March 1991
Order Number: 311785-002

iPSC®/2 and iPSC®/860
VME INTERFACE
REFERENCE MANUAL

intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	10/89
---	Revised by Change Notice 311876-001	02/90
-002	Revision	03/91

CAUTION

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

Controller Boards (and their attached cables) which are plugged into the BIA Board may cause additional interference. The user is responsible for ensuring that any added equipment does not cause the system to fail FCC rules. The user is responsible for resolving radiation problems caused by the added equipment. Cables entering the cube must be properly shielded, with the shield terminated at the point of entry.

RESTRICTED RIGHTS

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

PREFACE

This manual describes the VME Bus Interface Adapter (BIA).

NOTE

In this manual, the term "iPSC system(s)" refers to any or all of the following SSD products: iPSC®/2, iPSC®/2S, iPSC®/860, iPSC®/860S, and iPSC®/860Plus.

The BIA is an interface between an iPSC I/O node and a VME controller board.

The manual assumes that you are a system programmer familiar with the C programming language, the AT&T assembler, the iPSC/2 Parallel Supercomputer, and the VMEbus specification. The manual describes how to install the BIA, configure it, and write a driver for it. The driver runs on an iPSC/2 I/O node under the NX/2 operating system.

ORGANIZATION

Chapter 1	"Introduction," is an overview of the architecture and operation of the BIA.
Chapter 2	"Installation," describes how to install the BIA in a standard iPSC/2 cabinet. This chapter contains a complete list of jumper positions.
Chapter 3	"Hardware Reference," contains hardware information needed to write a BIA driver. The chapter describes the PBX bus, the bus connecting the I/O node and the BIA. In addition, the chapter contains a complete list of the VMEbus signals used by the BIA.

Chapter 4 "Programmer's Reference," describes how your application program might make use of the BIA driver you write. It defines the memory mapping between the I/O node and the BIA and describes the BIA's programmable registers. It also describes interrupt handling.

Appendix A "Source Code for Example Driver," contains a complete C source code listing of a simple working driver. The driver is intended to be used as a template for drivers that you may write.

NOTATIONAL CONVENTIONS

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be used exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

`Bold-Italic-Monospace`
Identifies user input (what you enter in response to some prompt).

`Bold-Monospace`
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

`<Break>` **`<s>`** **`<Ctrl-Alt-Del>`**

[ ] (Brackets) Surround optional items.

. . . (Ellipsis dots) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items of which you may select only one.

{ } (Braces) Surround two or more items of which you must select one.

APPLICABLE DOCUMENTS

iPSC®/2 and iPSC®/860 Programmer's Reference Manual
Describes iPSC system commands and system calls (both C and Fortran).

iPSC®/2 and iPSC®/860 System Administrator's Guide
Describes the system administration tasks related to operating and maintaining an iPSC system.

iPSC®/2 and iPSC®/860 User's Guide
Overviews the iPSC system, including hardware and software architectures. Tells how to develop and run programs.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

INTRODUCTION 1-1

BIA ARCHITECTURE 1-2

BIA REGISTERS 1-5

THE BIA DRIVER 1-5

CHAPTER 2
INSTALLATION

INTRODUCTION 2-1

HARDWARE CONFIGURATION 2-1

HARDWARE INSTALLATION 2-3

HARDWARE REMOVAL 2-3

SOFTWARE CONFIGURATION 2-3

CHAPTER 3
HARDWARE REFERENCE

INTRODUCTION 3-1

SIGNAL DEFINITIONS 3-1

BIA ACCESS 3-3

INTERRUPTS 3-4

BIA REGISTERS 3-5

- BIA Control Register 3-5
- BIA ID Register 3-6
- Clear Latched Interrupt Status Register 3-7
- Interrupt Mask Register 3-7
- Interrupt Status Register 3-8
- Loopback Test Register 3-10
- VME Address Register 3-10

BIA SPECIFICATIONS 3-11

- Physical 3-11
- Bus Specifications 3-12
- Electrical 3-12
- Environmental 3-13
- Configuration 3-13

CHAPTER 4
PROGRAMMER'S REFERENCE

INTRODUCTION 4-1

WRITING THE BIA DRIVER 4-1

ASSEMBLY ROUTINES 4-2

ACCESSING THE BIA REGISTERS 4-4
- Turning on Memory Mapping 4-4
- Converting from Physical to Logical 4-4
- Example: Writing the BIA Control Register 4-4

ACCESSING VME MEMORY 4-6
- Example: Reading a VME Address 4-8
- Block Mode 4-10
- The Byte Swapping Network 4-11
  - DATA ALIGNMENT 4-13
    - Aligned Transfers 4-13
    - Unaligned Transfers 4-15
  - ADDRESS TRANSLATION 4-16
  - BYTE SWAPPING 4-18
- 32-Bit Mode 4-19
- 16-Bit Mode 4-21
- 8-Bit Mode 4-23
- Array Mode 4-25

HANDLING INTERRUPTS 4-29

LOADING THE BIA DRIVER 4-32

DETERMINING THE DIRECT NODE NUMBER 4-33

COMMUNICATING WITH THE APPLICATION 4-33

BIA DIAGNOSTIC TESTS 4-34

APPENDIX A
SOURCE CODE FOR EXAMPLE DRIVER

INTRODUCTION A-1

ad.h A-1

bia.h A-4

driver.h A-7

port.s A-8

in: A-8

out: A-9

ad_global.c A-10

bia_global.c A-11

driver.c A-12

DtIntWait() A-13

DtAin(chan, inter) A-14

short DtAout(chan, val) A-15

main(argc, argv) A-15

interface.c A-21

long biactl(handle, command, buffer, size) A-21

openbia() A-22

aout(handle, chan, val) A-23

utils.c A-24

unsigned int *create_pointer(phy_addr) A-24

bia_init() A-24

clear_bia_isr() A-26

ack_bia_interrupt() A-26

nak_bia_interrupt() A-26

disable_pbx_interrupt() A-27

enable_pbx_interrupt() A-27

enable_bia_interrupt() A-27
disable_bia_interrupt() A-28
unsigned long set_array_comp(x) A-28
unsigned long set_32_bit(x) A-28
unsigned short set_16_bit(x) A-29
unsigned char set_8_bit(x) A-29
enable_lbx() A-29
unsigned long make_dt_pointer(phaddr) A-30
unsigned long make_bia_pointer(phaddr) A-30
void setup_page_mod(phaddr) A-31
init_board() A-31
printirq() A-32
char get_VME_vec(base_addr, vec) A-33

LIST OF ILLUSTRATIONS

Figure 1-1. Block Diagram of the I/O Node with a BIA Board 1-2

Figure 1-2. An I/O Node with a BIA and a VME Controller Board 1-3

Figure 1-3. Block Diagram of the BIA 1-4

Figure 2-1. Location of BIA Jumpers 2-2

Figure 3-1. The PBX/VME Interface 3-2

Figure 3-2. Memory Read 3-3

Figure 3-3. Memory Write 3-4

Figure 4-1. The VME Address 4-6

Figure 4-2. Accessing a Physical Address 4-7

Figure 4-3. Reading and Writing VME Addresses 4-9

Figure 4-4. Data Alignment for a 16-Bit Word 4-13

Figure 4-5. Data Alignment for Byte 0 4-14

Figure 4-6. Data Alignment for Byte 1 4-14

Figure 4-7. Data Alignment for Bytes 2 and 3 4-15

Figure 4-8. Address Translation for Byte 0 4-16

Figure 4-9. Address Translation for a 16-Bit Word 4-17

Figure 4-10. The Elements of the String "abcd" are in Different Addresses 4-18

Figure 4-11. When Byte-Swapped, the Elements of the String "abcd" are in the Same Addresses 4-18

Figure 4-12. 32-Bit Mode: Writing and Reading a 32-Bit Word 4-19

Figure 4-13. 16-Bit Mode: Writing and Reading a 16-Bit Word at Address 0 4-21

Figure 4-14. 8-Bit Mode: Writing and Reading a Byte at Address 0 4-23

Figure 4-15. Array Mode: Writing and Reading a 32-Bit Word at Address 0 4-25

LIST OF ILLUSTRATIONS

Figure 4-16. Array Mode: Writing and Reading a 16-Bit Word at Address 0 4-26

Figure 4-17. Array Mode: Writing and Reading a 16-Bit Word at Address 2 4-27

LIST OF TABLES

Table 2-1. Jumper Description for the BIA Board 2-1

Table 3-1. Bit Pattern for the BIA Control Register 3-6

Table 3-2. Bit Pattern for the BIA ID Register 3-7

Table 3-3. Bit Pattern for the Interrupt Mask Register 3-8

Table 3-4. Bit Pattern for the Interrupt Status Register 3-9

Table 3-5. Bit Pattern for the VME Address Register 3-11

Table 4-1. VME Memory Modes 4-12

Table 4-2. 32-Bit Mode 4-20

Table 4-3. 16-Bit Mode 4-22

Table 4-4. 8-Bit Mode 4-24

Table 4-5. Array Mode 4-28

INTRODUCTION 1

INTRODUCTION

The VME Bus Interface Adapter (BIA) connects an iPSC I/O node to a VMEbus compatible board. The BIA accepts standard 6U VME Rev C.1 Eurocard boards.

The VMEbus connects data processing, data storage, and peripheral control devices in a closely coupled hardware configuration. Typical VME devices that connect to a BIA include LAN controller boards, bus-to-bus links (VME to VME, VME to MULTIBUS® I), data acquisition boards, tape controller boards, DR11W controller boards (a type of bus adapter), and graphics controller boards.

The BIA resides in the odd-numbered slot in a standard cabinet. It is adjacent to its controlling I/O node. The VME controller board plugs into the BIA.

Figure 1-1 is a block diagram of an I/O node with a BIA board. Notice that the I/O node uses a bus called the PBX bus to communicate with the BIA. The BIA translates the 386™ signals on the PBX into VME signals understood by the VME controller board. Figure 1-2 shows an I/O node with a BIA and a VME controller board in a standard cabinet.

I/O nodes reside in a standard iPSC cabinet and are part of the Concurrent I/O system. I/O nodes are like compute nodes, but they have only a single channel on their Direct Connect Modules (DCMs).

Compute nodes have an LBX bus instead of the PBX bus. They use their LBX bus to communicate with an optional vector (VX) board. I/O nodes purchased with a BIA bring out the PBX in place of the LBX bus and cannot have an associated vector board. Some I/O nodes purchased without a BIA, however, do have the LBX bus, and hence will not work with a BIA.

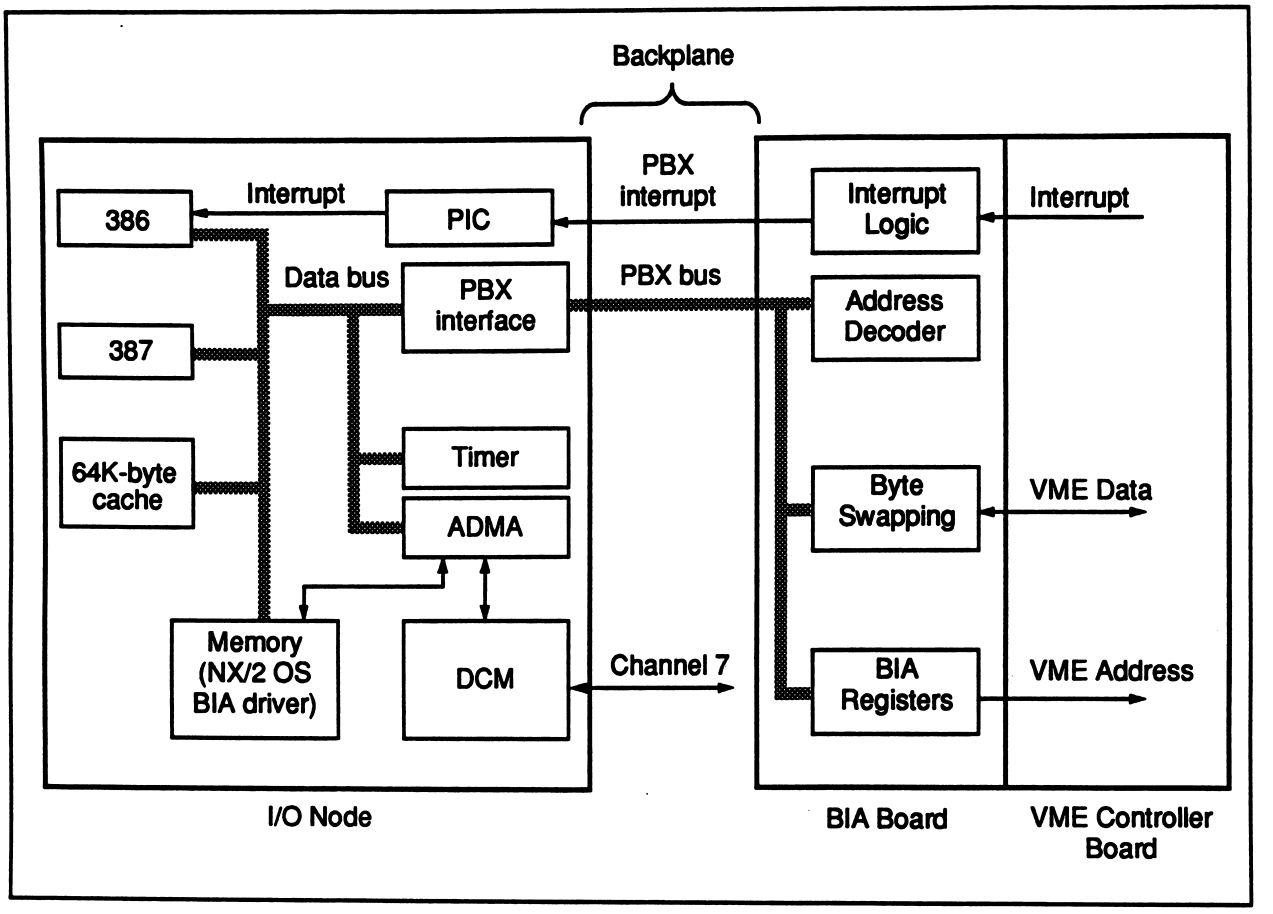


Figure 1-1. Block Diagram of the I/O Node with a BIA Board

BIA ARCHITECTURE

To use the BIA, you must write a BIA driver. This driver runs on the BIA's companion I/O node. Typically, your application runs on a compute node and sends messages to the BIA driver on the I/O node. The BIA driver must interpret those messages and communicate with the BIA.

The I/O node acts as a bus master for its local VME interface. The VME controller board acts as a bus slave. That is, the I/O node expects to do all the reading and writing across the VMEbus. The VME controller board may send interrupts to the BIA, but otherwise it does not control the bus. The BIA may interrupt the I/O node, which must then read the BIA's Interrupt Status Register to determine the nature of the interrupt.

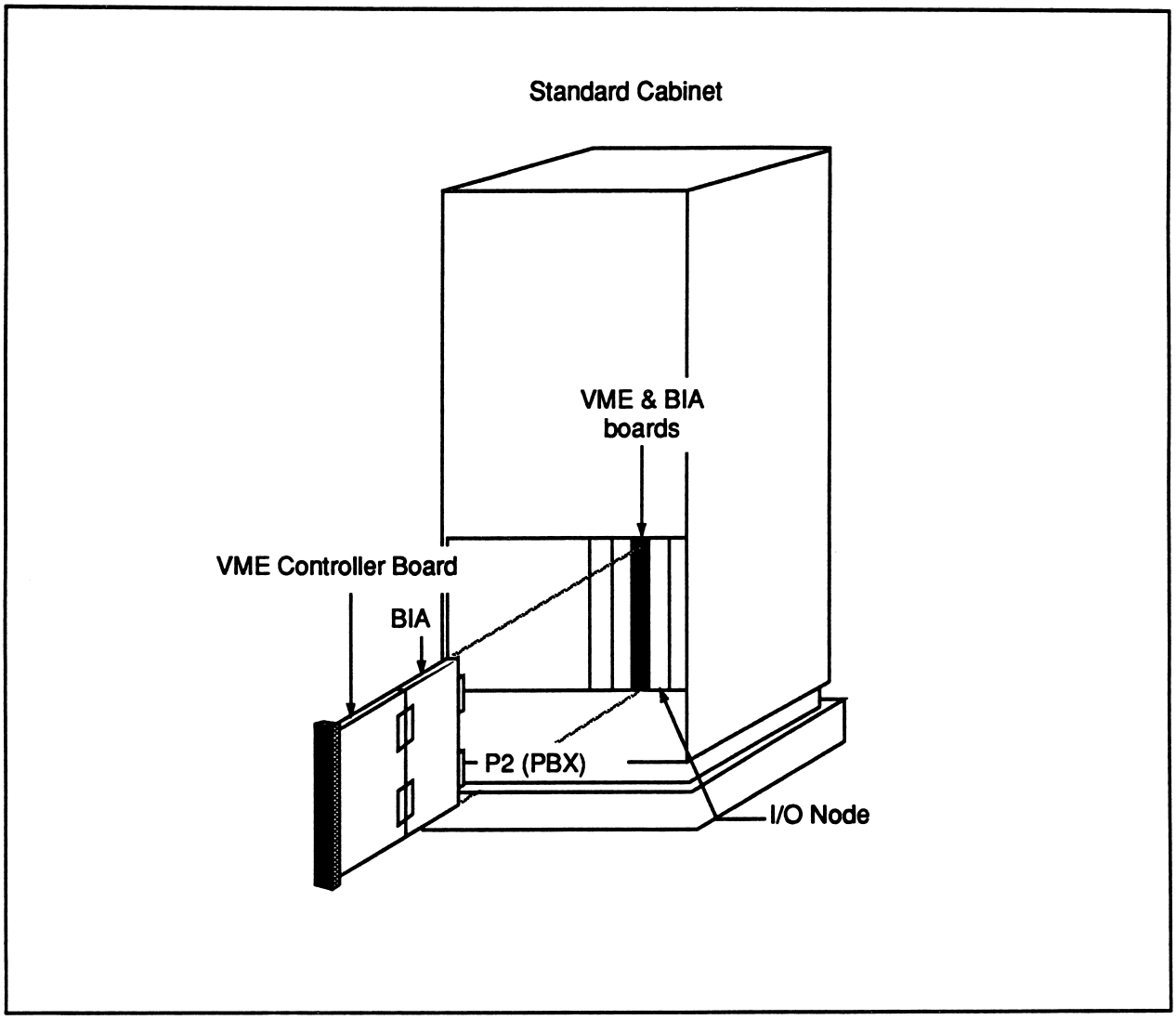


Figure 1-2. An I/O Node with a BIA and a VME Controller Board

An I/O node that has a BIA is limited to 8M bytes of memory. The 8M bytes of address space beginning at address 0x0080 0000 are reserved for communication with the BIA. The first 4M bytes above 8M bytes are for VME access. The BIA treats these 4M bytes as a window into the VME address space. The next 4M bytes are mapped to BIA registers. The BIA driver sets BIA modes and performs BIA initialization and testing by writing and reading these registers.

Figure 1-3 is a block diagram of the BIA, shown from a programmer's point of view.

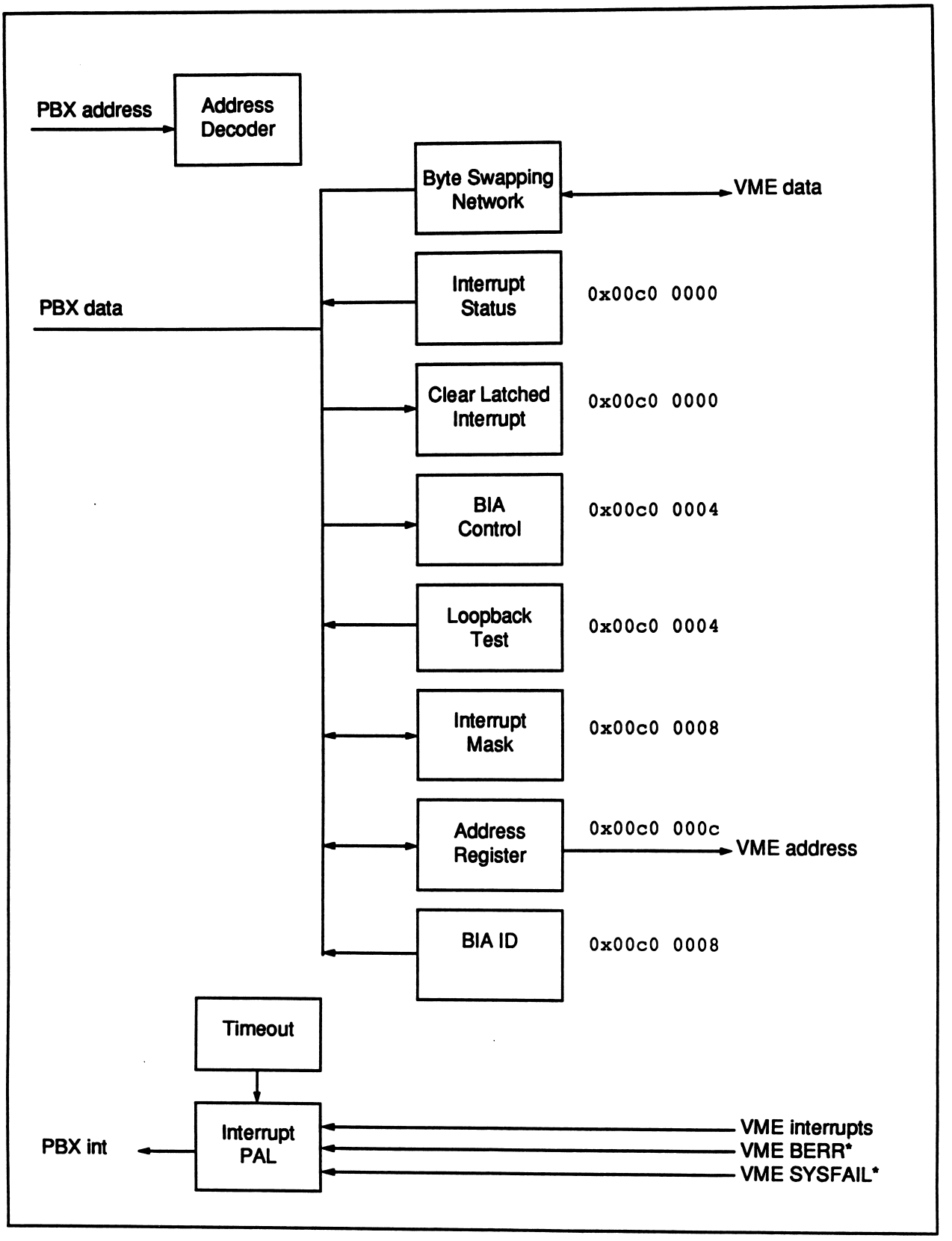


Figure 1-3. Block Diagram of the BIA

BIA REGISTERS

The BIA provides the following registers:

BIA Control Register	This register determines the BIA modes. By writing bits in this register, you can set the BIA's operating mode. For example, you can set the BIA to do VME block mode transfers. You can also perform various control actions needed during operation. For example, you can perform a VME interrupt acknowledge cycle and cause the VME address and data lines driven by the BIA to enter a high impedance state.
BIA ID Register	This register contains the BIA's revision number.
Clear Latched Interrupt Status	Two interrupts from the BIA are latched. These are a bus timeout error (**BIA BUS TIMEOUT***) and the VME bus error signal (**VME BERR***). Writing anything to this register clears these interrupts.
Interrupt Mask Register	By writing this register, you can mask off particular VME interrupts.
Interrupt Status Register	By reading this register, you can determine the nature of a PBX interrupt from the BIA. All seven VME interrupts are supported.
Loopback Test Register	This register is used when testing how the BIA performs byte swapping. Reading this register returns swapped data.
VME Address Register	By writing this register you can set the location of the BIA's 4M-byte window into the VME address space. You can also set the VME address modifier bits.

THE BIA DRIVER

The BIA driver must initialize the BIA after a power up. The BIA driver then accepts requests from an application program and returns appropriate data.

The BIA driver must also handle PBX interrupts. A PBX interrupt occurs when the BIA interrupts the I/O node. The BIA accepts interrupts from the VME device and then interrupts the I/O node.

To aid you in writing a BIA driver, this manual contains a simple driver for an A/D device. The intent is that you can use this example as a template to write drivers for more complicated VME devices.

INSTALLATION 2

INTRODUCTION

This chapter describes how to install and configure the BIA. Hardware configuration consists of choosing the BIA jumper positions and ensuring that an I/O node is in an adjacent even slot. Installation consists of plugging the BIA board into an iPSC slot and then plugging in the VME controller board. Software configuration consists of initialization performed by your BIA driver on the companion I/O node.

HARDWARE CONFIGURATION

A BIA board must have a companion I/O node. A BIA board resides in an odd slot; its companion I/O node resides in the even slot to the right.

Figure 2-1 shows the location of the BIA jumpers. Table 2-1 lists all the possible jumper positions.

Table 2-1. Jumper Description for the BIA Board

Jumper	Description
J1	BIA board select jumpers. The I/O node supports one BIA. Both board select jumpers should be IN. J1 should have a jumper from 1 to 2 and from 3 to 4.
J2	Timeout jumper. If the BIA expects a response from its VME controller, it will wait a certain amount of time before assuming that the VME controller is unresponsive. This jumper determines that time. The jumper should be in one of three possible positions. 1 to 2 Timeout is 10 microseconds. This is the default position. 3 to 4 Timeout is 50 microseconds. 5 to 6 Timeout is 100 microseconds.

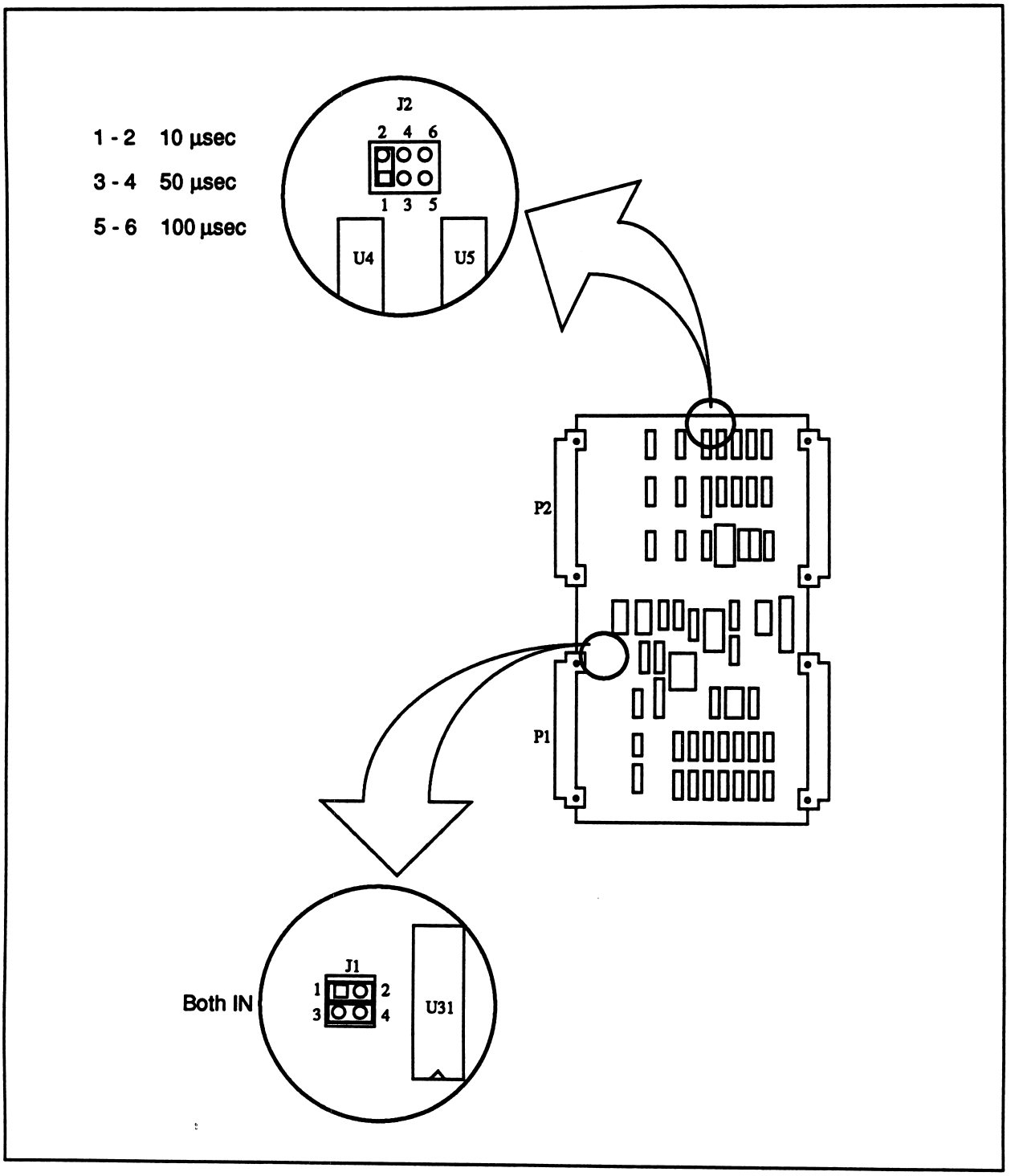


Figure 2-1. Location of BIA Jumpers

HARDWARE INSTALLATION

1. Move the lever at the bottom of the standard cabinet counterclockwise 90°. Open the cabinet's front door.
2. Turn off the power switch in the back of the cabinet.
3. Consult the documentation for the VME controller to determine any special installation instructions.
4. Plug the VME controller into the BIA board.
5. Insert the BIA board into the tracks of the empty odd slot. Be sure that the board is firmly seated. Screw in the board's front panel.
6. Apply power.

HARDWARE REMOVAL

1. Turn off power to the cabinet. The power switch is in the back.
2. Open the front door of the cabinet.
3. Unscrew the VME controller and remove it.
4. The BIA board may not have come out with the VME board. If it doesn't remove the adjacent node boards, reach into the card cage and remove the BIA board.

SOFTWARE CONFIGURATION

All BIA registers except the BIA Control Register power up in an indeterminate state. All bits in the BIA Control Register become 0 when the I/O node is reset — that is, when the signal **RESET*** goes low (true).

The BIA driver should ensure that everything is initialized correctly before the BIA is allowed to control the VMEbus. Chapter 4, "Programmer's Reference," describes how to set bits in the BIA registers.

The BIA driver should perform the following actions during initialization:

- Write the Clear Latched Interrupt Status Register.
- The signals **SYSFAIL***, **RESET***, and **VME 3STATE*** are initially low (true). The slave VME controller board cannot do anything until you set them false. To do this, set these bits in the BIA Control Register to 1 (false). The VME controller board then runs a power-up self-test.

- Poll the bits **SYSFAIL***, **RESET***, and **VME 3STATE*** in the BIA Control Register until they become 1. Note that even though you already set **SYSFAIL*** to 1, it appears as 0 until the slave passes its power-up self-test.
- Set other bits as required by the particular VME controller.
- Set the address modifier and address page bits as required.
- Write the desired interrupt mask to the BIA Interrupt Mask Register.

Set Bit 15, the master interrupt enable bit (**INTREN**), in the BIA Control Register to 1 (true).

HARDWARE REFERENCE 3

INTRODUCTION

This chapter describes the PBX bus and defines all the VME signals used by the BIA board. Then, the chapter describes the protocol and timing that governs the communication between the I/O node and the BIA.

The chapter also describes how the BIA handles interrupts, both interrupts generated by the VME controller and the interrupt generated by the BIA itself.

The chapter also provides any other hardware information that might aid you in configuring and using the BIA hardware.

SIGNAL DEFINITIONS

Figure 3-1 shows the signals used in the PBX/VME interface. Note that the complete set of VME signals is not supported. Refer to the *VMEbus Specification Manual* published by Motorola, Inc. for a description of the VME signals.

The VME user-defined LED signals control the light-emitting diodes (LEDs) on the I/O node. You can turn on the front door LEDs either by asserting these lines or by setting bits in the BIA Control Register.

The VME pins that control the LEDs are as follows:

P2 A4 When low, turns on the red LED.

P2 A3 When low, turns on the yellow LED.

P2 A2 When low, turns on the green LED.

These signals are ORed with the three bits in the BIA Control Register. Hence, if either the control bit is set or the P2 pin is active, the LED bit is set on the front door of the cabinet. The USM board can also read the status of the LED signals.

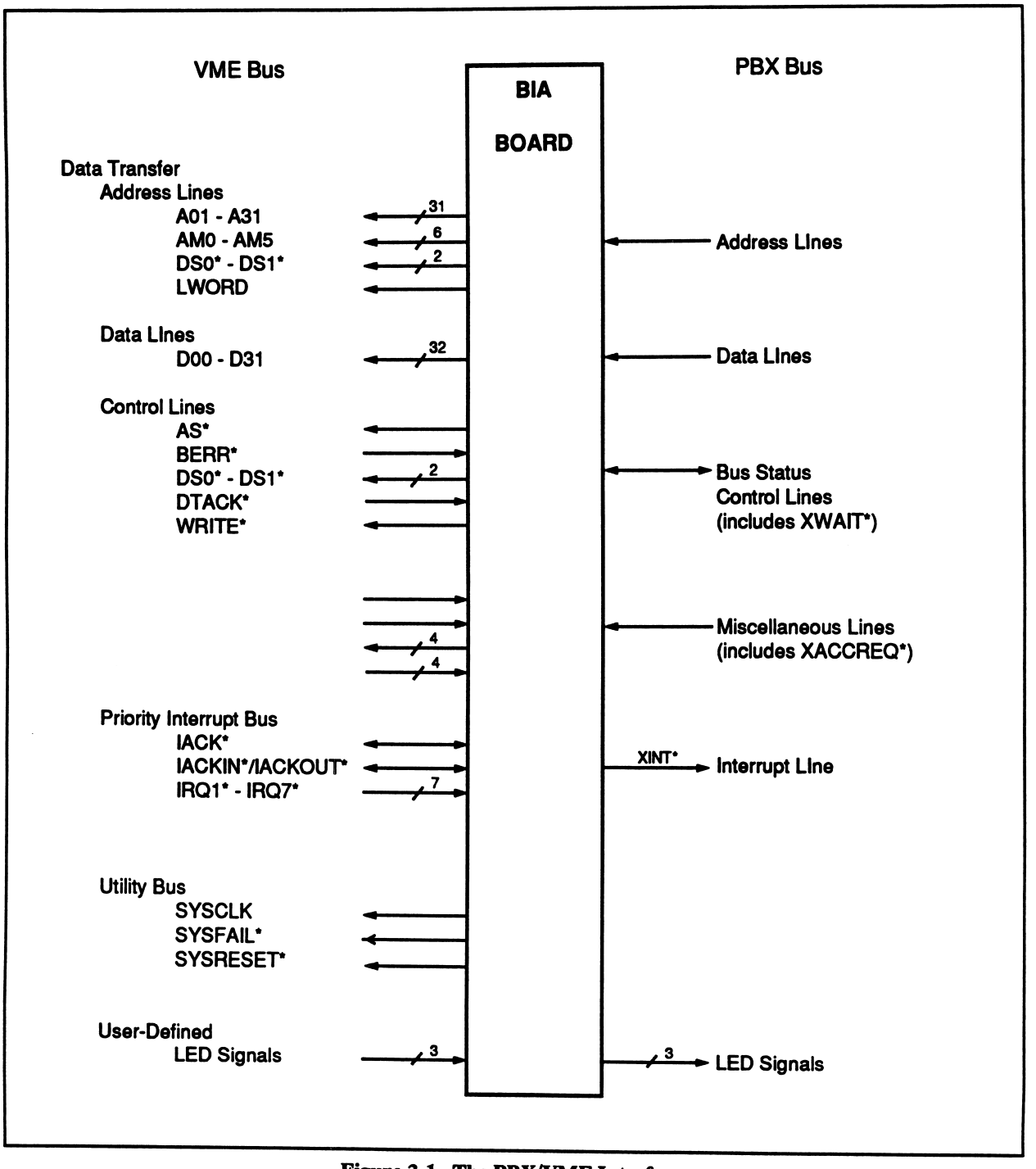


Figure 3-1. The PBX/VME Interface

BIA ACCESS

The PBX protocol governs the communication between the I/O node and the BIA. This is an asynchronous protocol that supports two types of access: a memory write and a memory read. Figure 3-2 is a timing diagram for a read operation, and Figure 3-3 is a timing diagram for a write operation.

At the beginning of a PBX access, the I/O node asserts the command signal **W/R***. This command line is asserted high for a write and low for a read. Then, the I/O node puts out the address on the address lines, **ADDR**. If the access is a write to the BIA, the I/O node also puts out the data on the data lines, **DATA**. Finally, the I/O node asserts **XACCREQ***. This line is asserted low.

The BIA samples **XACCREQ*** and performs the read or write operation when **XACCREQ*** goes low. The BIA then deasserts **XWAIT*** (raises it) to indicate that it has completed the operation. The I/O node board samples **XWAIT*** as soon as **XACCREQ*** is asserted. The BIA asserts **XWAIT*** except when it is ready to terminate the cycle.

The BIA must wait for **XACCREQ*** to be deasserted and then asserted again before initiating the next cycle.

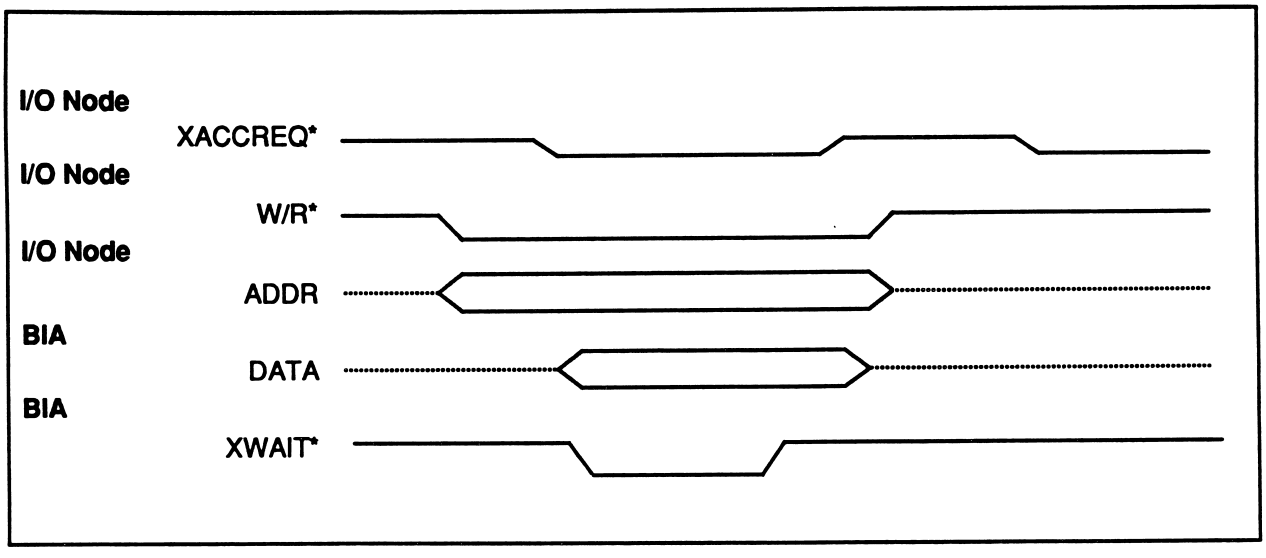


Figure 3-2. Memory Read

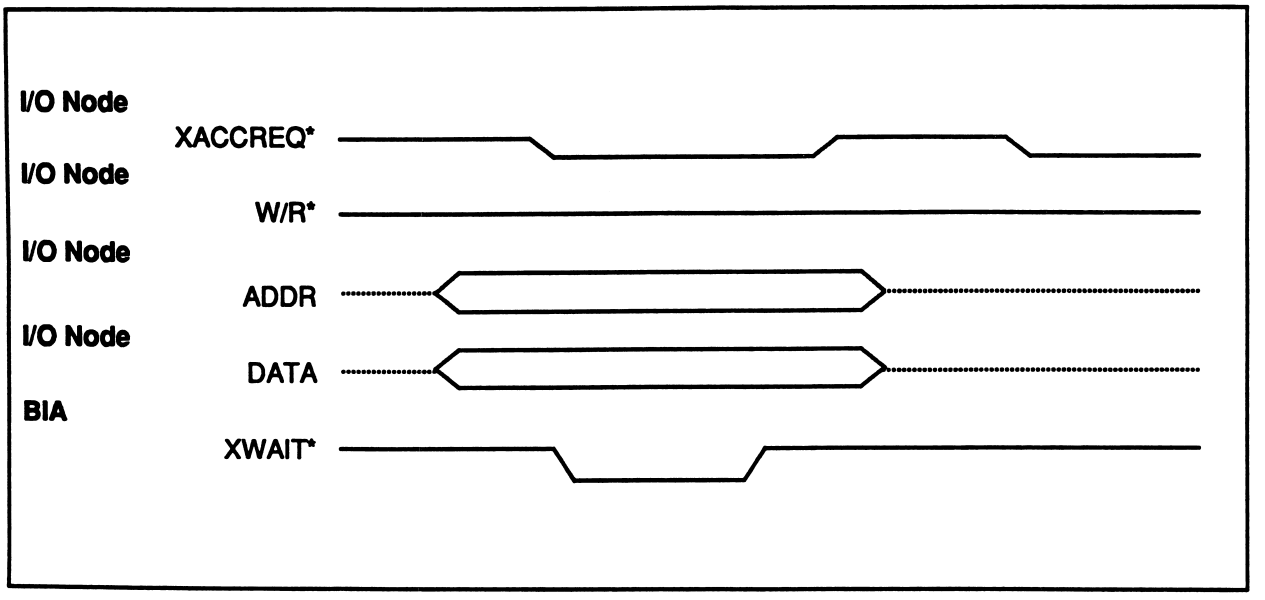


Figure 3-3. Memory Write

INTERRUPTS

The BIA produces one interrupt on its own. If the VME device does not respond within a certain time, the BIA generates a timeout interrupt. This appears to the I/O node as a **VME BERR*** interrupt with the **BIA BUS TIMEOUT*** bit set.

The BIA also accepts the following interrupts from the VMEbus:

VME SYSFAIL*

VME BERR*

VME IRQ1 through **VME IRQ7**

The BIA then interrupts the I/O node by asserting the PBX line **XINT***. The BIA driver running on the I/O node must read the Interrupt Status Register to determine the source of the interrupt.

The PBX protocol does not support specific interrupt acknowledge cycles. The I/O node acknowledges an interrupt by setting the **BIAIACK** bit in the BIA Control Register to 1 (true). If the interrupt is one of **IRQ1** through **IRQ2**, the I/O node puts an interrupt acknowledge code on the address bus and reads the interrupt vector. The interrupt acknowledge code for **IRQ1** is 0x0000 0001, that for **IRQ2** is 0x0000 0002, etc. What you do with the interrupt vector depends on the particular VME device. Then, clear the **BIAIACK** bit.

The signal **XINT*** remains asserted until the BIA driver can clear the interrupt. How the BIA driver clears the interrupt depends on the VME device. In some cases, setting the **BIAIACK** bit and performing the read may be sufficient. In other cases, you must also write to a register on the VME device, accessing it as a VME address.

Setting the **SYSFAIL*** bit to 0 also causes a **SYSFAIL*** interrupt. The BIA asserts the PBX line **XINT***.

When VME interrupts propagate through the BIA to the I/O node, they are delayed a maximum of 40 nanoseconds.

BIA REGISTERS

The BIA registers are as follows:

BIA Control Register
BIA ID Register
Clear Latched Interrupt Status
Interrupt Mask Register
Interrupt Status Register
Loopback Test Register
VME Address Register

All BIA registers except the BIA Control Register power up in an indeterminate state. The BIA Control Register comes up as all zeros. Some of the registers are write/read, meaning that you can read back what you have written. Other registers are write-only or read-only.

BIA Control Register

Address	0x00c0 0004
Width	16 bits
Access	Write-only

The BIA Control Register controls all the different modes that the BIA can be set to. All bits in this register are set to 0 after a reset.

Table 3-1. Bit Pattern for the BIA Control Register

Bit	Name	Description
D0	LEDCTRL0*	When 0 (true), turns on the green LED on the front panel.
D1	LEDCTRL1*	When 0 (true), turns on the red LED on the front panel.
D2	LEDCTRL2*	When 0 (true), turns on the yellow LED on the front panel. These LEDs are on after power-up and are usually turned off by the BIA driver during initialization.
D3	SHUFFL0	The least significant bit of the VME memory mode. See the section, "Byte Swapping," for more information.
D4-D7	Reserved	
D8	SYSFAIL*	When 0 (true), asserts the VME SYSFAIL* signal, which is shown as the VME SYSFAIL* bit in the Interrupt Status Register. Do not access the VME device when this bit is 0.
D9	RESET*	When 0 (true), asserts the VME board's reset line. The line stays asserted until you set the bit to 1. Do not access the VME device when this bit is 0.
D10	VME 3STATE*	When 0 (true), puts the BIA data and address lines into a high impedance state. Do not access the VME device when this bit is 0.
D11	SHUFFL1	The most significant bit of the VME memory mode. See the section, "Byte Swapping," for more information.
D12	TESTNTWRK	When 1 (true), allows data from the byte swapping network test latches to be read in.
D13	BLOCKMODE	When 1 (true), the BIA will perform block mode transfers.
D14	BIAIACK	Set this bit high to do a VME interrupt acknowledge. See the section, "Interrupt Handling," for more information.
D15	INTREN	This is the master interrupt enable bit. The BIA driver should set this bit to 1 after initialization.

BIA ID Register

Address 0x00c0 0000
Width 32 bits
Access Read-only

The BIA revision number is actually stored in the Interrupt Status Register. Because this information does not logically fit with interrupt status, it's useful to think of these bits as belonging to a different register.

Table 3-2. Bit Pattern for the BIA ID Register

Bit	Name	Description
D0-D15		Interrupt Status Register bits.
D16	**BIAREV0**	The least significant bit of the BIA revision number.
D17	**BIAREV1**	BIA revision bit.
D18	**BIAREV2**	BIA revision bit.
D19	**BIAREV3**	BIA revision bit.
D20	**BIAREV4**	BIA revision bit.
D21	**BIAREV5**	BIA revision bit.
D22	**BIAREV6**	BIA revision bit.
D23	**BIAREV7**	The most significant bit of the BIA revision number
D24-D31	Reserved	

Clear Latched Interrupt Status Register

Address 0x00c0 0000
Width No data are actually written.
Access Write-only

Note that this register has the same address as the Interrupt Status Register. However, the Interrupt Status Register is a read-only register. If you write anything to this address, you clear the latched **BIA BUS TIMEOUT*** and **VME BERR*** bits. This sets the bits **BIA BUS TIMEOUT*** and **VME BERR*** in the Interrupt Status Register to 1 (false).

Interrupt Mask Register

Address 0x00c0 0008
Width 16 bits
Access Write/read

The Interrupt Mask Register enables individual interrupts. All bits are in an indeterminate state after a power up or reset. Ensure that these bits are set correctly before setting **INTREN**, the master interrupt enable bit, in the BIA Control Register.

Table 3-3. Bit Pattern for the Interrupt Mask Register

Bit	Name	Description
D0	**VME SYSFAIL***	When 0 (true), enables the interrupt due to a VME catastrophic failure.
D1	**BIA BUS TIMEOUT***	When 0 (true), enables the interrupt that occurs when the VME device did not respond within the allotted time set by jumpers on the BIA board.
D2	**VME BERR***	When 0 (true), enables the interrupt that occurs when the VMEbus has a bus error.
D3	**VME IRQ1***	When 0 (true), enables the VME interrupt, **VME IRQ1***.
D4	**VME IRQ2***	When 0 (true), enables the VME interrupt, **VME IRQ2***.
D5	**VME IRQ3***	When 0 (true), enables the VME interrupt, **VME IRQ3***.
D6	**VME IRQ4***	When 0 (true), enables the VME interrupt, **VME IRQ4***.
D7	**VME IRQ5***	When 0 (true), enables the VME interrupt, **VME IRQ5***.
D8	**VME IRQ6***	When 0 (true), enables the VME interrupt, **VME IRQ6***.
D9	**VME IRQ7***	When 0 (true), enables the VME interrupt, **VME IRQ7***.
D10	Reserved	
D11	Reserved	
D12	Reserved	
D13	Reserved	
D14	Reserved	
D15	Reserved	

Interrupt Status Register

Address 0x00c0 0000
Width 16 bits
Access Read-only

The BIA supports the seven VME interrupts, but presents only one interrupt to the I/O node. The I/O node must then read the Interrupt Status Register to determine the nature of the interrupt.

Table 3-4. Bit Pattern for the Interrupt Status Register

Bit	Name	Description
D0 D1 D2 D3 D4 D5 D6	**VME IRQ1*** **VME IRQ2*** **VME IRQ3*** **VME IRQ4*** **VME IRQ5*** **VME IRQ6*** **VME IRQ7***	**VME IRQ1*** through **VME IRQ7*** are the VME interrupt lines. When a VME device interrupts, the bit corresponding to that interrupt is low.
D7	Reserved	Appears as 1.
D8	**VME DTACK***	Indicates that a slave has either accepted data from or placed data onto the VME data bus. In a normal VME cycle, however, **VME DTACK*** goes high before the end of the PBX bus cycle.
D9	**VME SYSFAIL***	This bit is 0 (true) until the VME board has passed its power-up test or when a catastrophic VME failure has occurred. You can force this bit to 0 (true) by setting the **SYSFAIL*** bit in the BIA Control Register to 0 (true).
D10	**BIA BUS TIMEOUT***	When 0 (true), indicates that the VME device did not respond within the allotted time. This time is set by a jumper on the BIA. When this occurs, the **VME BERR*** bit is also 0 (true). This bit is latched.
D11	**VME BERR***	This bit is a latched version of the **VME BERR*** signal from the VMEbus. Note that if a slave does not respond within the allotted time both **BIA BUS TIMEOUT*** and **VME BERR*** go true. A slave may make **VME BERR*** true for some other reason dependent on the VME device, in which case **BIA BUS TIMEOUT*** is unaffected.
D12	**SPARE**	This bit is high because it's pulled up with a resistor. You have the option of bringing in another status bit by clipping the resistor R5 and attaching the wire to pin 11 of U49.
D13-D15	Reserved	

Loopback Test Register

Address	0x00c0 0004
Width	32 bits
Access	Read-only

Note that this register has the same address as the BIA Control Register. However, the Loopback Test Register is read-only while the BIA Control Register is write-only. Also, the Loopback Test Register is a 32-bit register while the BIA Control Register is 16 bits.

Reading the Loopback Test Register returns the data written through the byte swapping network when the BIA is in test mode. In this way you can check to see whether your data are swapped correctly as well as verify that the data path to the VMEbus is functional.

Use the Loopback Test Register as follows:

1. Put the BIA in test mode. Set the bit **TESTNTWRK** in the BIA Control Register to 1 (true).
2. Write your data to the VME Address Register (0x00c0 000c).
3. Read the Loopback Test Register.

When the BIA is in test mode, do not read any other BIA registers and do not write to VME memory. To remove the BIA from test mode, set the **TESTNTWRK** bit in the BIA Control Register to 0 (false).

Note that any previous data in the VME Address Register are overwritten.

VME Address Register

Address	0x00c0 000c
Width	32 bits
Access	Write/read

The VME Address Register contains the bits that let you set the start of the 4M-byte VME window. The register also contains the VME address modifier bits. These bits determine the type of VME access. Refer to *The VMEbus Specification* for more information about the VME address modifier bits.

Note that although all 32 bits of this register are not used, you must still write a 32-bit word when you access it. The extra bits have don't-care values. The start of the memory window is in bits D22 through D31. The address modifier bits are in bits D0 through D5.

Table 3-5. Bit Pattern for the VME Address Register

Bit	Name	Description
D0	VME ADDR MOD0	Address modifier bit 0
D1	VME ADDR MOD1	Address modifier bit 1
D2	VME ADDR MOD2	Address modifier bit 2
D3	VME ADDR MOD3	Address modifier bit 3
D4	VME ADDR MOD4	Address modifier bit 4
D5	VME ADDR MOD5	Address modifier bit 5
D6-D21	Reserved	Write these bits as don't-care.
D22	VME A22	VME address bit 22. The least significant bit for the BIA address window.
D23	VME A22	VME address bit 23.
D24	VME A24	VME address bit 24.
D25	VME A25	VME address bit 25.
D26	VME A26	VME address bit 26.
D27	VME A27	VME address bit 27.
D28	VME A28	VME address bit 28.
D29	VME A29	VME address bit 29.
D30	VME A30	VME address bit 30.
D31	VME A31	VME address bit 31. The most significant bit for the BIA address window.

BIA SPECIFICATIONS

Physical

BIA Form Factor	11.18 x 23.33 cm (4.4 x 9.187 inches)
VME Form Factor	Accepts standard 6U VME Eurocard boards. 233 x 160 mm. (9.187 x 6.3 inches)
Number of Required Slots	One per I/O node, each I/O node on a separate bus.
Card Cage	Requires 19-inch iPSC card cage and standard cabinet.

Bus Specifications

Data Transfer Types	8, 16, 24, and 32-bit accesses; software controlled byte swapping hardware provided for format conversion during transfers.
Address Types	16, 24, and 32-bit
Address Space	Moveable 4M-byte directly addressable VME address window provides access to all 32 bits of VME address space via the VME Address Register.
Operational Modes	Supports slave mode; I/O node is the VME master. Supports VME block mode transfers. Supports unaligned transfers (UAT).
Interrupts	Interrupt lines **IRQ1** through **IRQ7** available; software maskable. **SYSFAIL***, **BERR***, VME timeout.
VME Cycles Supported	Standard; extended; short I/O; supervisory/nonprivileged; block interrupt acknowledge. Address modifiers can be set to any value via the VME Address Register.
VME Bus Signals	**SYSCLK** 16 MHz system clock **SYSRESET*** system reset **SYSFAIL*** system fail **BERR*** bus error **ACFAIL*** is terminated according to the VMEbus specification but is not readable or writable. For a complete list of VME signals, refer to Figure 3-1.

Electrical

Power available to VME board	DC power +5V at 10 amps, +12V at 1 amp and -12V at 1 amp provided to VME device
BIA Power Consumption	+5V at 4 amps

Environmental

Temperature	10 to 35°C (50 to 95°F)
Humidity	10-90% relative humidity, noncondensing
Safety and Emissions Standards	Designed to meet UL478, CSA C22.2 No. 154, VDE 0806, VDE 0871 Class A, IEC 380, and FCC 15 CFRJ Class A when mounted in the specified cabinet.

Configuration

- The VME board may have some of its own cabling. Consider placing the BIA in the lower card cage if its cabling unduly restricts access to the upper slots.
- The iPSC system provides a great deal of latitude for VME cable placement. If you have difficulty routing your VME cables, call SSD Customer Support.

1-800-421-2823 (Customer Support Hotline)
(44) 793 641 469 (in England)
Your Local Intel Sales Office (in Europe)
support@isc.intel.com (Internet address)

PROGRAMMER'S REFERENCE 4

INTRODUCTION

This chapter provides the software background necessary to design and write a BIA driver. The chapter describes how to access the BIA registers and VME memory. The chapter also describes the BIA's memory organization, describes how to compile and load the BIA driver, and gives an overview of how the BIA driver should handle PBX interrupts and communicate with an application program on another node.

WRITING THE BIA DRIVER

The BIA driver is written in C and runs on the BIA's companion I/O node. After powerup, it initializes the BIA and then waits for a message from an application program. When it receives a message, it performs the requested operation and returns any requested data to the application program in the form of a message.

The BIA driver controls the BIA by reading and writing BIA registers. It accesses the VME controller board by reading and writing VME memory addresses. The I/O node's address space is divided into node memory, BIA registers, and VME memory.

The BIA driver must also handle interrupts from the BIA. These are called PBX interrupts. When the VME controller board generates an interrupt, the BIA in turn interrupts the I/O node. The I/O node sees that interrupt as an exception. As part of its initialization, the BIA driver attaches a procedure to that exception, and that procedure handles the interrupt.

In summary, BIA driver operations consist of reading and writing particular memory locations, message passing, and handling PBX interrupts.

Although the BIA driver is written in C, it must use two assembly routines. During initialization, the BIA driver must enable PBX interrupts by setting a bit in the I/O node's Programmable Interrupt Controller. The two assembly routines **in()** and **out()** (described in the next section and shown as part of the example driver in Appendix A) provide that capability.

ASSEMBLY ROUTINES

The BIA driver must use two assembly routines, **in()** and **out()**. These routines read from and write to I/O ports on the I/O node. Include them in the same file. Then, assemble the file and link it with your C program.

For example, assume that the two assembly routines are in the file *port.s*. To assemble the file and call the resulting object file *port.o*, issue the command:

```
as -o port.o port.s
```

Then, include the file *port.o* on a cc compilation line along with the rest of the modules making up your BIA driver. For example, the appropriate cc line in your makefile might look as follows:

```
biadriver:port.o biadriver.o
   cc -o biadriver port.o biadriver.o -node
```

Note that the **in()** and **out()** routines are written in AT&T assembler, which is slightly different from Intel assembler. You do, however, need an Intel assembly instruction not present in the AT&T assembler. To get this instruction, include the actual opcode with the **.byte** pseudo operative.

Here's the code for **in()** and **out()**.

```
/  Calling Sequence
/        in(port);
/
/  Description:
/        Input a byte from an I/O port.
/
/  Parameters:
/        port:    the I/O port address
/
/  Returns:
/        The byte value of the port
/
.globl   in
in:
   push %ebp
   mov   %esp, %ebp
   mov   8(%ebp),%edx
   mov $0, %eax
   .byte 0xEC      / opcode for inb %dx
   pop %ebp
   ret
```

```
/  Calling Sequence
/        out(port,value);
/
/  Description:
/        Output a byte to an I/O port.
/
/  Parameters:
/        port:    the I/O port address
/        value:   the value to output
/
/  Returns:
/        none
/
.globl   out
out:
   push  %ebp
   mov   %esp, %ebp
   mov   8(%ebp),%edx
   mobv  12(%ebp), %al
   .byte 0xEE      / opcode for outb %dx
   pop %ebp
   ret
```

ACCESSING THE BIA REGISTERS

Each BIA register has an address in the I/O node's address space. Before accessing any BIA register, your program must turn on PBX memory mapping. Then, to access a BIA register, you just read from or write to the appropriate address.

Turning on Memory Mapping

To turn on memory mapping, write 0x28 to the I/O node's port 0x80. Call the assembly routine **out()** from within your C program. For example,

```
out(0x80, 0x28);
```

Converting from Physical to Logical

Note, however, that your BIA driver runs as an application on the I/O node. The addresses in your program are user addresses (logical addresses) and not true physical addresses. The memory-mapped I/O addresses listed in this manual for the BIA registers are physical addresses. To convert that physical address into the corresponding logical address accessible by your program, you must add the constant 0xc000 0000.

Example: Writing the BIA Control Register

Here is a code fragment that sets the Interrupt Enable bit in the BIA Control Register. The BIA registers begin at BIA_REG (0x00c0 0000). The physical address of the BIA Control Register is BIA_REG + CONTROL (0x00c0 0004). Finally, you must add the constant 0xc0000000 to get the logical address. To write the BIA Control Register from your BIA driver, you must actually write the address 0xc0c0 0004.

Note that because the BIA Control Register is write-only, a common practice is to store its value in a variable and then write that variable to the register. If you do this consistently, then you can always read the variable to find out the value of the register.

```
#define CONTROL 0x0004
#define BIA_REG 0x00c00000
#define INT_ENBL 0x0800
```

```
unsigned short bia_ctr_reg, *bia_ctrl;
     .
     .
     .
/*Get logical address of the BIA Control Register */
bia_ctrl = (unsigned short *)(BIA_REG + CONTROL + 0xc0000000);

bia_ctrl_reg |= INT_ENBL;
*bia_ctrl = bia_ctrl_reg;
```

When you access, write/read registers, such as the VME Address Register, take care to read and write the register as the same data type. For example, the VME Address Register is 32 bits long and written as an **unsigned long**. Hence, you should also read it as an **unsigned long**.

ACCESSING VME MEMORY

The VMEbus provides a 4G-byte address space. VME addresses range from 0x0000 0000 to 0xffff ffff. The BIA has a 4M-byte window into this address space.

Hence, you can think of the VME address as consisting of a PAGE and a PAGE_OFFSET. Bits A31 through A22 represent the PAGE, and bits A21 through A0 represent the PAGE_OFFSET. Figure 4-1 illustrates this representation.

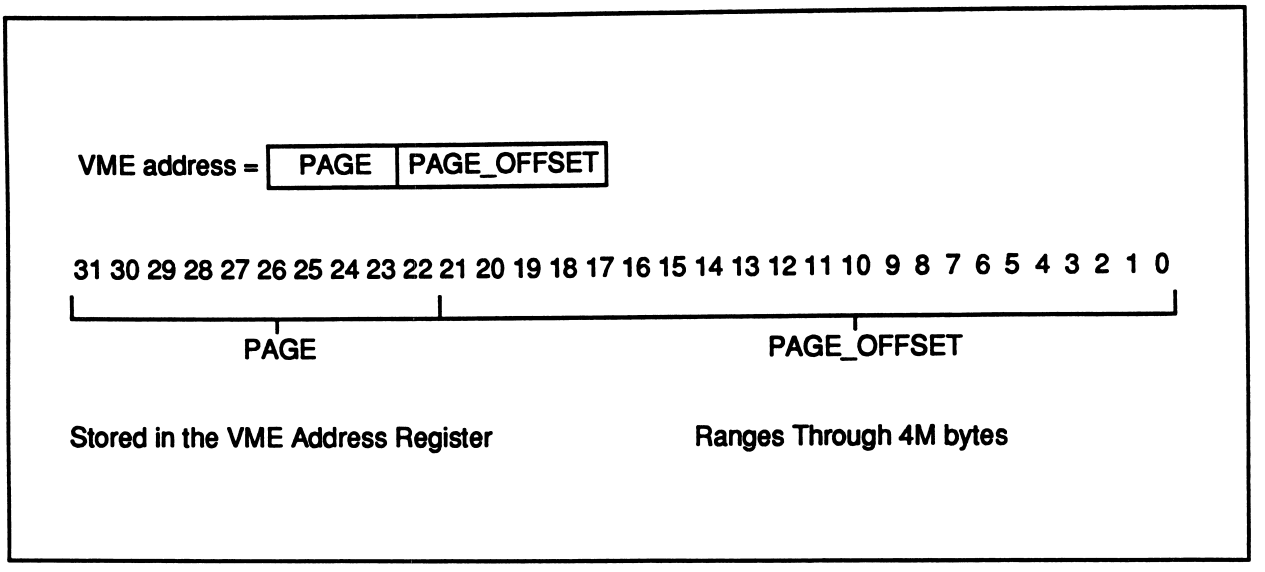


Figure 4-1. The VME Address

The PAGE is stored in the VME Address Register. The PAGE_OFFSET ranges through 4M bytes, from 0x0000 0000 to 0x003f ffff.

The I/O node's address space has its lower 8M bytes reserved for node memory. The next 4M bytes make up the VME window. The 4M bytes beyond those are reserved for BIA register access. These addresses are physical addresses.

When you actually read or write the addresses, you must use the corresponding logical addresses. To get the logical address that corresponds to a physical address, add the constant 0xc000 0000. Figure 4-2 illustrates the relationship between user (logical) and physical addresses.

Be careful not to access an address beyond those reserved for BIA register access. That is, do not access an address greater than 0x00ff ffff. This may cause the PBX node to hang.

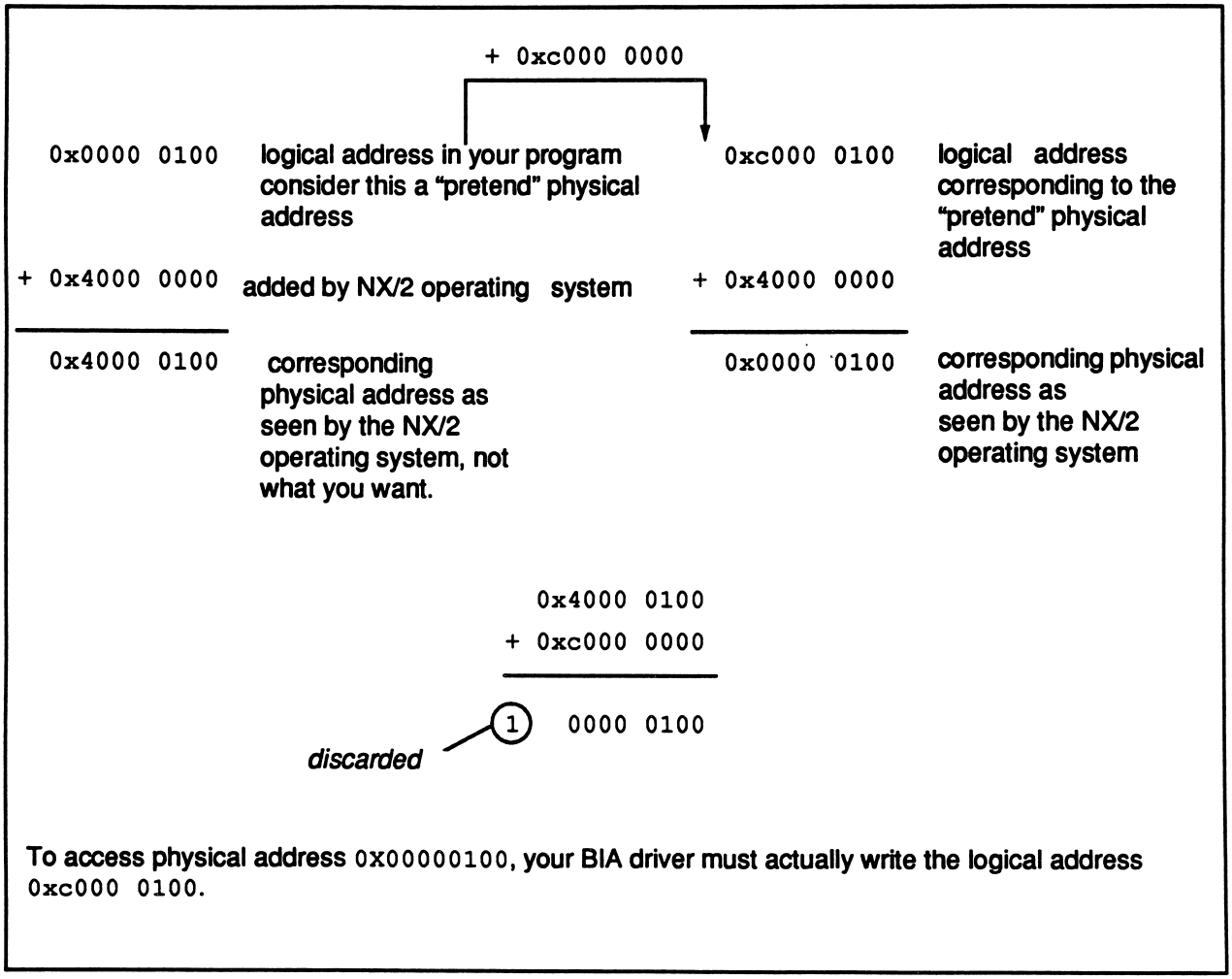


Figure 4-2. Accessing a Physical Address

Example: Reading a VME Address

Here is an example that shows how to read the VME address 0xf000 0100. Because the VME Address Register is a 32-bit register, the VME memory mode should be 32-Bit Mode. The VME memory modes are described later in this chapter.

First, you must ensure that the upper ten bits of the VME Address Register contain 1111 0000 00, the value of PAGE. Then, you must mask out PAGE from the VME address to get the PAGE-OFFSET. Next, add 8M bytes to move the PAGE_OFFSET into the I/O node's address space. Finally, add 0xc000 0000 to get the corresponding logical address.

Here's a C code fragment that does that.

```
#define BIA_VME 0x00800000
#define MOD_MASK 0x0000003f
#define PAGE_MASK 0xffc00000
unsigned long addr_val, *bia_addr_reg;
unsigned char *addr, byte_val;
 .
 .
 .
/* VME Address to be read */
 addr_val = 0xf0000100;
/* Put in PAGE bits into VME Address Register */
 *bia_addr_reg &= MOD_MASK; /* Keep addr modifier bits */
 *bia_addr_reg |= addr_val & PAGE_MASK; /* OR in the PAGE */

/* Mask out the PAGE to get PAGE_OFFSET */
 addr_val &= ~PAGE_MASK;
/* Add the I/O space offset to get into VME range */
 addr_val += BIA_VME;

/*Get the logical address that corresponds to the physical you
want */
 addr = (unsigned char *)(addr_val + 0xc0000000);

/* Read the address */
 byte_val = *addr;
 .
 .
 .
```

Figure 4-3 illustrates the operation of the example.

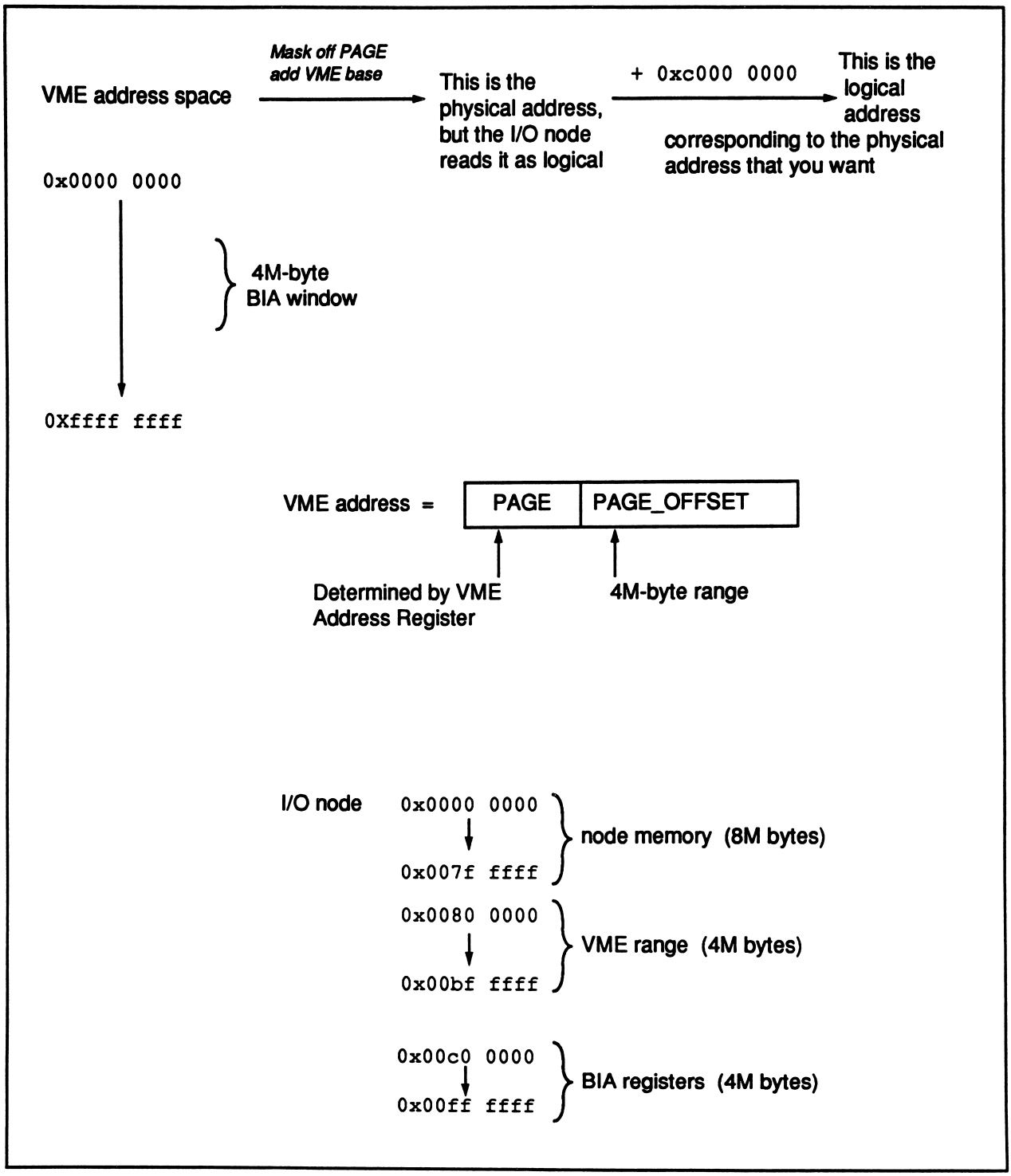


Figure 4-3. Reading and Writing VME Addresses

Block Mode

The BIA enters block mode when the bit **BLOCKMODE** in the BIA Control Register is set to 1.

When the BIA is in block mode, the address incrementing happens automatically. For example, assume that you are reading 256 bytes from the VMEbus into a byte array called *byte_array[]*. The starting VME address is stored in *addr*. When the BIA is in block mode, you can perform the read as follows:

```
int i;
unsigned char byte_array[256], *addr;
 .
 .
 .
 for(i=0;i<256;i++)
  byte_array[i] = *addr;
```

When the BIA is *not* in block mode, you must increment the address yourself.

```
int i;
unsigned char byte_array[256], *addr;
 .
 .
 .
 for(i=0;i<256;i++) {
  byte_array[i] = *addr;
  addr++;
 }
```

When in block mode, you must be careful not to read beyond a 256-byte boundary in memory. A 256-byte boundary has the two lower hex digits of its address equal to 0. For example,

0x0000 0000 256-byte boundary
0x0000 00ff

0x0000 0100 The next 256-byte boundary
0x0000 01ff

0x0000 0200 The next 256-byte boundary
0x0000 02ff

If you want to read beyond a 256-byte boundary, you must specify a new address just before hitting the boundary. To specify a new address, just reassign *addr*.

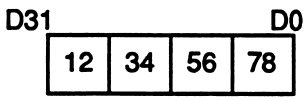
For example, if you want to read 256 long words into a **long** array, if you begin at a 256-byte boundary, and if block mode is in effect, your code might look as follows:

```
int i,j;
unsigned long long_array [256], *addr, *start_addr;
 .
 .
 .
 for(j=0;j<4;j++) {
  addr = start_addr;
  for(i=0;i<64;i++)
   byte_array[i + j*64] = *addr;
  start_addr += 64;
 }
```

The address *start_addr* is a 256-byte boundary.

The Byte Swapping Network

The Intel byte ordering convention specifies that the least significant byte of a number is stored at the lowest memory address. For example, following the Intel convention, the 32-bit long word 0x1234 5678 is written on the data bus as follows:



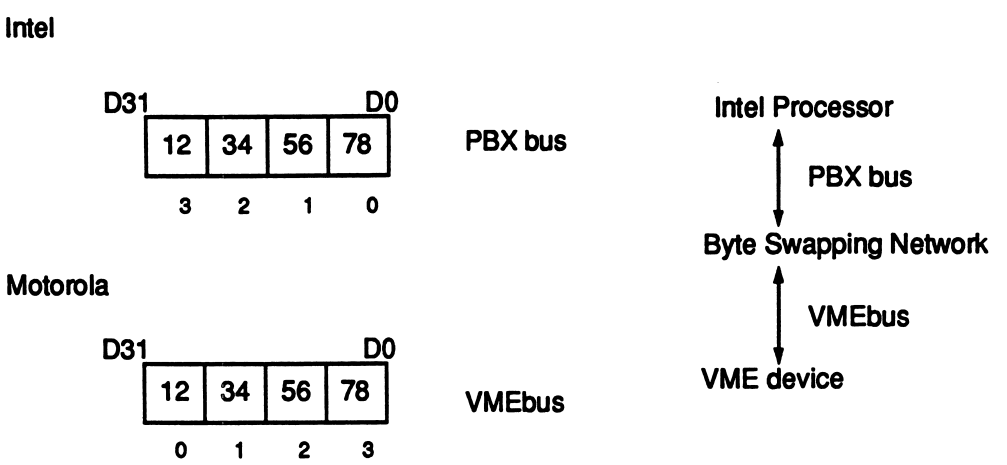
and stored in memory as follows:

start address+0	78	Intel convention
start address+1	56	
start address+2	34	
start address+3	12	

Motorola on the other hand writes the 32-bit word on the data bus in the same way but stores the bytes in memory as follows:

start address+0	12	Motorola convention
start address+1	34	
start address+2	56	
start address+3	78	

Another way of looking at this is to realize that, when writing a 32-bit word, Intel numbers bytes from 0 to 3 on the PBX bus starting at the least significant byte. Motorola on the other hand numbers the bytes 3 to 0 on the VMEbus, also starting at the least significant byte.



The Intel processor transfers data on the PBX bus. The data must then go onto the VMEbus to get into VME memory. The VME device transfers data on the VMEbus. The data must then go onto the PBX bus to get to the Intel processor.

The BIA's byte swapping network allows the BIA's Intel processor to talk to the VME world. Two bits in the BIA Control Register determine how the BIA implements byte swapping. These bits are called **SHUFFL1** and **SHUFFL0**. Think of the combination as setting a mode. There's no insight to be gained by trying to attach a meaning to each individual bit.

Table 4-1 lists the four valid modes. The modes are characterized by data alignment, address translation, and byte swapping.

Table 4-1. VME Memory Modes

SHUFFL1	SHUFFL0	Description
0	0	Array Mode. Swaps bytes, but does not translate addresses. Performs data alignment.
0	1	8-Bit Mode. Use this mode for writing and reading bytes. Swap bytes but does not translate addresses. Performs data alignment
1	0	16-Bit Mode. Use this mode for writing and reading 16-bit words. Does not swap bytes and does not translate addresses. Performs data alignment
1	1	32-Bit Mode. Use this mode for writing 32-bit words. Does not swap bytes, but does translate addresses.

The rest of this section defines the terms data alignment, address translation, and byte swapping. Then, the section shows how the four modes use these concepts.

For a series of short transfers in different VME Memory Modes, you may find that you get higher transfer rates by choosing one VME Memory Mode and then shuffling the bytes in software on the I/O node rather than changing VME memory modes. For block transfers, setting the VME Memory Mode for each transfer should result in the best performance.

DATA ALIGNMENT

In aligned transfers, bytes are accessed at byte boundaries, 16-bit words at 16-bit boundaries and 32-bit words at 32-bit boundaries. For example, writing a 32-bit word to address 0 is an aligned transfer; writing it to address 1 is an unaligned transfer.

Aligned Transfers

All byte transfers and aligned 16-bit word transfers on the VMEbus occur on D15-D0. This means that any bytes or words written on D31-D16 of the PBX bus must be shifted down to D15-D0 as shown in Figure 4-4.

The VMEbus would look the same if you were writing the 16-bit word to address 0. The PBX bus, though, would have the 0x1234 in its lower half.

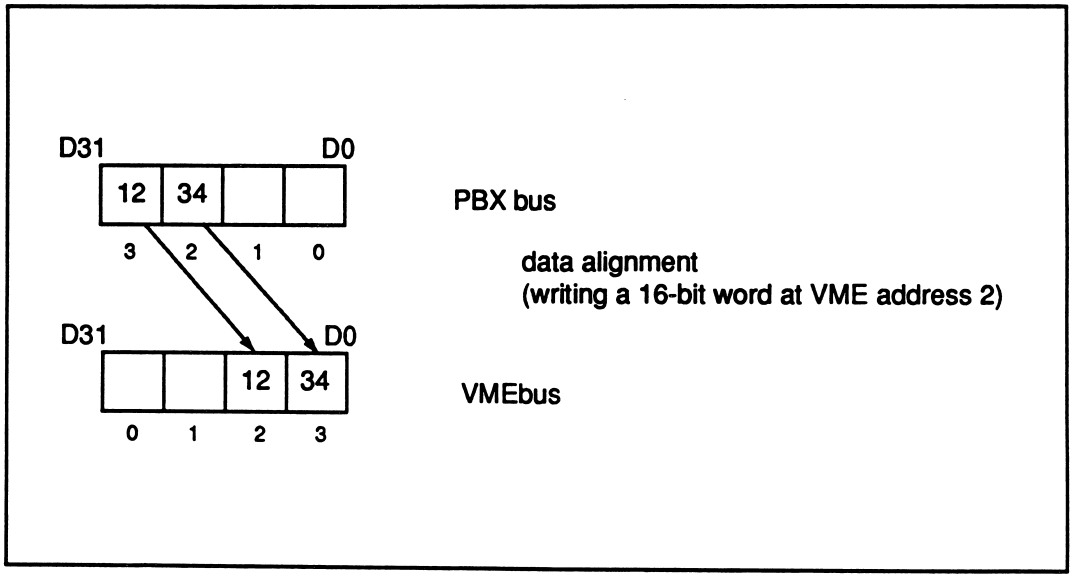


Figure 4-4. Data Alignment for a 16-Bit Word

Now consider writing bytes. For example, assume that you want to write a byte to VME address 0. The byte appears on D7-D0 of the PBX bus. If there were no data alignment, you would want the byte on D31-D24 of the VMEbus to get it in address 0. Because of data alignment, however, the proper location is D15-D8, as shown in Figure 4-5. Think of the addresses 0 1 on the VMEbus shifted over the addresses 2 3.

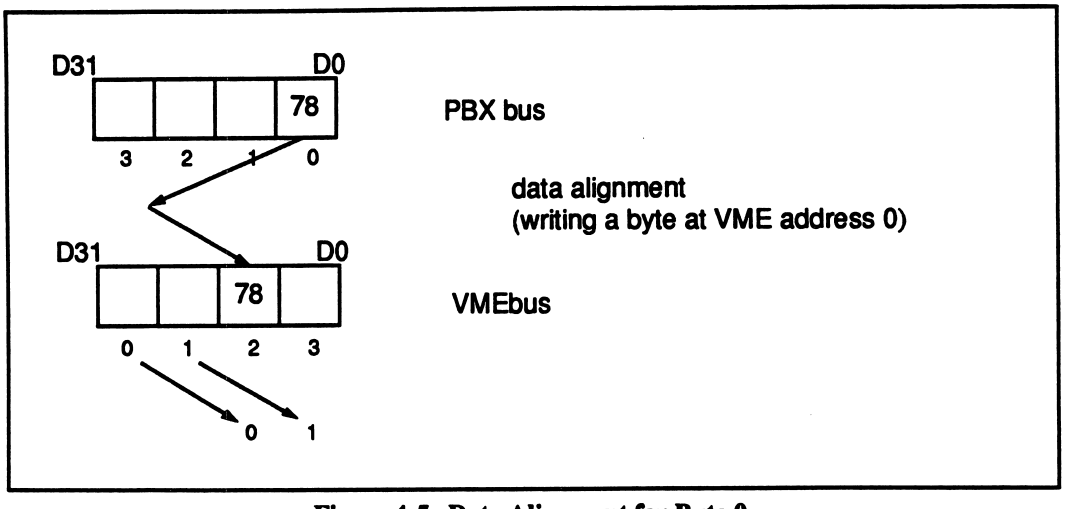


Figure 4-5. Data Alignment for Byte 0

Analogously, if you wanted to write the byte to VME address 1, the proper location would be D7-D0 on the VMEbus. This is shown in Figure 4-6.

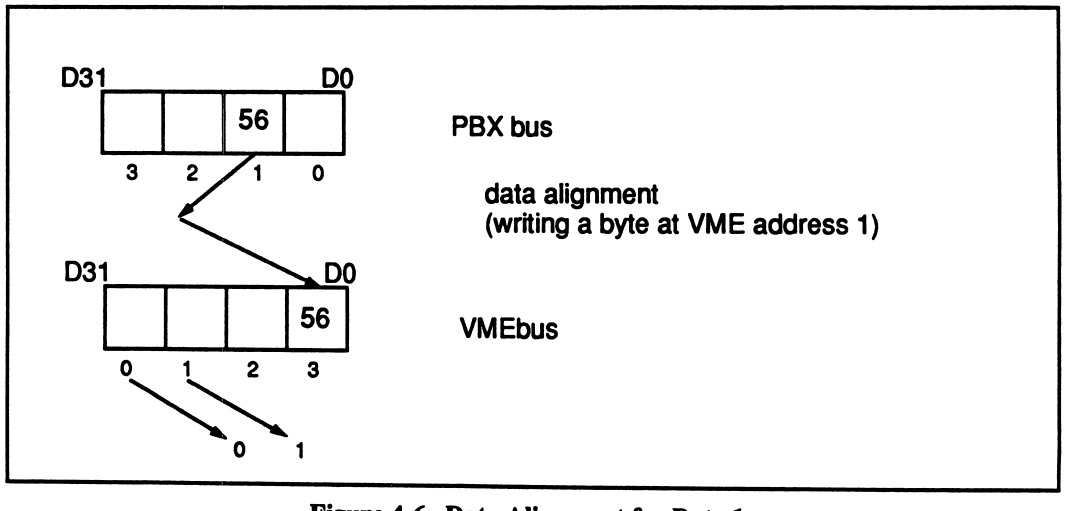


Figure 4-6. Data Alignment for Byte 1

The proper location for writing a byte to address 2 is D15-D8, and the proper location for writing a byte to address 3 is D7-D0. Refer to Figure 4-7.

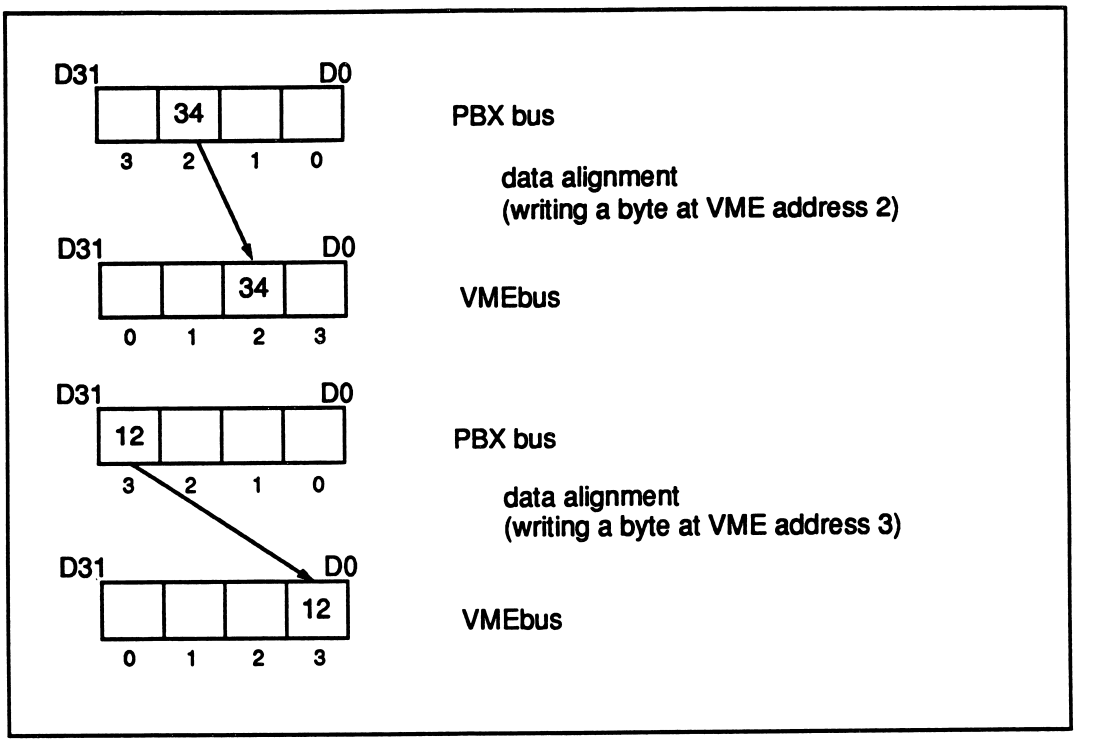


Figure 4-7. Data Alignment for Bytes 2 and 3

Unaligned Transfers

Unaligned transfers are broken into two transfers. For example, writing a 32-bit word to address 1 is done as two transfers: a 3-byte write at address 1 and a 1-byte write at address 4.

ADDRESS TRANSLATION

First of all, consider what would happen if there were no address translation, just data alignment. Assume that you wrote the 32-bit word 0x1234 5678 into VME memory at address 0. If the Intel processor wants the least significant byte of that word (0x78), it requests address 0 and expects the data on D7-D0 on its PBX bus.

Address 0, however, as far as the VME device is concerned, contains the most significant byte (0x12). Because of data alignment, this byte actually appears on D15-D8. The result is that you get the wrong data at the wrong location on the bus.

With address translation, the VME device sees address 3 when the Intel processor requests address 0. The result is that you get the right data. Figure 4-8 illustrates address translation for a byte.

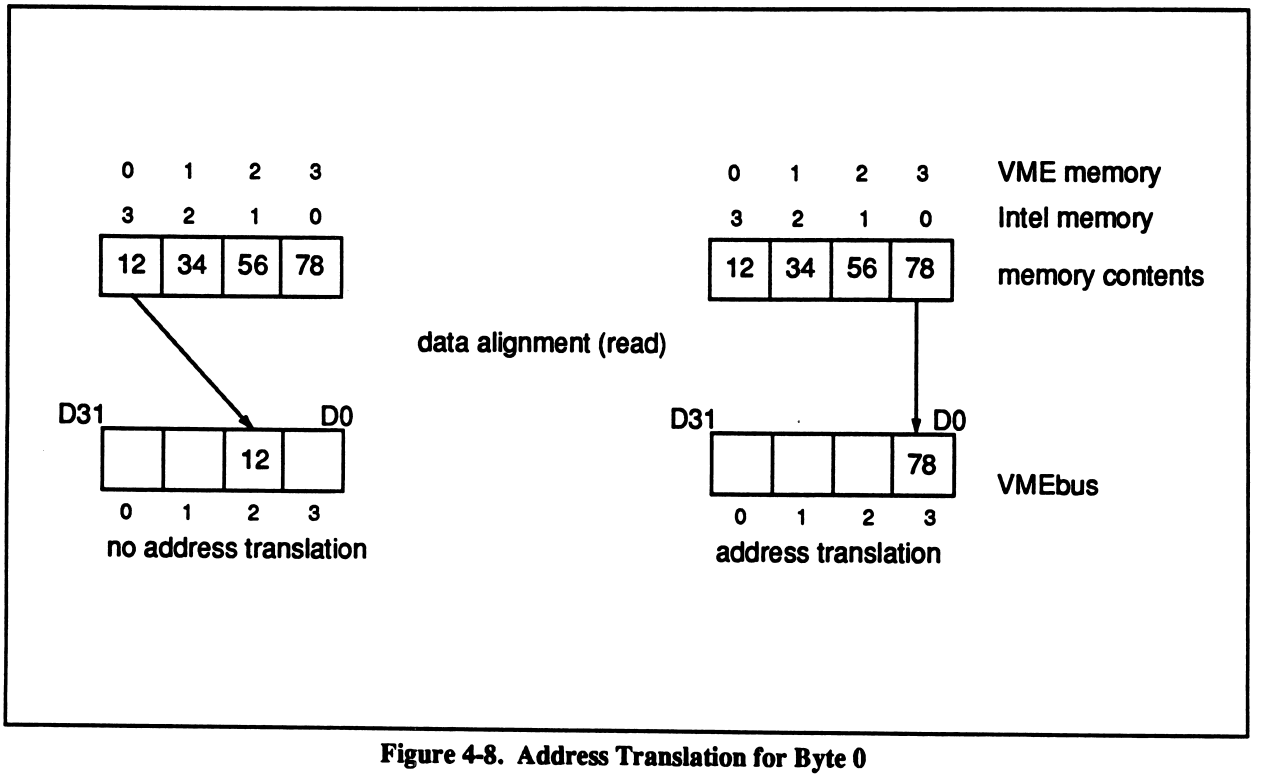


Figure 4-8. Address Translation for Byte 0

Now consider reading a word. Again assume that you have the 32-bit word 0x1234 5678 stored at VME address 0. If the Intel processor wants the word 0x1234, it requests address 2 and expects the data on D31-D16.

Address 2, however, as far as the VME device is concerned, contains 0x5678. The VME device puts this on D15-D0, as shown in the left half of Figure 4-9.

With address translation, the VME device sees address 0 when the Intel processor requests a 16-bit word at address 2. Because of data alignment, the 16-bit word actually appears on D15-D0 of the VMEbus.

Note that address translation is not enough. To get the right data in the right place, the BIA must also take into account the data alignment and push the 0x1234 up to D31-D16 on the PBX bus. The right half of Figure 4-9 illustrates address translation and data alignment when reading a 16-bit word at address 2.

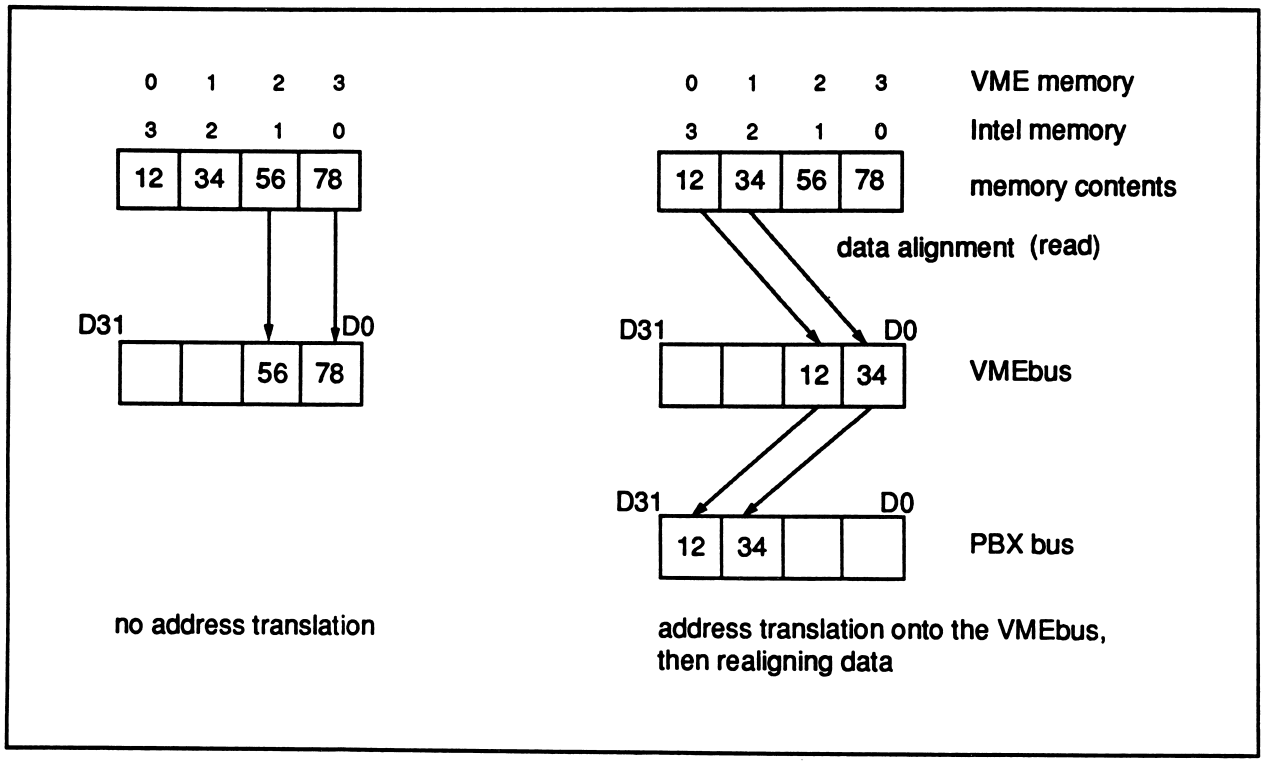


Figure 4-9. Address Translation for a 16-Bit Word

BYTE SWAPPING

Consider an array data structure. Both Intel and VME conventions place the first element of the array at address 0 and increment the address for subsequent elements. For example, assume that you write the string "abcd" as 32 bits. In Intel memory, the 'a' goes into address 0 while in VME memory the 'a' goes into address 3. The bytes are backwards. Figure 4-10 illustrates what happens if there is no byte swapping.

If you want the same locations to contain the same bytes, the BIA must perform byte swapping. Figure 4-11 illustrates what happens after byte swapping.

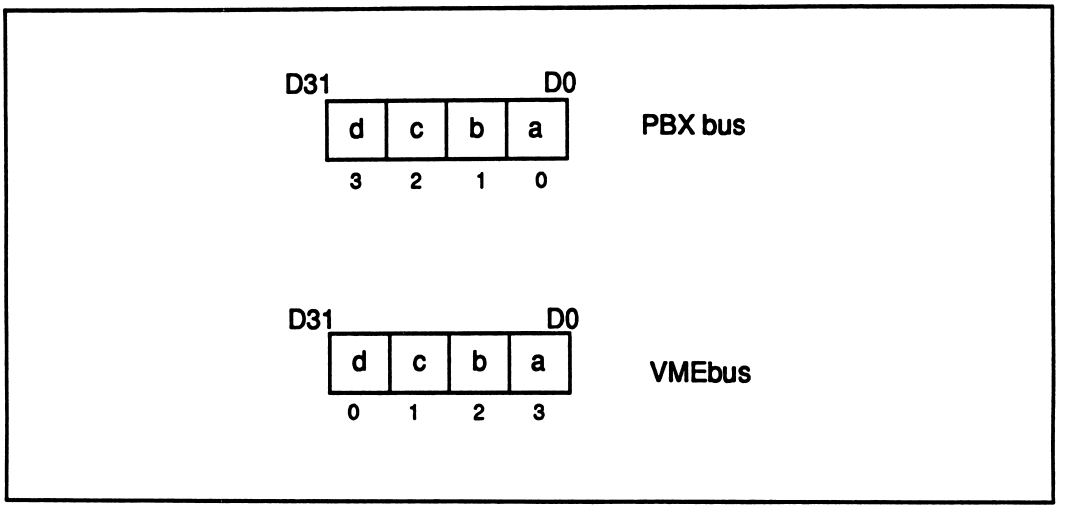


Figure 4-10. The Elements of the String "abcd" are in Different Addresses

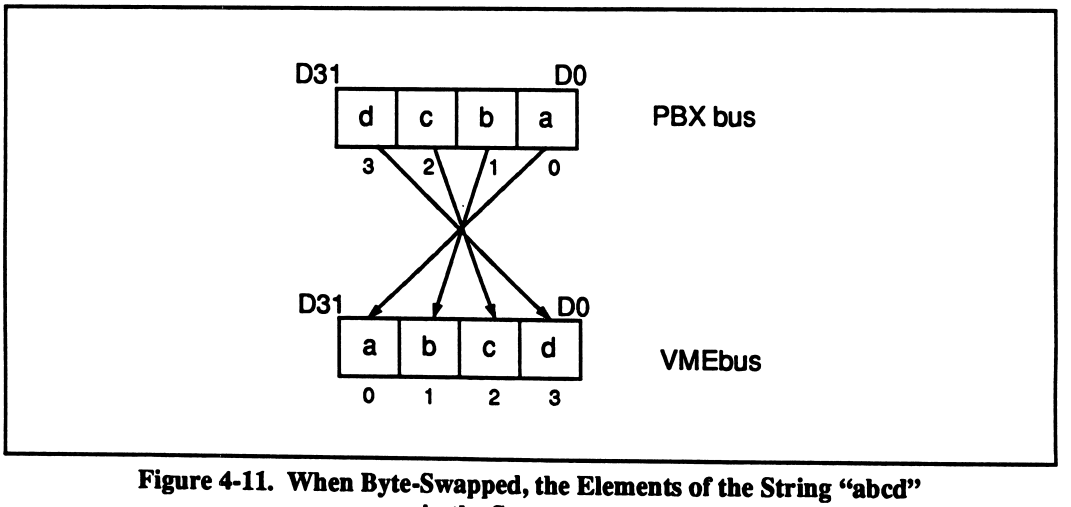


Figure 4-11. When Byte-Swapped, the Elements of the String "abcd" are in the Same Addresses

32-Bit Mode

Byte swapping	No
Address translation	Yes
Data alignment	Yes

Use this mode when writing 32-bit words into 32-bit memory. For example, this is the mode you should use for writing the BIA registers as **unsigned longs**.

In this mode VME memory appears as Intel memory. When you write 0x1234 5678 to VME address 0, the Intel processor reads 0x1234 5678 at address 0.

Note that in this mode if the Intel processor accesses portions of previously written 32-bit words as 16-bit words, it gets what it expects because of the address translation. For example, if you then read the word at address 0, you get 0x5678 as expected. Figure 4-12 illustrates writing and reading a 32-bit word in 32-Bit Mode.

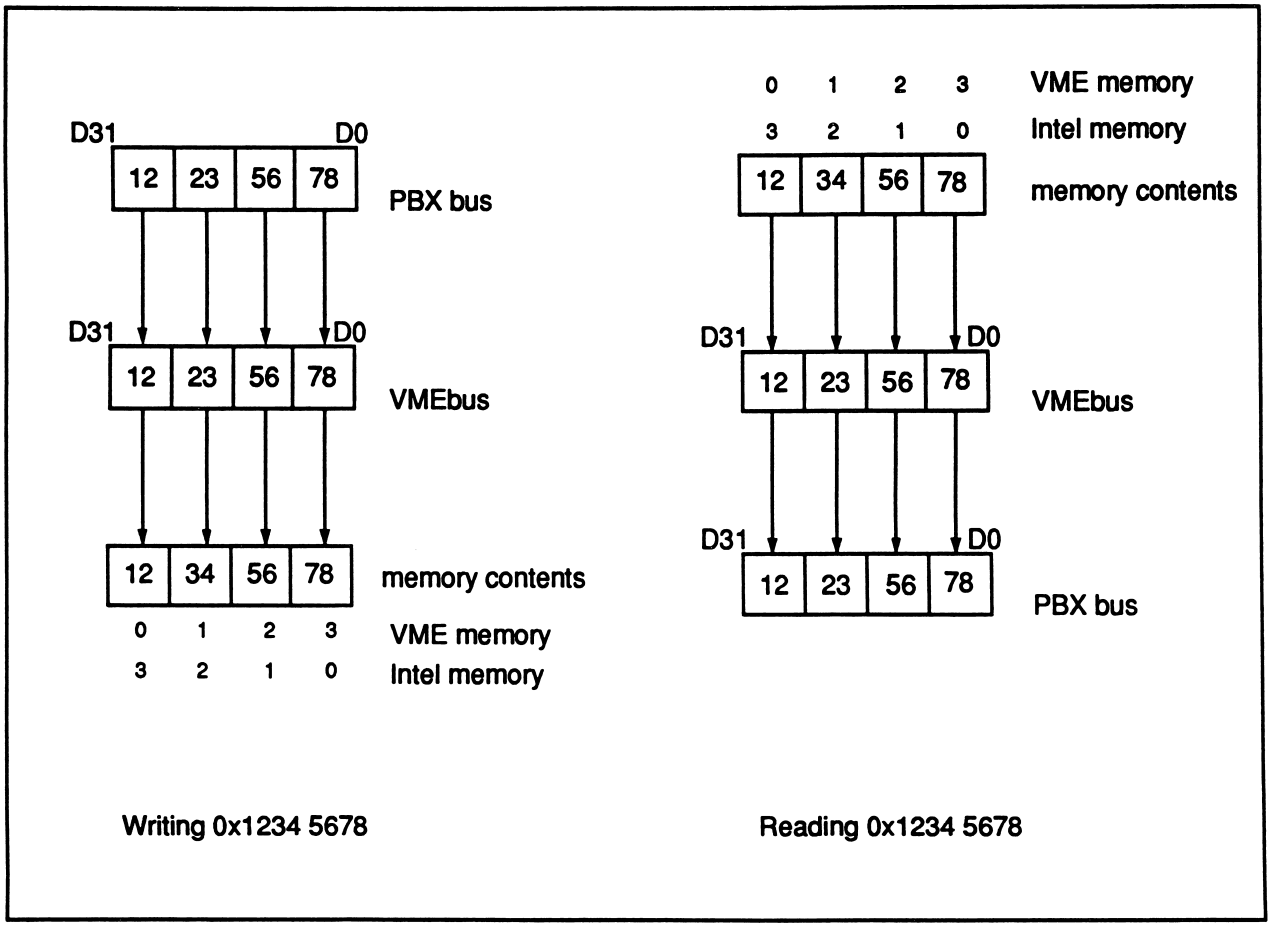


Figure 4-12. 32-Bit Mode: Writing and Reading a 32-Bit Word

Table 4-2. 32-Bit Mode

	PBX		VME	
	Address	**Data**	**Address**	**Data**
32-Bit	0	D31-D0	0	D31-D0
16-Bit	0	D15-D0	2	D15-D0
	1	D23-D8	1	D23-D8
	2	D31-D16	0	D15-D0
8-Bit	0	D7-D0	3	D7-D0
	1	D15-D8	2	D15-D8
	2	D23-D16	1	D7-D0
	3	D31-D24	0	D15-D8
24-Bit	unaligned	D23-D0	unaligned	D23-D0
	unaligned	D31-D8	unaligned	D31-D8

32-Bit Mode is intended for 32-bit reads and writes. Table 4-2 shows how it behaves for 16-bit and 8-bit reads and writes as well as 32-bit reads and writes. For example, consider writing a byte to address 2. The data appear on D23-D16 on the PBX bus. Because there is address translation, PBX address 2 corresponds to VME address 1. Because there is data alignment, the data appear on D7-D0.

16-Bit Mode

Byte swapping	No
Address translation	No
Data alignment	Yes

Use this mode when writing16-bit words into 16-bit memory. For example if your VME device has 16-bit registers mapped to certain VME address locations, use 16 Bit Mode.

Consider writing 0x1234 to address 0. The Intel processor puts the 16-bit word on D15-D0. Because of data alignment, that's also the word's position on the VMEbus. Because there is no address translation, the word is written into VME address 0.

When you read VME address 0, the 16-bit word appears on the lower half of the VMEbus. That's also the word's position on the PBX bus. Figure 4-13 illustrates writing and reading a 16-bit word at address 0 in 16 Bit Mode.

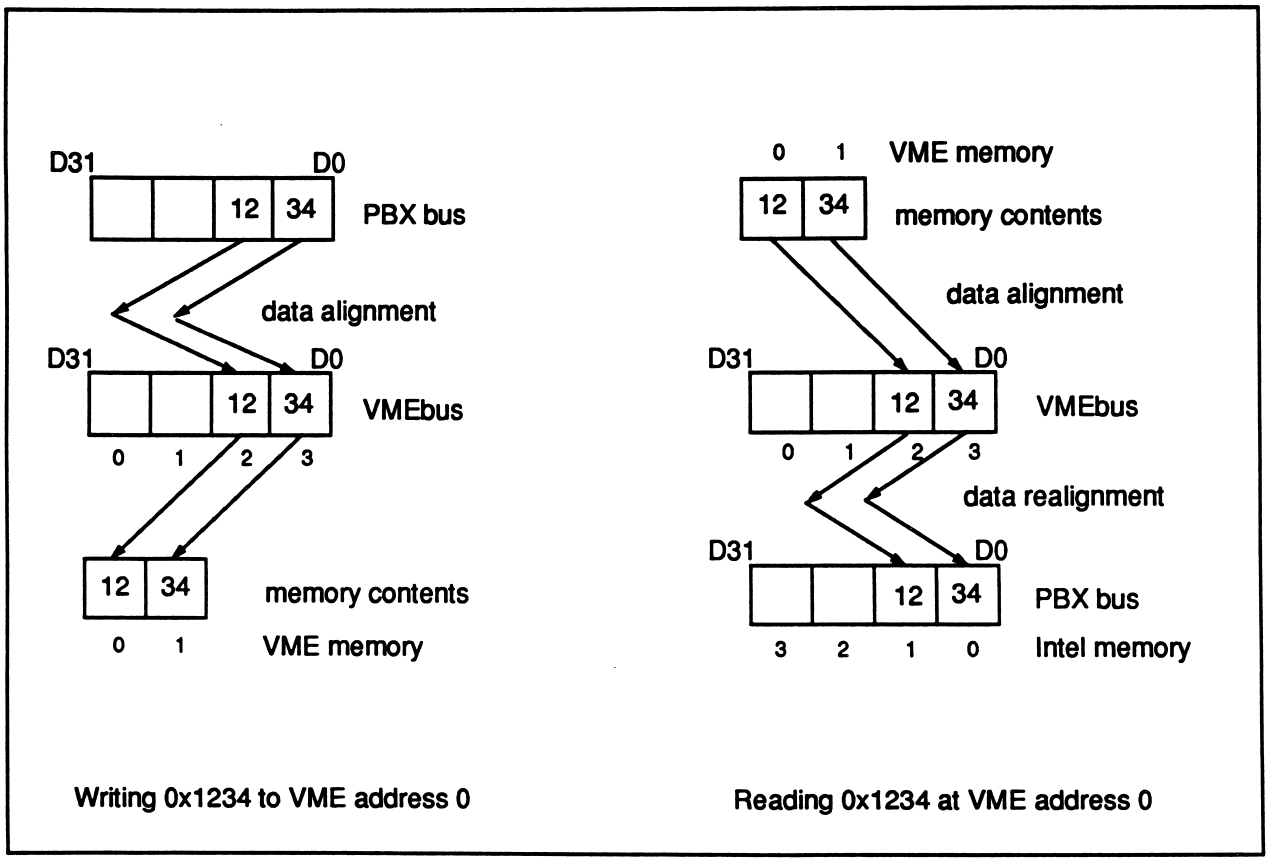


Figure 4-13. 16-Bit Mode: Writing and Reading a 16-Bit Word at Address 0

Table 4-3. 16-Bit Mode

	PBX		VME	
	Address	**Data**	**Address**	**Data**
16-Bit	0	D15-D0	0	D15-D0
	2	D31-D16	2	D15-D0
8-Bit	0	D7-D0	1	D7-D0
	1	D15-D8	0	D15-D8
	2	D23-D16	3	D7-D0
	3	D31-D24	2	D15-D8

16-Bit Mode is intended for 16-bit reads and writes. Table 4-3 shows how it behaves for both 16-bit and 8-bit reads and writes. For example, consider writing a byte to address 2. The data appear on D23-D16 on the PBX bus. Because there is data alignment, the data appear on D7-D0 on the VMEbus. Because there is no address translation, this position on the VMEbus corresponds to VME address 3.

8-Bit Mode

Byte swapping	Yes
Address translation	No
Data alignment	Yes

Use this mode when writing bytes into 8-bit memory. For example, if your VME device has 8-bit registers mapped to certain VME address locations, use 8-Bit Mode.

Consider writing 0x78 to address 0. The Intel processor puts the byte on D7-D0. Because of data alignment, you would expect the byte on D7-D0 of the VMEbus, but because of byte swapping it appears on D15-D8.

When you read address 0, the byte appears on D15-D8 of the VMEbus. It's byte-swapped to D7-D0 and data-realigned to the same place. Figure 4-14 illustrates writing and reading a byte at address 0 in 8-Bit Mode.

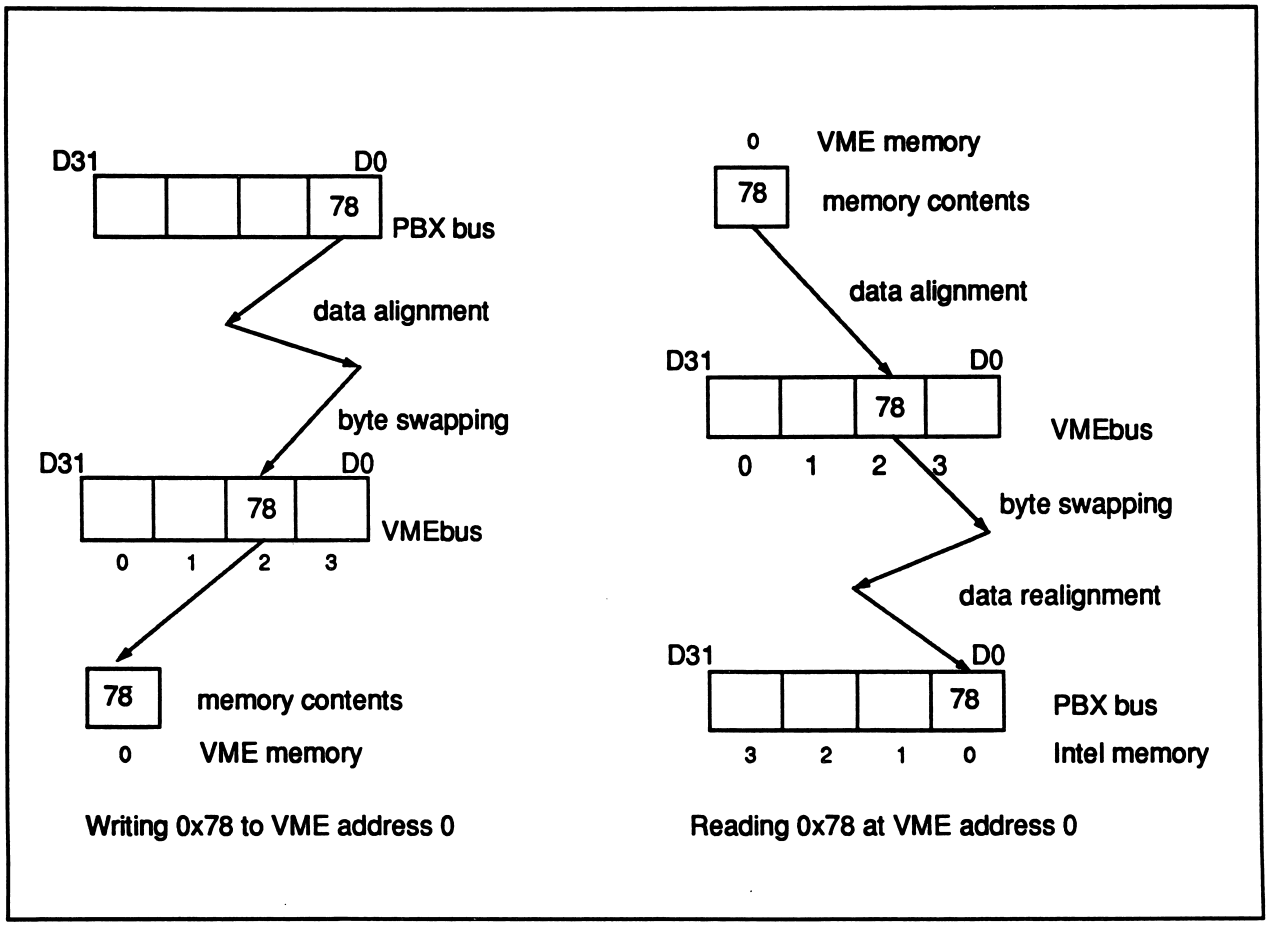


Figure 4-14. 8-Bit Mode: Writing and Reading a Byte at Address 0

Table 4-4. 8-Bit Mode

	PBX		VME	
	Address	Data	Address	Data
8-Bit	0	D7-D0	0	D15-D8
	1	D15-D8	1	D7-D0
	2	D23-D16	2	D15-D8
	3	D31-D24	3	D7-D0

8-Bit Mode is intended for byte reads and writes. Table 4-4 shows how it behaves for all Note that 8-Bit Mode acts identically to Array Mode for byte reads and writes.

Array Mode

Byte swapping	Yes
Address translation	No
Data alignment	Yes

Use this mode when writing arrays. In this mode VME byte addresses contain the same data as Intel byte addresses. This means that bytes are treated the same in 8-Bit Mode and Array Mode. The differences arise with 32-bit words and 16-bit words.

For example, consider writing the 32-bit word 0x1234 5678 to VME address 0. When byte swapping is in effect, each VME byte address contains the same bits as the corresponding Intel byte address. Figure 4-15 illustrates writing and reading a 32-bit word at address 0 in Array Mode.

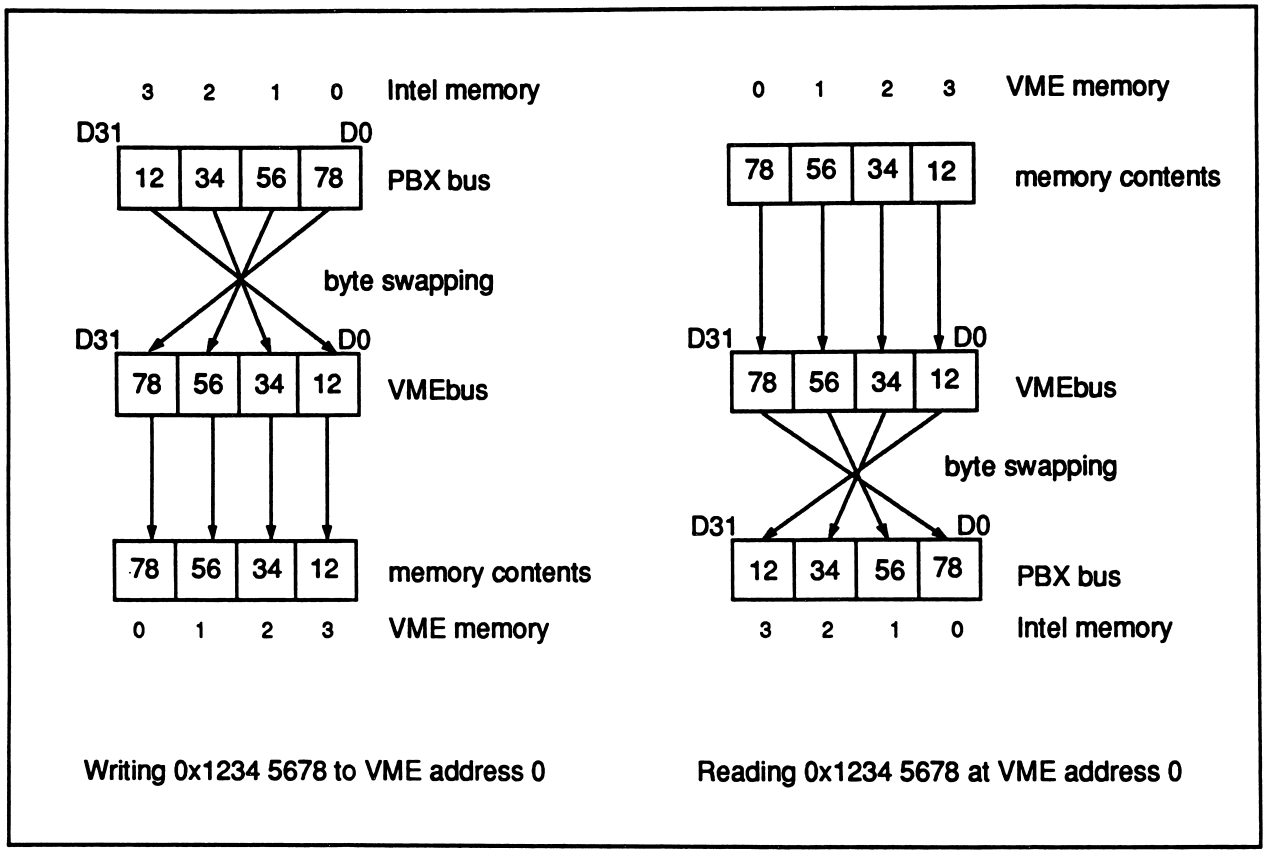


Figure 4-15. Array Mode: Writing and Reading a 32-Bit Word at Address 0

Now consider writing a 16-bit word to VME 16-bit memory at address 0. Data alignment ensures that the bytes are on the lower half of the VMEbus. Byte swapping ensures that the same bytes go into the same byte addresses. Figure 4-16 illustrates writing and reading a 16-bit word at address 0 in Array Mode.

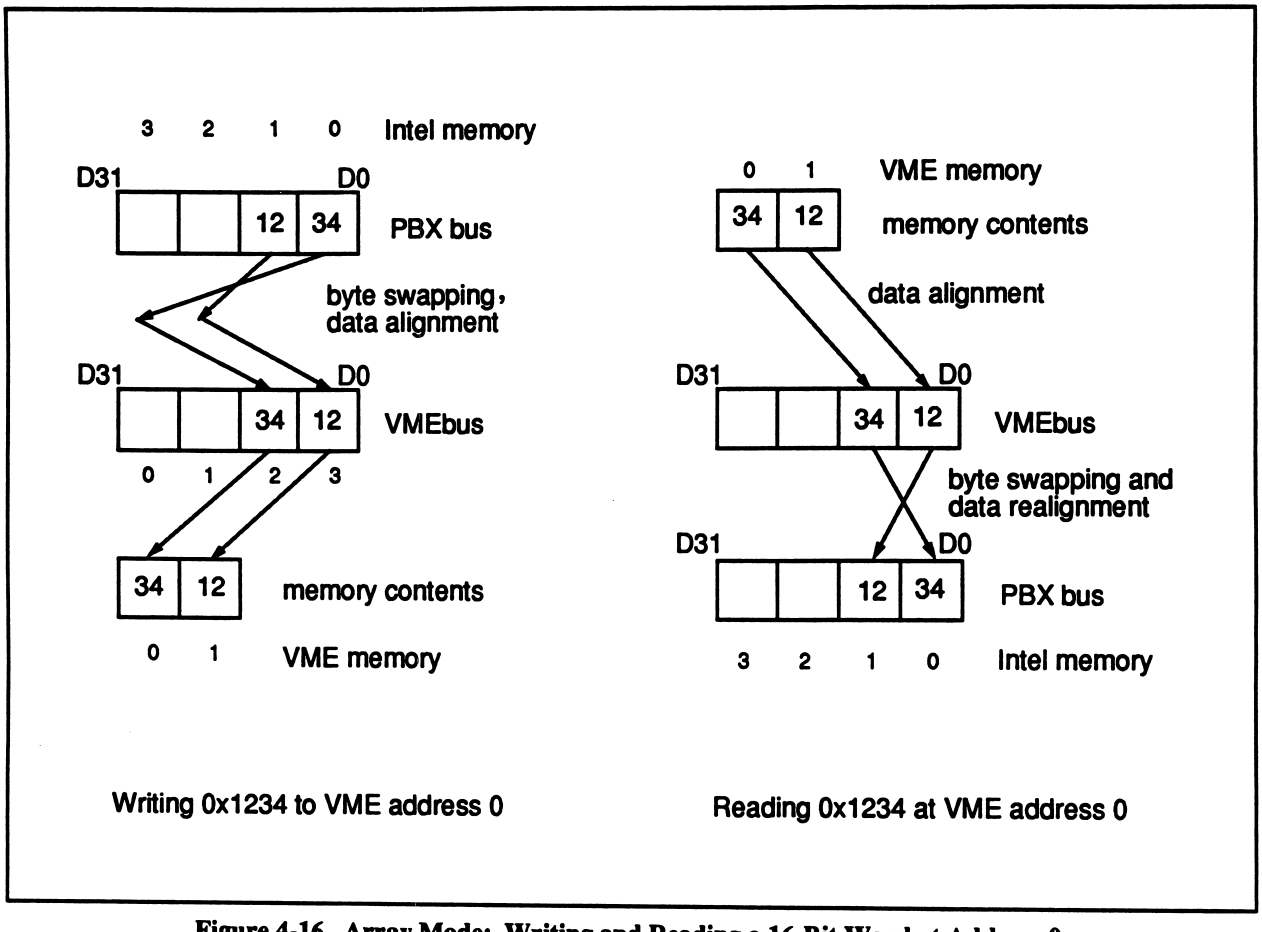


Figure 4-16. Array Mode: Writing and Reading a 16-Bit Word at Address 0

Now consider writing a 16-bit word to VME 16-bit memory at address 2. The bytes are on the upper half of the PBX bus, but data alignment ensures that they are on the lower half of the VMEbus. Byte swapping ensures that the same bytes go into the same byte addresses. Figure 4-17 illustrates writing and reading a 16-bit word at address 2 in Array Mode. Table 4-5 shows how Array Mode operates.

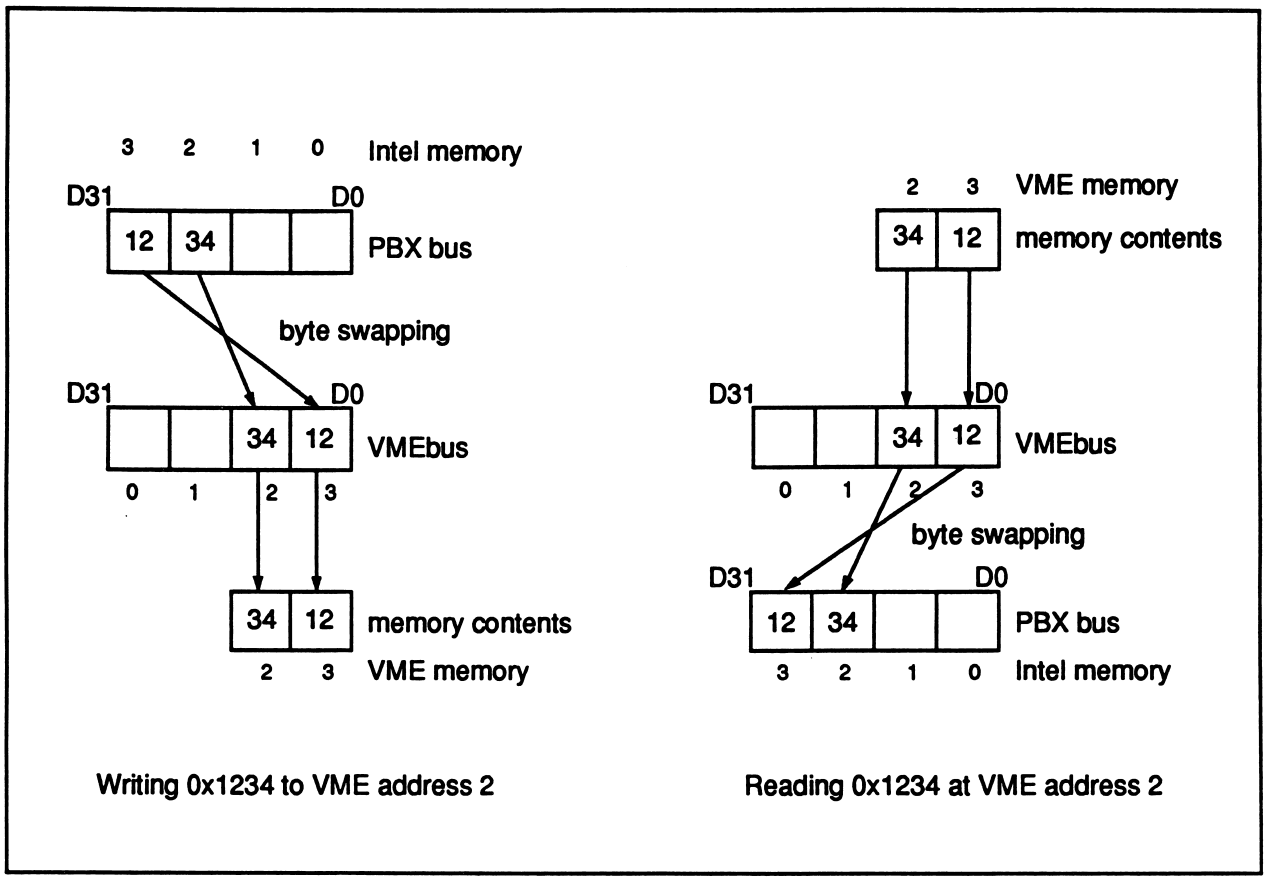


Figure 4-17. Array Mode: Writing and Reading a 16-Bit Word at Address 2

Table 4-5. Array Mode

	PBX		VME	
	Address	**Data**	**Address**	**Data**
32-Bit	0	D7-D0	0	D31-D24
(D0-D31)	0	D15-D8	0	D23-D16
	0	D23-D16	0	D15-D8
	0	D31-D24	0	D7-D0
16-Bit	0	D7-D0	0	D15-D8
(D15-D0)	0	D15-D8	0	D7-D0
(D8-D23)	1	D15-D8	1	D23-D16
	1	D23-D16	1	D15-D8
(D31-D16)	2	D23-D16	2	D15-D8
	2	D31-D24	2	D7-D0
8-Bit	0	D7-D0	0	D15-D8
	1	D15-D8	1	D7-D0
	2	D23-D16	2	D15-D8
	3	D31-D24	3	D7-D0
24-Bit	unaligned	D7-D0	unaligned	D31-D24
(D23-D0)	unaligned	D15-D8	unaligned	D23-D16
	unaligned	D23-D16	unaligned	D15-D8
(D31-D8)	unaligned	D15-D8	unaligned	D23-D16
	unaligned	D23-D16	unaligned	D15-D8
	unaligned	D31-D24	unaligned	D7-D0

HANDLING INTERRUPTS

You must write an exception handler to take care of PBX interrupts from the BIA. The NX/2 operating system expects one of two kinds of PBX/LBX interrupts. One is from the VX vector processing board; the other is from the BIA. These boards are mutually exclusive, so there is never any ambiguity.

Write your PBX interrupt handler in C, turn on PBX interrupts, and then attach your PBX handler to exception 29 with the system call **handler()**. To turn on PBX interrupts, use the assembly routine **out()** to write to the node's Programmable Interrupt Controller (PIC). Set bit 5 in the I/O node's port 0xc6 to 0. To do that, read in the port's value, AND in 0xdf, and write the updated value.

This handler must perform the following actions:

1. Read the Interrupt Status Register. This tells you whether the interrupt is one of the seven VME interrupts, a **VME BERR***, or a **VME SYSFAIL*** timeout. The Interrupt Status Register is not debounced. Hence, you may decide to read it twice to ensure that you get the correct value.

2. If the interrupt is **VME BERR*** or a **VME SYSFAIL*** timeout, do what needs to be done and then clear interrupt status by writing the Clear Latched Interrupt Status Register. Interrupt handling is now complete.

3. If the interrupt is one of the seven VME interrupts, servicing is device dependent. Some devices require an interrupt acknowledge; other devices just require that you read a VME device register. Be sure to check the user documentation for the VME device.

   Interrupt Acknowledge

   1. Set the **BIAIACK** bit in the BIA

   2. If the interrupt is one of IRQ1 through IRQ7, put out the appropriate interrupt acknowledge code on the address bus. The interrupt acknowledge code for IRQ1 is 0x0000 0002, that for IRQ2 is 0x0000 0004, etc.

      The interrupt acknowledge code is a three-bit code that must appear on address lines ADDR3-ADDR1. ADDR0 is ignored. So you could be in 8-Bit Mode and read 0x0000 0003 for IRQ1, 0x0000 0004 for IRQ2, etc.

      If the returned data is a byte, it appears on D7-D0 of the VMEbus. One way to read the data is to go into 16-Bit Mode and treat only the lower byte of the word that you read as significant.

      The data that you read (called the interrupt vector) may not be of use to you. It depends on your VME device. In most cases, however, you should still read the vector, even if you don't use it. For example, if your VME device is a BIT3 Adaptor, still read the vector, but ignore it. Instead, use the vector obtained from a register on the BIT3 board. Consult the appropriate BIT3 documentation.

      When you do use the vector, your response is usually to jump to that address. In C, you would cast the value to a function pointer and call the function.

3. Clear the **BIAIACK** bit in the BIA Control Register.

Release on Read Read the appropriate VME address. There is no need to toggle the **BIAIACK** bit.

Here is an example of a C code fragment that turns on PBX interrupts and attaches your handler procedure to the exception 29. It uses both the **out()** and **in()** assembly routines.

```
void bia_handler();
main()
{
long bia_etype;

 out(0xc6,in(0xc6) & 0xdf);
 bia_etype = 29;
 handler(bia_etype, bia_handler);
  .
  .
  .
}

void bia_handler()
{
  .
  .
  .
}
```

Here is an example of a C code fragment that handles a VME interrupt. This fragment might appear inside your handler. The code fragment reads the Interrupt Status Register, and determines whether the interrupt is **VME BERR***, **VME SYSFAIL*** timeout, or a VME interrupt. If the interrupt is a VME interrupt, the code fragment sets the **BIAIACK** bit in the BIA Control Register, reads the interrupt vector, and then clears **BIAIACK**.

```
#define CONTROL 0x0004
#define ISR 0x0000
#define CLR_INT 0x0000

#define ISR_SYSFAIL_ 0x0200
#define ISR_BIA_TO_ 0x0400
#define ISR_BERR_ 0x0800
#define ISR_IRQ1_ 0x0001
#define ISR_IRQ2_ 0x0002
#define ISR_IRQ3_ 0x0004
  .
  .
  .
```

```
#define BIA_REG 0x00c00000
#define BIA_VME 0x00800000
#define ISR_ALL_IRQ 0x007f
#define PAGE_MASK 0xffc00000
#define VME_IACK 0x4000

#define SHFL0       0x0008      /* LSB of memory mode */
#define SHFL1       0x0800      /* MSB of memory mode */
#define MODE_16_BIT SHFL1       /* 16 bit access Mode */
#define MODE_8_BIT  SHFL0       /* 8 bit access Mode */
#define MODE_32_BIT (SHFL0 | SHFL1) /* 32 bit access MOde */

unsigned short *bia_isr, *bia_ctrl, bia_ctrl_reg;
unsigned long base_addr;
unsigned short *b;
unsigned char vec; /* Depends on device */
 .
 .
 .
 /* Set 16-Bit Mode */
 bia_ctrl = (unsigned short *)(BIA_REG + CONTROL + 0xc0000000);
 bia_ctrl_reg &= ~(SHFL1 | SHFL0);
 bia_ctrl_reg |= MODE_16_BIT;
 *bia_ctrl = bia_ctrl_reg;

 base_addr = 0x00000000;
 base_addr &= ~PAGE_MASK;

/* When the base_addr is 0x0000 0000, masking out the PAGE
is really not necessary. */

 base_addr += BIA_VME;
 base_addr += 0xc0000000;

 bia_isr = (unsigned short *)(BIA_REG + ISR +0xc0000000);
 bia_ctrl = (unsigned short *)(BIA_REG + CONTROL +0xc0000000);
 bia_clr_isr = (unsigned short *)(BIA_REG + CLR_INT
+0xc0000000);


 if((~(*bia_isr)) & ISR_SYSFAIL_) {
 .
 .
/* Process a VME SYSFAIL* */

}
 else if((~(*bia_isr)) & ISR_BERR_) {
 .
 .
```

```
/* Process a VME BERR*. Then clear the latched interrupt status
by writing to the Clear Latched Interrupt Status Register. */

  *bia_clr_isr = 0;
}
 else {
  switch ((~(*bia_isr)) & ISR_ALL_IRQ) {
   case ISR_IRQ1_:
    b = (unsigned short *)(base_addr + 2 * 1);
    break;
   case ISR_IRQ2_:
    b = (unsigned short *)(base_addr + 2 * 1);
    break;
   case ISR_IRQ3_:
    b = (unsigned short *)(base_addr + 2 * 1);
    break;
 .
 .
 .
 } /* End of switch */

/* Set the BIAIACK bit in the BIA Control Register to 1*/
 *bia_ctrl |= VME_IACK;

/* Read the vector */
 vec = (char)*b;

/* Clear the IACK bit in the BIA Control Register */
 *bia_ctrl &= ~VME_IACK;
} /* End of else */
```

LOADING THE BIA DRIVER

The BIA driver runs as an application on the I/O node that's connected to the BIA. After you write the driver, compile it and load it on the I/O node. Use the **load** command and specify the direct node number identifying the I/O node that you want.

For example, assume that the source for your BIA driver is called *bia.c* and that it's located on the SRM. Also assume that the two assembly routines **in()** and **out()** are in *port.o*. You would compile it as follows:

```
cc -o bia bia.o port.o -node
```

To load it on the I/O node whose direct node number is 642, issue the following command:

```
load -D 642 bia
```

The direct node number is 640 + the physical slot number of the compute node whose channel 7 is connected to the I/O node. See the *iPSC®/2 and PSC®/860 System Administrator's Guide* for more information about direct node numbers. The -D option gives the program I/O privileges and you must specify it when loading programs that interact with the BIA board.

DETERMINING THE DIRECT NODE NUMBER

To determine the direct node number corresponding to an I/O node, look at the file *cubeconf* in the directory */usr/ipsc/conf*. Specifically, look at the CC*n* line. To get the direct node number, add 512 to the network node address in the CC*n* line.

The CC*n* lines list the network addresses of the I/O nodes in a standard cabinet. The *n* identifies the card cage and corresponds to the *n* in the USM*n* line.

Each card cage has its own USM. *n* is the USM ID and is jumpered on the USM itself. It starts at 0 and increments by one for every 16 slots in the system.

The CC*n* line has an entry for each slot in the card cage. The entry is a space if the slot is empty or contains a non-I/O node. If the slot contains an I/O node, the entry is the I/O node's network address.

For example, consider the following CC*n* line:

```
CC1 130, ,134, ,138, ,142, ,146, ,150, , , , ,
```

This line states card cage number 1 has six I/O nodes. They are in slots 0, 2, 4, 6, 8, and 10. Their network node numbers are 130, 134, 138, 142, 146, and 150. The corresponding direct node numbers are 642, 646, 650, 654, 658, and 662.

For a compute node, the network address is its slot number. Hence, for a compute node, the direct node number is 512 + its slot number.

COMMUNICATING WITH THE APPLICATION

When your application wants to use the BIA, it sends a message to the BIA's companion I/O node, specifying the I/O node's direct node number. Usually, an application specifies the logical node number; but when the value is 512 or above, the NX/2 operating system knows that direct node numbers are involved.

You may decide to send the BIA driver a structure, which it would then interpret; or you may decide to have the driver determine what it should do from the message type. The message passing system calls are described in the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual.*

BIA DIAGNOSTIC TESTS

The Cube Diagnostic Program (CDP) provides diagnostic tests for the BIA.

The tests verify the data written to the two write/read registers, the Interrupt Mask Register and the VME Address Register. Also, the tests verify VME data by reading the Loopback Test Register.

Reading the Loopback Test Register returns the data written through the byte swapping network when the BIA is in test mode. In this way you can check to see whether your data are swapped correctly.

Use the Loopback Test Register as follows:

1. Put the BIA in test mode. Set the bit **TESTNTWRK** in the BIA Control Register to 1 (true).

2. Write your data to the VME Address Register (0x00c0 000c).

3. Read the Loopback Test Register.

When the BIA is in test mode, do not read any other BIA registers and do not write to VME memory. To remove the BIA from test mode, set the **TESTNTWRK** bit in the BIA Control Register to 0 (false).

CDP uses the Loopback Test Register to test if the byte swapping network is working correctly.

Also, the BIA provides the ability to force certain kinds of interrupts to the I/O node. You can force a **VME SYSFAIL*** interrupt and a **VME BERR*** timeout interrupt.

To force a **VME SYSFAIL*** interrupt, the BIA driver must set the **SYSFAIL*** bit in the BIA Control Register to 0. This also causes the **VME SYSFAIL*** bit in the Interrupt Status Register to become 0 (true).

To force a **VME BERR*** timeout interrupt, the BIA driver can set the **RESET*** bit in the BIA Control Register to 0, then perform any kind of VME memory access. When the BIA is in the reset state, it will not respond, and hence a timeout error results. This also causes the **BIA BUS TIMEOUT*** and the **VME BERR*** bits in the Interrupt Status Register to become (0) true. A timeout interrupt also occurs if the VME board is not present. To return the BIA to operation (that is, take it out of the reset state), set **RESET*** to 1.

SOURCE CODE FOR EXAMPLE DRIVER A

INTRODUCTION

This chapter contains an example of a complete working BIA driver. This driver was written for a VME analog/digital, digital/analog controller. The intent is that you can use this driver as a template for more complicated BIA drivers. These files are in the directory */usr/ipsc/examples/bia-driver*.

ad.h

```
#define BASE              0x00000000        /* Base address of the AD board*/
#define CSR               BASE               /* A/D Control/Status register*/
#define ADCHAN            0x00000002               /* A/D Channel Register*/
#define PACLK             0x00000004               /* Pacer Clock Register*/
#define ADDATA            0x00000006                  /* A/D Data Register*/
#define DAC0              0x00000008                /* D/A 0 Data Register*/
#define DAC1              0x0000000A                /* D/A 1 Data Register*/
#define DIOREG            0x0000000C                  /* DIO Data Register*/

/*\
***   A/D Control Status Register bit values
\*/

#define CSR_ERR           0x8000
#define CSR_ERR_INT_EN    0x4000
#define CSR_ERR_CLR       0x2000
#define CSR_INIT_CLR      0x1000
#define CSR_UNUSED        0x0800
#define CSR_SCN_EN        0x0400
#define CSR_DAT_OUT1      0x0200
#define CSR_DAT_OUT2      0x0100
#define CSR_DONE          0x0080
#define CSR_DONE_INT_EN   0x0040
#define CSR_EOS           0x0020
#define CSR_EOS_INT_EN    0x0010
```

```
#define CSR_SCN_TRG_EN     0x0008
#define CSR_INC_EN         0x0004
#define CSR_TIMEOUT        0x0002
#define CSR_START          0x0001

/*\
***   Data out1 and Data out2 bits:
\*/

#define B0_B15_I           0x0
#define B0_B7_I_B8_B15_O   0x0100
#define B0_B7_O_B8_B15_I   0x0200
#define B0_B15_O           0x0300

/* Register setting operations */
#define SCSR               0x0000
#define SADCHAN            0x0001
#define SPACLK             0x0002
#define SADDATA            0x0003
#define SDAC0              0x0004
#define SDAC1              0x0005
#define SDIOREG            0x0006
#define SDTREGS            0x0007

/* Register getting operations */

#define GCSR               0x1000
#define GADCHAN            0x1001
#define GPACLK             0x1002
#define GADDATA            0x1003
#define GDAC0              0x1004
#define GDAC1              0x1005
#define GDIOREG            0x1006
#define GDTREGS            0x1007

/* High level operations */

#define ASSIGN             0x2000
#define DEASSIGN           0x2001

#define AINP               0x1012
#define AINI               0x1013
#define AOUT               0x1014
```

```
struct DTREGS {
   unsigned short reg_CSR;
   unsigned short reg_ADCHAN;
   unsigned short reg_PACLK;
   unsigned short reg_ADDATA;
   unsigned short reg_DAC0;
   unsigned short reg_DAC1;
   unsigned short reg_DIOREG;
};

struct AIO {
   unsigned short chan;
   unsigned short val;
};
/* Data Translation board Registers */

extern unsigned short *base;
extern unsigned short *csr;
extern unsigned short *adchan;
extern unsigned short *paclk;
extern unsigned short *addata;
extern unsigned short *dac0;
extern unsigned short *dac1;
extern unsigned short *dioreg;
```

bia.h

```
typedef unsigned char    byte;
typedef unsigned short   word16;
typedef unsigned long    word32;

#define TRUE             1
#define FALSE            0

#define MAX_BIA          3              /* Maximum number of BIA boards in system */
#define DEF_BIA          0                       /* Default BIA board number */
#define NUMPATS          7                      /* Number of static patterns */

#define BIA_VME          0x00800000           /* Base address of BIA VME space */
#define BIA_REG          0x00c00000           /* Base address of BIA registers */
#define BIA_OFFSET       0x01000000        /* Address offset for each BIA board */
#define BIA_VMESIZE BIA_REG-BIA_VME               /* VME address space per BIA */
#define BIA_REGSIZE BIA_OFFSET-BIA_REG/* Register address space per BIA */

#define PAG_MASK         0xffc00000                    /* Mask for page bits */
#define MOD_MASK         0x0000003f                  /* Mask for modifier bits */
#define P_M_MASK         (PAG_MASK|MOD_MASK)        /* Mask for PAG_MOD register */


/* These defines are offsets from a BIA base address for each register */
#define ISR              0x0000                    /* Read interrupt status reg */
#define CLR_INT          0x0000                   /* Clear interrupt status reg */
#define SWAP_REG         0x0004                  /* Read byte swapping test reg */
#define CONTROL          0x0004                       /* Write BIA control reg */
#define IMR              0x0008                /* Read/write interrupt mask reg */
#define PAG_MOD          0x000c               /* Read/write VME page address reg */


/* These defines are for the interrupt status register bits */
#define ISR_IRQ1_        0x0001                          /* Low true, VME IRQ1 */
#define ISR_IRQ2_        0x0002                          /* Low true, VME IRQ2 */
#define ISR_IRQ3_        0x0004                          /* Low true, VME IRQ3 */
#define ISR_IRQ4_        0x0008                          /* Low true, VME IRQ4 */
#define ISR_IRQ5_        0x0010                          /* Low true, VME IRQ5 */
#define ISR_IRQ6_        0x0020                          /* Low true, VME IRQ6 */
#define ISR_IRQ7_        0x0040                          /* Low true, VME IRQ7 */
#define ISR_U1           0x0080                               /* Unused bit 1 */
#define ISR_DTACK_       0x0100        /* Low true, signals slave read/wrote VME */
#define ISR_SYSFAIL_     0x0200           /* Low true, signals VME SYSFAIL line */
#define ISR_BIA_TO_      0x0400             /* Low true, BIA bus time out error */
#define ISR_BERR_        0x0800                    /* Low true, VME bus error */
#define ISR_S1           0x1000                             /* Spare bit 1 */
#define ISR_U2           0x2000                            /* Unused bit 2 */
#define ISR_U3           0x4000                            /* Unused bit 3 */
#define ISR_U4           0x8000                            /* Unused bit 4 */
```

```
#define ISR_NONE          0xf7ff          /* Value indicating no interrupts pending */
#define ISR_ALL_IRQ       0x007f                  /* Mask value for IRQ levels only */


/* These defines are for the BIA CONTROL register bits */
#define LED_GRN           0x0001                                        /* Green LED */
#define LED_RED           0x0002                                          /* Red LED */
#define LED_YEL           0x0004                                       /* Yellow LED */
#define SHFL0             0x0008                                /* LSB of memory mode */
#define CTRL_U1           0x0010                                     /* Unused bit 1 */
#define CTRL_U2           0x0020                                     /* Unused bit 2 */
#define CTRL_U3           0x0040                                     /* Unused bit 3 */
#define CTRL_U4           0x0080                                     /* Unused bit 4 */
#define SYSFAIL_          0x0100              /* Low true, asserts VME SYSFAIL line */
#define RESET_            0x0200                                /* Low true, asserts ?? */
#define TRISTATE_         0x0400           /* Low true, VME address, data, ctrl high */
#define SHFL1             0x0800                                 /* MSB of memory mode */
#define TEST_NET          0x1000          /* Enables reading of test network latches */
#define BLK_MODE          0x2000                /* Enables VME block mode transfers */
#define VME_IACK          0x4000          /* Enables VME interrupt acknowledge cycle */
#define INT_ENBL          0x8000        /* Enables interrupts through BIA to PBX node */

/* Values for the Memory Access Modes */

#define MODE_ARRAY        0                                      /* Array Compatibility */
#define MODE_16_BIT       SHFL1                                   /* 16 bit access Mode */
#define MODE_8_BIT        SHFL0                                    /* 8 bit access Mode */
#define MODE_32_BIT       (SHFL0 | SHFL1)                         /* 32 bit access MOde */

/* Value to reset entire BIA control register to */
#define RESET_CTRL        (SYSFAIL_ | RESET_ | TRISTATE_)
#define LED_ON_DELAY      1500L                              /* LED on for 1.5 seconds */
#define LED_OFF_DELAY     500L                               /* LED off for .5 seconds */


/* These defines are for the interrupt mask register bits */
#define IMR_SYSFAIL       0x0001                        /* Mask off VME SYSFAIL line */
#define IMR_BIA_TO        0x0002                 /* Mask off BIA bus time out error */
#define IMR_BERR          0x0004                          /* Mask off VME bus error */
#define IMR_IRQ1          0x0008                               /* Mask off VME IRQ1 */
#define IMR_IRQ2          0x0010                               /* Mask off VME IRQ2 */
#define IMR_IRQ3          0x0020                               /* Mask off VME IRQ3 */
#define IMR_IRQ4          0x0040                               /* Mask off VME IRQ4 */
#define IMR_IRQ5          0x0080                               /* Mask off VME IRQ5 */
#define IMR_IRQ6          0x0100                               /* Mask off VME IRQ6 */
#define IMR_IRQ7          0x0200                               /* Mask off VME IRQ7 */
#define IMR_U1            0x0400                                    /* Unused bit 1 */
#define IMR_U2            0x0800                                    /* Unused bit 2 */
#define IMR_U3            0x1000                                    /* Unused bit 3 */
#define IMR_U4            0x2000                                    /* Unused bit 4 */
#define IMR_U5            0x4000                                    /* Unused bit 5 */
#define IMR_U6            0x8000                                    /* Unused bit 6 */
```

```
#define IMR_ALL_IRQ      0x03f8                                  /* All the IRQs */
#define IMR_MASKALL      0xffff                          /* Mask off all interrupts */
#define IMR_NO_MASK      0                             /* No masking of any interrupts */


/* These defines are for the PAG_MOD register */
#define STD_SUP_BLK      0x003f                     /* A24 supervisory block transfer */
#define STD_SUP_DATA     0x003d                        /* A24 supervisory data access */
#define STD_NO_P_BLK     0x003b                  /* A24 non-privileged block transfer */
#define STD_NO_P_DATA    0x0039                     /* A24 non-privileged data access */
#define SHORT_SUP        0x002d                           /* A16 supervisory access */
#define SHORT_NO_P       0x0029                       /* A16 non-privileged access */
#define EXT_SUP_BLK      0x000f                     /* A32 supervisory block transfer */
#define EXT_SUP_DATA     0x000d                        /* A32 supervisory data access */
#define EXT_NO_P_BLK     0x000b                  /* A32 non-privileged block transfer */
#define EXT_NO_P_DATA    0x0009                     /* A32 non-privileged data access */

/* BIA globals */

extern word16            *bia_ctrl;                  /* For writing CONTROL register */
extern word16            *bia_isr;                             /* For reading ISR */
extern word16            *bia_clr_isr;                        /* For clearing ISR */
extern word16            *bia_imr;                     /* For writing/reading IMR */
extern word32            *bia_page_mod;/* For writing/reading page/modifier register */
extern word32            *bia_swap_reg;    /* For reading swap network test register */

extern word16            bia_ctrl_reg;    /* CONTROL reg is write only, save it here */

extern word32            bia_base_reg;          /* Base of registers for default BIA */

extern word32            vme_base_reg    /* Base of VME memory space for default BIA */

/* Flags */
extern byte              bia_e_flg;        /* Enable/disable A32 extended addressing */

/*\
***   PIC ports and values
\*/

#define SPIC_PORTB       0xC6
#define NBCR_PORT        0x80
#define NUSMINT          0x20
#define LBXENBL          (NUSMINT | 0x8)
```

driver.h

```
#define PROT_VERSION 0

#define BIA_BUFF_SIZE      1024*4

#define BIA_MSG_TYPE       123

#define MAKE_BIA_HANDLE(pid, node)          ((node << 16) | (pid & 0xffff))

#define GET_DEST_NODE(handle)               ((handle >> 16) & 0xffff)
#define GET_DEST_PID(handle)                (handle & 0xffff)

struct BIA_BUFFER {
   unsigned long request;
   unsigned long status;
   unsigned long source_pid;
   unsigned long source_node;
   unsigned short prot_version;
   unsigned short sequence;
   char value[BIA_BUFF_SIZE];
};
```

port.s

```
.file    "port.s"

/ This module contains port in and out (byte wide) interface routines

.align    4

/    Calling Sequence:
/          in(port);
/
/    Description:
/          Input a byte from an I/O port.
/
/    Parameters:
/          port:    I/O port address.
/
/    Returns:
/          The input value.
/
.globl    in
in:

        push    %ebp
        mov     %esp, %ebp
        mov     8(%ebp), %edx
        mov     $0, %eax
        .byte     0xEC              / inb     %dx
        pop     %ebp
        ret

/    Calling Sequence:
/          out(port, value);
/
/    Description:
/          Output to an I/O port.
/
/    Parameters:
/          port:    The I/O port address.
/          value:    The value to output.
/
/    Returns:
/          none
/
.globl    out
```

```
out:

        push    %ebp
        mov     %esp, %ebp
        mov     8(%ebp), %edx
        movb    12(%ebp), %al
        .byte   0xEE                    / outb    %dx
        pop     %ebp
        ret
```

ad_global.c

```
/*
  *This module contains global definitions for DT registers
*/
unsigned short *base;                          /* Base of DT memory (jumpered)*/
unsigned short *csr;                               /* Control Status Register*/
unsigned short *adchan;                      /* Analog to Digital Channel reg*/
unsigned short *paclk;                                 /* Pacer Clock Register*/
unsigned short *addata;                    /* Analog to Digital Data Register*/
unsigned short *dac0;                                  /* Digital to Analog #0*/
unsigned short *dac1;                                  /* Digital to Analog #1*/
unsigned short *dioreg;                                 /* Digital IO Register*/
```

bia_global.c

```
/*
 *This module contains BIA Global register and variables
*/
#include "bia.h"                        /* BIA constants and memory locations */

/* BIA globals */

word16 *bia_ctrl;                              /* For writing CONTROL register */
word16 *bia_isr;                     /* For reading interrupt status register */
word16 *bia_clr_isr;                /* For clearing interrupt status register */
word16 *bia_imr;               /* For writing/reading interrupt mask register */
word32 *bia_page_mod;      /* For writing/reading page and modifier register */
word32 *bia_swap_reg;               /* For reading swap network test register */

word16 bia_ctrl_reg;        /* Since CONTROL reg is write only, save it here */


word32 bia_base_reg =                    /* Base of registers for default BIA */
   (BIA_OFFSET * DEF_BIA + BIA_REG);

word32 vme_base_reg = (BIA_OFFSET * DEF_BIA + BIA_VME);
/* Base of VME memory space for default BIA */

byte   bia_e_flg = FALSE;
                                    /* Enable/disable A32 extended addressing */
```

driver.c

```
/*
 *This module contains A Sample Driver for the Data Translation
 *Analog Digital I/O board.
*/

#include <stdio.h>
#include <errno.h>
#include "ad.h"
#include "driver.h"
#include "bia.h"

/*
 *  This is the driver process for the Data translation Analog/Digital
 *  I/O board.
 *  General structure of the dirver:
 *    - The driver is an NX process which runs on the Service Nodes adjacent
 *      to the slot contaning the BIA/DT1401.
 *    - Processes on the Cube "drive" the A/D board via this process.
 *    - The interface between the user processes and the Driver is
 *      the "biactl" system call.
 *    - Format of the "biactl":
 *          biactl(handle, COMMAND, buffer, size)
 *      where,
 *         handle:  Identifies the BIA board and the driver process.
 *         COMMAND: A valid command to the BIA or the Attached board.
 *         buffer:  A pointer to a buffer which either contains the
 *                  data to be given to the board, or a place to get
 *                  information from the board.
 *         size:    The size of the buffer.
 */
FILE *BiaErrorFile;
struct BIA_BUFFER BiaBuffer;
struct DTREGS regs;
struct AIO Aio;
unsigned long BiaOwner;
unsigned short BiaValue;
unsigned short BiaRequest;
long BiaDebug;
```

```
DtIntWait()
{
   unsigned short Csr;
   unsinged short vec;
   /*
    * If the owner is not there anymore ignore it
    */
   if( BiaOwner ){

      /* Get the pid and node of the owner of the bia board */
      BiaBuffer.source_pid = GET_DEST_PID(BiaOwner);
      BiaBuffer.source_node = GET_DEST_NODE(BiaOwner);

      /* Look at the error bit on the csr return error if it is on */
      Csr = *csr;
      if (Csr & CSR_ERR){
         BiaBuffer.status = CSR_ERR;
      }
      else{
      /* If all is OK read the data and copy it to message buffer */

         Aio.val = *addata;
         memcpy((char *)BiaBuffer.value,
                (char *) &Aio, sizeof(Aio));
         BiaBuffer.status = 0;
      }

      /* Ack the bia interrupt and clear error bits */
      get_VME_vec(BASE, &vec);
      clear_bia_isr();

      /* Send the data to the owner */
      if (_csend(BIA_MSG_TYPE, &BiaBuffer, sizeof(BiaBuffer),
                 BiaBuffer.source_node,
                 BiaBuffer.source_pid) < 0){
         fprintf(BiaErrorFile, "bia-dt interrupt handler: Send failed.\n");
         exit(1);
      }
   }
}
/*
 *   This routine read data from specified AD channel.
 *   If inter is on the end of conversion is notified by an interrupt
 *   otherwise it polls for the end of conversion
 */
```

```
DtAin(chan, inter)
unsigned short chan;
unsigned short inter;
{


   /* Poll the time-out bit */
   while ((*csr & CSR_TIMEOUT));

   /* Set the channel */
   *adchan = chan;

   if (inter){
      /* Setup the interrupt handler and start the conversion */
      handler(29, DtIntWait);
      *csr = (CSR_START | CSR_DONE_INT_EN);
   }
   else{
      /* Start the conversion and poll the DONE bit */
      *csr = CSR_START;
      while (!(*csr & CSR_DONE));
      /* If there is an error return error otherwise return data */
      if (*csr & CSR_ERR){
         BiaBuffer.status = CSR_ERR;
      }
      else{
         Aio.val = *addata;
         memcpy((char *)BiaBuffer.value,
                              (char *) &Aio, sizeof(Aio));
         BiaBuffer.status = 0;
      }
   }
}
```

```
short DtAout(chan, val)
unsigned short chan;
unsigned short val;
{
   /* Out put a value to DAC number "chan" */
   short stat;
   stat = 0;
   switch(chan){
      case 0:
                              *dac0 = val;
                              break;
      case 1:
                              *dac1 = val;
                              break;
      default:
                              stat = -1;
                              break;
   }
   return(stat);
}


main(argc, argv)
int argc;
char **argv;
{
   int numrecs = 0;
   unsigned long tmp;

   if (argc == 2){
      BiaErrorFile = fopen(argv[1], "w");
      BiaDebug = 1;
   }
   else{
      BiaErrorFile = stderr;
      BiaDebug = 0;
   }

   BiaOwner = 0;   /* Nobody owns the bia board */

   init_board(); /* Initialize the BIA  and  the DT board */
```

```
/*
*    Continuously wait for a message and act based on the requests
*/

   for(;;){

#ifdef DEBUG
      printf("Bia driver: posting receive %d\n", numrecs++);
#endif

/*
 *    Reveive a message.
*/

      if (_crecv(BIA_MSG_TYPE, &BiaBuffer, sizeof(BiaBuffer)) < 0){
                           fprintf(BiaErrorFile, "bia-dt driver: Crecv
failed\n");
      exit(1);
      }

      /*\
      ***   Construct the owner id. If the requester is not the owner
      ***   and the board already has an owner return an error.
      ***   otherwise perform the request.
      \*/
      tmp = MAKE_BIA_HANDLE(BiaBuffer.source_pid, BiaBuffer.source_node);
      if((tmp != BiaOwner) && BiaOwner){
         BiaBuffer.status = EACCES;
         if (_csend(BIA_MSG_TYPE, &BiaBuffer, sizeof(BiaBuffer),
                  BiaBuffer.source_node, BiaBuffer.source_pid) < 0) {
            fprintf(BiaErrorFile, "bia-dt driver: Send failed.\n");
            exit(1);
         }
       continue;
      }

      BiaRequest = (unsigned short) BiaBuffer.request;

      BiaBuffer.status = 0;
      switch (BiaRequest){
```

```
/*\
***   Set register operations
\*/
        case SCSR:
           memcpy( (char *) &BiaValue, (char *)BiaBuffer.value,
                    sizeof(BiaValue));
           *csr = BiaValue;
           break;
        case SADCHAN:
           memcpy( (char *) &BiaValue, (char *)BiaBuffer.value,
                   sizeof(BiaValue));
           *adchan = BiaValue;
           break;
        case SPACLK:
           memcpy( (char *) &BiaValue, (char *)BiaBuffer.value,
                    sizeof(BiaValue));
           *paclk = BiaValue;
           break;
        case SADDATA:
           memcpy( (char *) &BiaValue, (char *)BiaBuffer.value,
                    sizeof(BiaValue));
           *addata = BiaValue;
           break;
        case SDAC0:
           memcpy( (char *) &BiaValue, (char *)BiaBuffer.value,
                    sizeof(BiaValue));
           *dac0 = BiaValue;
           break;
        case SDAC1:
           memcpy( (char *) &BiaValue, (char *)BiaBuffer.value,
                    sizeof(BiaValue));
           *dac1 = BiaValue;
           break;
        case SDIOREG:
           memcpy( (char *) &BiaValue, (char *)BiaBuffer.value,
                    sizeof(BiaValue));
           *dioreg = BiaValue;
           break;
```

```
/*\
***   Get register operations
\*/

 case GCSR:
    BiaValue = *csr;
    memcpy((char *)BiaBuffer.value,
           (char *) &BiaValue, sizeof(BiaValue));
    break;
 case GADCHAN:
    BiaValue = *adchan;
    memcpy((char *)BiaBuffer.value,
           (char *) &BiaValue, sizeof(BiaValue));
    break;
 case GPACLK:
    BiaValue = *paclk;
    memcpy((char *)BiaBuffer.value,
           (char *) &BiaValue, sizeof(BiaValue));
    break;
 case GADDATA:
    BiaValue = *addata;
    memcpy((char *)BiaBuffer.value,
           (char *) &BiaValue, sizeof(BiaValue));
    break;
 case GDAC0:
    BiaValue = *dac0;
    memcpy((char *)BiaBuffer.value,
           (char *) &BiaValue, sizeof(BiaValue));
    break;
 case GDAC1:
    BiaValue = *dac1;
    memcpy((char *)BiaBuffer.value,
           (char *) &BiaValue, sizeof(BiaValue));
    break;
 case GDIOREG:
    BiaValue = *dioreg;
    memcpy((char *)BiaBuffer.value,
           (char *) &BiaValue, sizeof(BiaValue));
    break;
 case GDTREGS:
    regs.reg_CSR = *csr;
    regs.reg_ADCHAN = *adchan;
    regs.reg_PACLK = *paclk;
    regs.reg_ADDATA = *addata;
    regs.reg_DAC0 = *dac0;
    regs.reg_DAC1 = *dac1;
    regs.reg_DIOREG = *dioreg;
    memcpy((char *)BiaBuffer.value, (char *) &regs, sizeof(regs));
    break;
```

```
        /*\
        ***  Board assignment requests
        \*/

        case ASSIGN:
           init_board();
           BiaOwner =
              MAKE_BIA_HANDLE(BiaBuffer.source_pid, BiaBuffer.source_node);
#ifdef DEBUG
printf("Assign: Owner = %x\n", BiaOwner);
#endif
          break;
        case DEASSIGN:
           init_board();
#ifdef DEBUG
printf("Deassign: Owner = %x\n", BiaOwner);
#endif
           BiaOwner = 0;
           break;

       /*\
       *** Analog IO operations.
       \*/

        case AINI:        /* Analog input with interrupt */
           memcpy( (char *) &Aio, (char *)BiaBuffer.value,
                   sizeof(Aio));
           DtAin(Aio.chan, TRUE);
           continue;  /* do not send a message back until interrupt */
           break;

        case AINP:        /* Analog input with polling */
           memcpy( (char *) &Aio, (char *)BiaBuffer.value,
                   sizeof(Aio));
           DtAin(Aio.chan, FALSE);
           break;

        case AOUT:
           memcpy( (char *) &Aio, (char *)BiaBuffer.value,
                   sizeof(Aio));
           BiaBuffer.status = DtAout(Aio.chan, Aio.val);
           break;

        default:
           fprintf(BiaErrorFile, "bia-dt driver: Illegal Request\n");
           BiaBuffer.status = EBADRQC;
     }
```

```
#ifdef DEBUG
printf("Bia driver: responding to pid %d node %d\n",
        BiaBuffer.source_pid,
        BiaBuffer.source_node);
#endif
      /* Send the message back to the requestor */
      if (_csend(BIA_MSG_TYPE, &BiaBuffer, sizeof(BiaBuffer),
                BiaBuffer.source_node,
                BiaBuffer.source_pid) < 0){
         fprintf(BiaErrorFile, "bia-dt driver: Send failed.\n");
         exit(1);
      } /* end of if */
   } /* end of switch */
} /* end of main */
```

interface.c

```
/*
 *This module contains application interface to the driver
 */
#include <stdio.h>
#include <errno.h>
#include "ad.h"
#include "driver.h"

struct BIA_BUFFER BiaBuffer;

extern int errno;

long biactl(handle, command, buffer, size)
unsigned long handle;
unsigned long command;
char *buffer;
unsigned long size;
{
   long destnode;
   long destpid;

   /*
    * Error checking:
    * If handle is zero or size of buffer is too large return error.
    */
   if (!handle){
      fprintf(stderr, "biactl: no such device\n");
      errno = ENXIO;
      return(-1);
   }
   if (size > BIA_BUFF_SIZE){
      fprintf(stderr, "biactl: Buffer too large\n");
      errno = EQBLEN;
      return(-1);
   }
```

```
    /*
     * Send the data to the driver, and get the results back.
     */
    destnode = GET_DEST_NODE(handle);
    destpid = GET_DEST_PID(handle);

    BiaBuffer.request = command;
    memcpy((char *)BiaBuffer.value, (char *)buffer, size);

    BiaBuffer.source_pid = mypid();
    BiaBuffer.source_node = _mydirect();

    csend(BIA_MSG_TYPE, &BiaBuffer, sizeof(BiaBuffer), destnode, destpid);
    crecv(BIA_MSG_TYPE, &BiaBuffer, sizeof(BiaBuffer));

    memcpy((char *)buffer, (char *)BiaBuffer.value, size);
    errno = BiaBuffer.status;

    if (errno){
       return(-1);
    }
    else{
       return(0);
    }
}
/*
 * Assign the BIA Device to the process.
 * The handle is a combination of the direct node number and
 * process number of the driver process.
 * Note: For this example the pid of the driver process is hardwired to 222
 *       and, the Node number of the driver process is 512 (Direct).
 */

openbia()
{
    unsigned long handle;

    handle = MAKE_BIA_HANDLE(222, 512);
    if(!biactl(handle, ASSIGN, "", 0)){
       return(handle);
    }
    else
       return(-1);
}
```

```
/*
 * Unassign the Board
 */
closebia(handle)
unsigned long handle;
{
   if(!biactl(handle, DEASSIGN, "", 0))
      return(0);
   else
      return(-1);
}

/*
 * Return the value read from the AIN channel
 */
unsigned short ain(handle, chan)
unsigned long handle;
unsigned short chan;
{
   struct AIO Aio;

   Aio.chan = chan;
   biactl(handle, AINI, &Aio, sizeof(Aio));
   return(Aio.val);
}

/*
 * Output a value to the Aout channel
 */

aout(handle, chan, val)
unsigned long handle;
unsigned short chan;
unsigned short val;
{
   struct AIO Aio;

   Aio.chan = chan;
   Aio.val = val;
   biactl(handle, AOUT, &Aio, sizeof(Aio));
}
```

utils.c

```
/*
 *This module contains utility code for BIA support.
 */
#include <sys/types.h>
#include "bia.h"
#include "ad.h"

/*

*****************************************************************************
 * Convert a physical address to logical address

*****************************************************************************
 */

unsigned int *create_pointer(phy_addr)
unsigned int phy_addr;
{
    phy_addr += 0xC0000000;
    return (unsigned int *) phy_addr;
}

/*

*****************************************************************************
 * Initialize the BIA board

*****************************************************************************
 */

bia_init()                      /* Initialize the BIA to known state */
{
   /*
    * Set up all the global pointers
    */
   bia_ctrl = (unsigned short *)create_pointer(bia_base_reg + CONTROL);
   bia_isr = (unsigned short *)create_pointer(bia_base_reg + ISR);
   bia_imr = (unsigned short *)create_pointer(bia_base_reg + IMR);
   bia_clr_isr = (unsigned short *)create_pointer(bia_base_reg + CLR_INT);
   bia_page_mod = (unsigned long *)create_pointer(bia_base_reg + PAG_MOD);
   bia_swap_reg = (unsigned long *)create_pointer(bia_base_reg + SWAP_REG);
```

```
    /*
     * Set up the BIA CONTROL register
     */
    bia_ctrl_reg = RESET_CTRL;/* Save CONTROL register (not readable) */
    *bia_ctrl = bia_ctrl_reg;

    /*
     * Set up the BIA interrupt mask register
     */
    *bia_imr = IMR_MASKALL;

    /*
     * Clear the BIA interrupt status register
     */
    *bia_clr_isr = 0;                                    /* Value is no care */

    /*
     * Clear the BIA page address register
     */
    if (bia_e_flg == FALSE)
        /*
         * Page address set to 0, modifier is A24 supervisory data access
         */
        *bia_page_mod = STD_SUP_DATA;
    else
        /*
         * Page address set to 0, modifier is A32 supervisory data access
         */
        *bia_page_mod = EXT_SUP_DATA;

#ifdef DEBUG
    printf("init bia: bia_ctrl_reg %x\n", bia_ctrl_reg);
#endif
}
```

```
/*
*******************************************************************************
 * Set the clear ISR register to 0; This will clear all latched interrupt
*******************************************************************************
 */
clear_bia_isr()
{
   *bia_clr_isr = 0;
}

/*
*******************************************************************************
 * Acknoledge the BIA interrupt
*******************************************************************************
 */
ack_bia_interrupt()
{
   bia_ctrl_reg |= VME_IACK;
   *bia_ctrl = bia_ctrl_reg;
#ifdef DEBUG
   printf("ack int: bia_ctrl_reg %x\n", bia_ctrl_reg);
#endif
}

/*
*******************************************************************************
 * Turn the ACK bit off of the BIA control register.
*******************************************************************************
 */
nak_bia_interrupt()
{
   bia_ctrl_reg &= ~VME_IACK;
   *bia_ctrl = bia_ctrl_reg;
#ifdef DEBUG
   printf("nak int: bia_ctrl_reg %x\n", bia_ctrl_reg);
#endif
}
```

```
/*
*************************************************************************
 * This will mask interrupts commig from the PBX to NX.
*************************************************************************
 */
disable_pbx_interrupt()
{
   out(SPIC_PORTB,(in(SPIC_PORTB) | ~(0xDF)));
}

/*
*************************************************************************
 * This will enable (unmask)  interrupts commig from the PBX to NX.
*************************************************************************
 */
enable_pbx_interrupt()
{
   /* Unmask the PIC for PBX interrupts */
   out(SPIC_PORTB,(in(SPIC_PORTB) & 0xDF));
}

/*
*************************************************************************
 * Enables The BIA to generate interrupts
*************************************************************************
 */
enable_bia_interrupt()
{
   *bia_imr = IMR_NO_MASK;
   bia_ctrl_reg |= INT_ENBL;     /* Enable PBX interrupt from BIA */
   *bia_ctrl = bia_ctrl_reg;
#ifdef DEBUG
   printf("enable_bia_int: bia_ctrl_reg %x\n", bia_ctrl_reg);
#endif
}
```

```
/*
*************************************************************************
 * disables The BIA interrupts
*************************************************************************
 */
disable_bia_interrupt()
{
   *bia_imr = IMR_MASKALL;
   bia_ctrl_reg &= ~INT_ENBL;     /* Disable PBX interrupt from BIA */
   *bia_ctrl = bia_ctrl_reg;
#ifdef DEBUG
   printf("disable_bia_int: bia_ctrl_reg %x\n", bia_ctrl_reg);
#endif
}
/*
*************************************************************************
 * Byte and address swap modes
*************************************************************************
 */
unsigned long set_array_comp(x)
unsigned long x;
{
   bia_ctrl_reg &= ~(SHFL1 | SHFL0);
   bia_ctrl_reg |= MODE_ARRAY;
   *bia_ctrl = bia_ctrl_reg;
   return(x);
}
unsigned long set_32_bit(x)
unsigned long x;
{
   bia_ctrl_reg &= ~(SHFL1 | SHFL0);
   bia_ctrl_reg |= MODE_32_BIT;
   *bia_ctrl = bia_ctrl_reg;
#ifdef DEBUG
   printf("set 32:bia_ctrl_reg %x\n", bia_ctrl_reg);
#endif
   return(x);
}
```

```
unsigned short set_16_bit(x)
unsigned short x;
{
   bia_ctrl_reg &= ~(SHFL1 | SHFL0);
   bia_ctrl_reg |= MODE_16_BIT;
   *bia_ctrl = bia_ctrl_reg;
#ifdef DEBUG
   printf("set 16: bia_ctrl_reg %x\n", bia_ctrl_reg);
#endif
   return (x);
}

unsigned char set_8_bit(x)
unsigned char x;
{
   bia_ctrl_reg &= ~(SHFL1 | SHFL0);
   bia_ctrl_reg |= MODE_8_BIT;
   *bia_ctrl = bia_ctrl_reg;
#ifdef DEBUG
   printf("set 8: bia_ctrl_reg %x\n", bia_ctrl_reg);
#endif
   return (x);
}

/*

*****************************************************************************
 * Enable LBX (PBX) addres mapping. Locations after 8 megabytes in NX
 * memory space are mapped to LBX space.

*****************************************************************************
 */

enable_lbx()
{
   out(NBCR_PORT, LBXENBL);
}
```

```
/*

*************************************************************************
 * This routine converts a VME physical address to NX/bia/VME
 * logical address.

*************************************************************************
 */

unsigned long make_dt_pointer(phaddr)
unsigned long phaddr;
{
   phaddr &= ~PAG_MASK;    /* Get rid of the page bits */
   phaddr |= vme_base_reg; /* Go to the bia VME address space */
   /* Move it into NX space */
   phaddr =  (unsigned long) create_pointer(phaddr);
   return (phaddr);
}

/*

*************************************************************************
 * This routine converts a physical address to NX/bia
 * logical address.

*************************************************************************
 */

unsigned long make_bia_pointer(phaddr)
unsigned long phaddr;
{
   phaddr = (unsigned long)create_pointer(bia_base_reg + phaddr);
   return (phaddr);
}
```

```
/*
*************************************************************************
 * Setup the BIA page mod register based on the physical address given
 * for the 4 megabyte VME address window.
*************************************************************************
 */

void setup_page_mod(phaddr)
unsigned long phaddr;
{
   *bia_page_mod &= MOD_MASK;                    /* Keep the modifier bits */
   *bia_page_mod |= phaddr & PAG_MASK;              /* Or in the page bits */
}

/*
*************************************************************************
 * Inialize the DT board
 * and registers
*************************************************************************
 */

init_board()
{
   /*
    *    Initialize the BIA board
    *    Select the STD Supervisory data access mode.
    */

   enable_lbx();
   bia_init();
   disable_bia_interrupt();
   set_16_bit();
   setup_page_mod(BASE);
```

```
    /*
     *    Initialize the AD board  register variables.
     */

    base    = (unsigned short *) make_dt_pointer(BASE);
    csr     = (unsigned short *) make_dt_pointer(CSR);
    adchan  = (unsigned short *) make_dt_pointer(ADCHAN);
    paclk   = (unsigned short *) make_dt_pointer(PACLK);
    addata  = (unsigned short *) make_dt_pointer(ADDATA);
    dac0    = (unsigned short *) make_dt_pointer(DAC0);
    dac1    = (unsigned short *) make_dt_pointer(DAC1);
    dioreg  = (unsigned short *) make_dt_pointer(DIOREG);

    /*
     *    Initialize the AD board.
     */

    *csr = CSR_INIT_CLR;
    enable_bia_interrupt();
}

/*

*****************************************************************************
 * Print the BIA interrupt

*****************************************************************************
 */

printirq()
{
    switch ((~(*bia_isr)) & (0x0E00 | ISR_ALL_IRQ)) {
        case ISR_IRQ1_:
             printf("IRQ1\n");
           break;
        case ISR_IRQ2_:
             printf("IRQ2\n");
           break;
        case ISR_IRQ3_:
             printf("IRQ3\n");
           break;
        case ISR_IRQ4_:
             printf("IRQ4\n");
           break;
        case ISR_IRQ5_:
             printf("IRQ5\n");
           break;
```

```
      case ISR_IRQ6_:
            printf("IRQ6\n");
         break;
      case ISR_IRQ7_:
            printf("IRQ7\n");
         break;
      case ISR_SYSFAIL_:
            printf("ISR_SYSFAIL\n");
         break;
      case ISR_BIA_TO_:
            printf("ISR_BIA_TO\n");
         break;
      case ISR_BERR_:
            printf("ISR_BERR\n");
         break;
      default:
         printf("multiple interrupt, ISR: 0x%04x\n", ~(*bia_isr) & 0xffff);
         break;
   }
}

/*

*****************************************************************************
 * Get VME vector

*****************************************************************************
 */

char get_VME_vec(base_addr, vec)
unsigned long base_addr;
unsigned short *vec;
{
short *b;
int int_no;
   switch ((~(*bia_isr)) & ISR_ALL_IRQ) {
      case ISR_IRQ1_:
         int_no = 1;
         break;
      case ISR_IRQ2_:
         int_no = 2;
         break;
      case ISR_IRQ3_:
         int_no = 3;
         break;
```

```
        case ISR_IRQ4_:
            int_no = 4;
            break;
        case ISR_IRQ5_:
            int_no = 5;
            break;
        case ISR_IRQ6_:
            int_no = 6;
            break;
        case ISR_IRQ7_:
            int_no = 7;
            break;
        default:
            fprintf("Invalid interrupt, ISR: 0x%04x", *bia_isr);
            *vec = 0;
            return(-1);
            break;
    }
    b = (short *)make_dt_pointer(base_addr + int_no*2);
    bia_ctrl_reg |= VME_IACK;
    *bia_ctrl = bia_ctrl_reg;
    *vec = *b;
    bia_ctrl_reg &= ~VME_IACK;
    *bia_ctrl = bia_ctrl_reg;
    return(int_no);

}
```