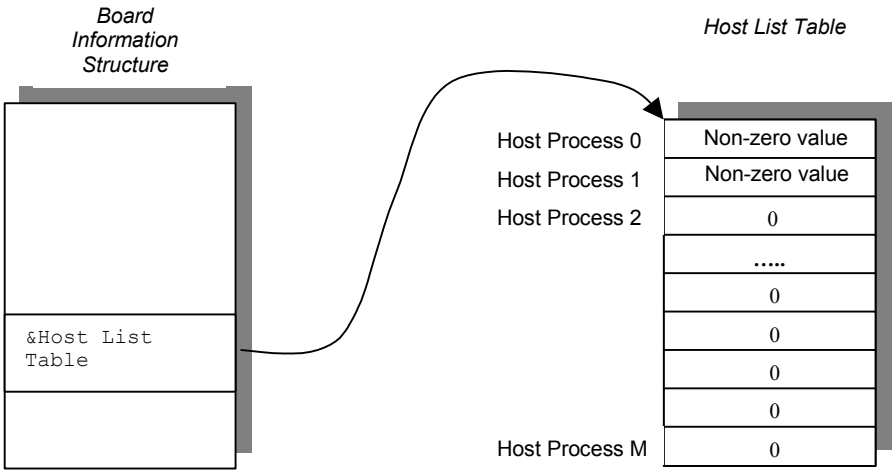


Figure A.3 - Host Process ID Assignment with Two Host Processes Already Attached

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 back plane), 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.

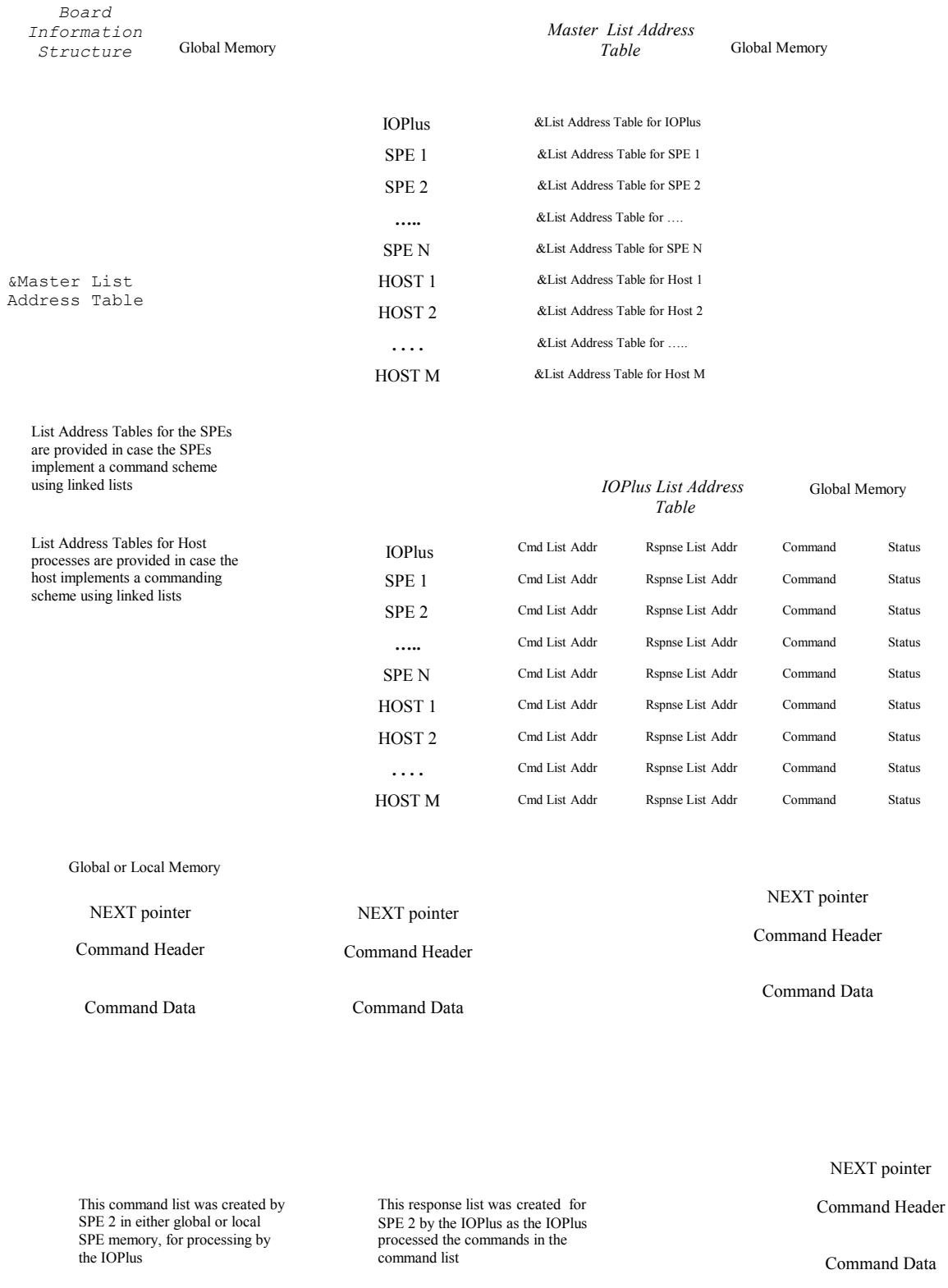


Figure A.4 - All List Address Table Entries in Global Memory

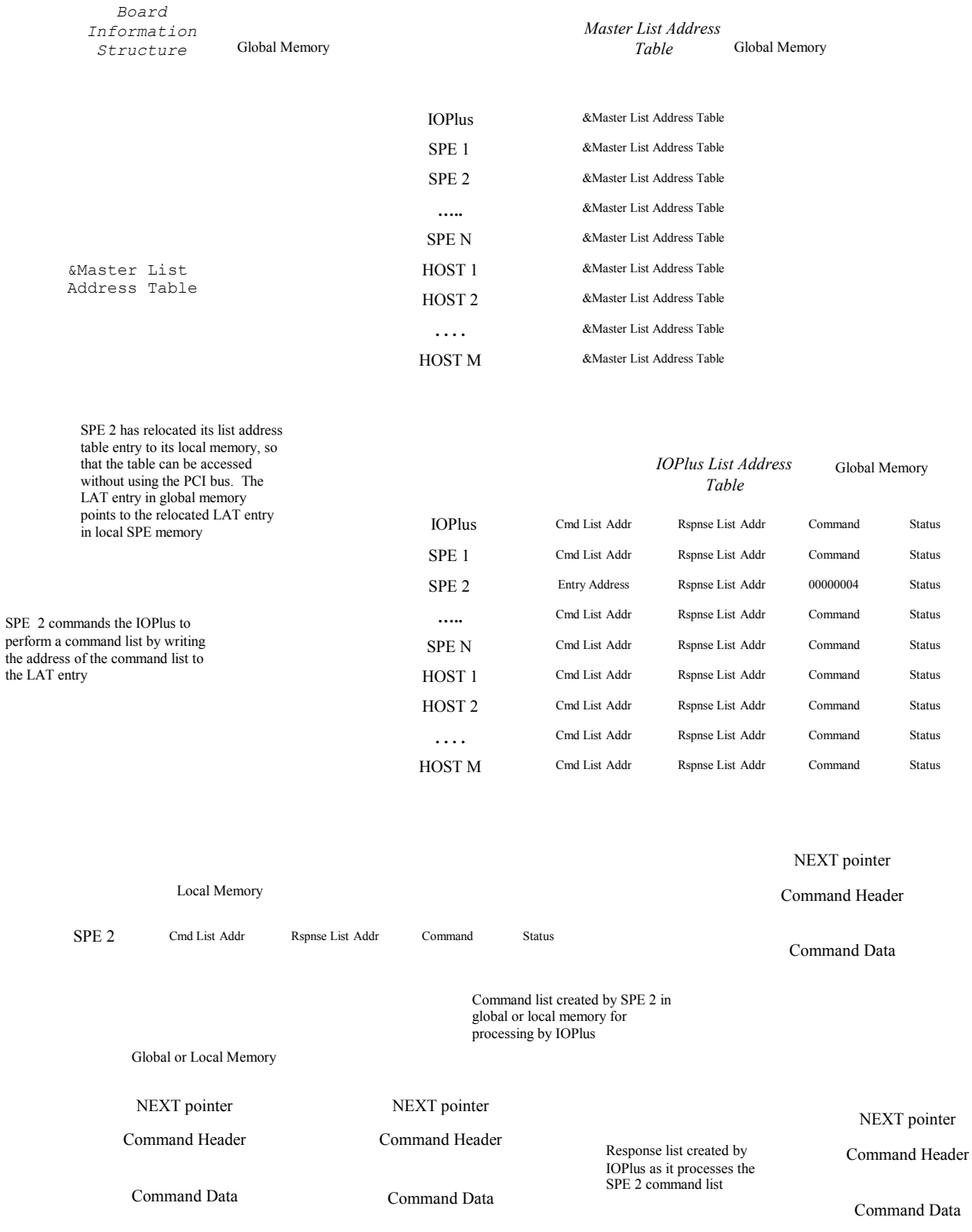


Figure A.5 - List Address Table Entry Relocated to Local Memory

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 (e.g. 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:

8) 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:
9) Allocate 16 bytes of local memory to hold the relocated "list address table" entry.
10) Locate the "list address table" using the pointer contained in the "master list address table".
11) Locate the appropriate "list address table" entry using the software component's relative processor ID as an index into the "list address table".
12) 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.
13) Set the relocation bit in the "command option" field of the "list address table" entry in global memory.
14) The table entry is now relocated from global to local memory.

Initiating:

7) 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.
8) 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.
9) 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.
10) 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.
11) Generate an interrupt to the IOPlus.
12) 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:

3) 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.
4) 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_gen*. 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

6) 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.

7) Entry names must be unique.

8) Reserved entry names (as shown above) have special meaning to the IOPlus software, and should not be used by application software.

9) 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.

10) 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