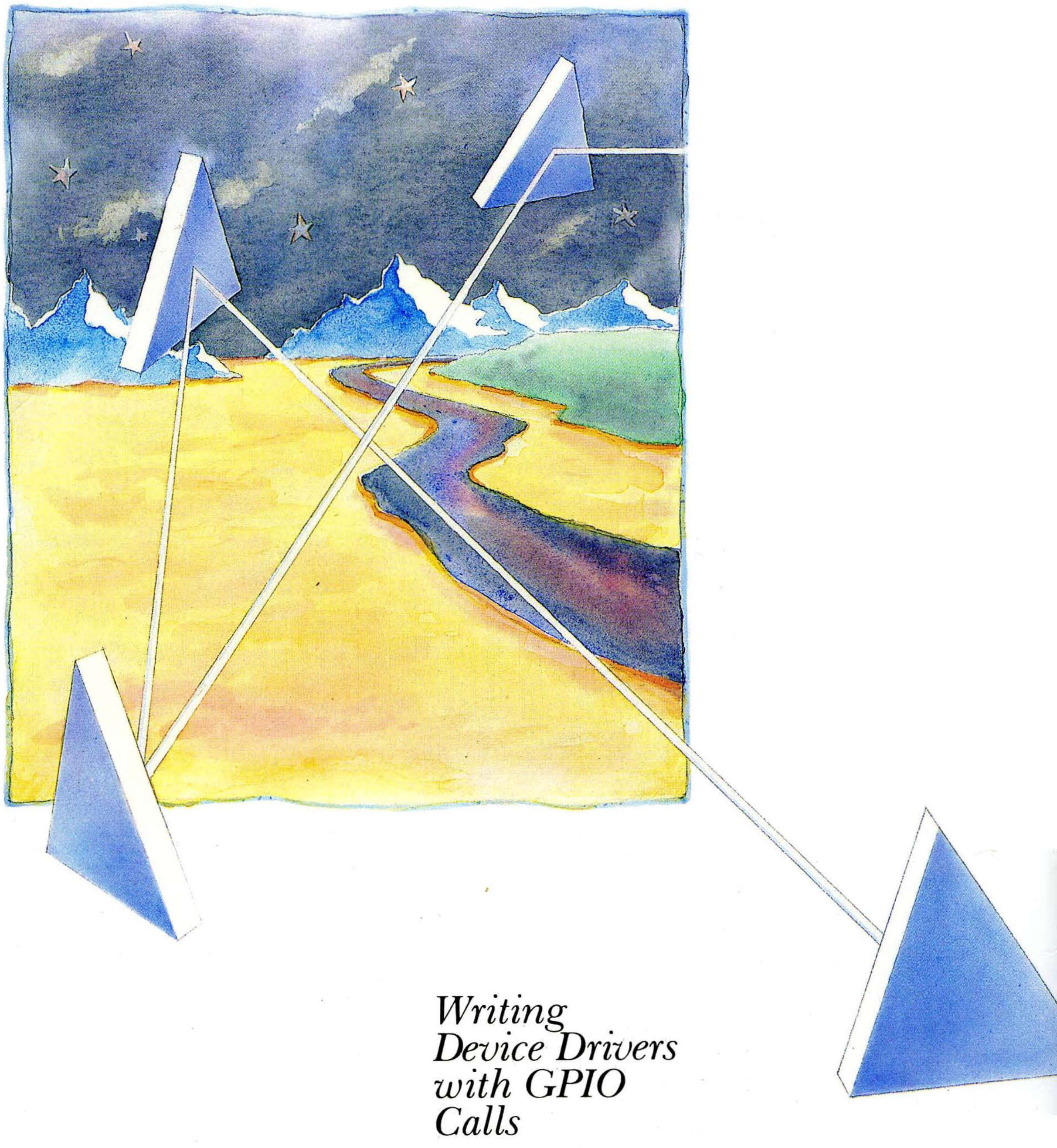


Writing Device Drivers with GPIO Calls

000959-A00

apollo

Writing Device Drivers with GPIO Calls

Order No. 000959-A00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: November 1981
Latest Printing: July 1988

This document was produced using the Interleaf Technical Publishing Software (TPS) and the InterCAP Illustrator I Technical Illustrating System, a product of InterCAP Graphics Systems Corporation. Interleaf and TPS are trademarks of Interleaf, Inc.

Preface

Writing Device Drivers with GPIO Calls describes how to write device drivers for Domain® nodes, using the General Purpose Input/Output (GPIO) software package.

Audience

This manual is intended for programmers who must write drivers for devices that Apollo® does not support. Readers of this manual should be familiar with the hardware of the I/O device and with its software requirements, and should have a working knowledge of Pascal or C.

Organization

We've organized this manual as follows:

Part 1	**I/O Hardware and Software**
Chapter 1	Describes the MULTIBUS* interface with Domain nodes, address translation between MULTIBUS memory and processor memory, and the rules for configuring MULTIBUS controllers.
Chapter 2	Describes the VMEbus and its interface with our system to help you to write drivers for VMEbus devices.

* MULTIBUS is a trademark of the Intel Corporation.

Chapter 3	Describes the PC AT* compatible bus and its interface with our system to help you to write drivers for PC AT devices.
Chapter 4	Provides an overview of the major components of I/O software (that is, the application, GPIO software, and the device driver).
Part 2	**Writing a Driver**
Chapter 5	Describes the different types of insert files that you can include in your driver and how to set them up.
Chapter 6	Describes the call side of the driver and how to write the routines that belong there.
Chapter 7	Describes how to transfer data using DMA, memory mapped I/O, and programmed I/O.
Chapter 8	Describes the interrupt side of the driver and different approaches to processing interrupts.
Chapter 9	Describes how to construct a global driver.
Chapter 10	Describes how to bind and debug the driver.
Chapter 11	Describes how to build the device descriptor file.
Chapter 12	Describes how to acquire and release the device.
Part 3	**Reference Information**
Appendix A	Describes the GPIO commands that the user invokes to run the driver.
Appendix B	Describes the calling format and parameters of the GPIO routines.
Appendix C	Provides some tips on setting up the CSR page and using data types in C.
Appendix D	Provides performance and timing information that relates to driver execution on our operating system.
Appendix E	Provides a program listing of a device driver coded in C.
Appendix F	Provides a program listing of a device driver coded in Pascal.

A glossary of terms appears at the back of the manual.

---

* PC AT is a registered trademark of International Business Machines Corporation.

Summary of Technical Changes

This manual has been revised for Software Release 10.

Related Manuals

The file **/install/doc/apollo/os.v.***latest software release number***__manuals** lists current titles and revisions for all available manuals.

For example, at SR10.0 refer to **/install/doc/apollo/os.v.10.0__manuals** to check that you are using the correct version of manuals. You may also want to use this file to check that you have ordered all of the manuals that you need.

(If you are using the Aegis environment, you can access the same information through the Help system by typing **help manuals.**)

Refer to the *Domain Documentation Quick Reference* (002685) and the *Domain Documentation Master Index* (011242) for a complete list of related documents. For more information on GPIO, refer to the following documents:

The *Aegis Command Reference* (002547) manual describes the command environment as well as the function and format of the commands that users can invoke.

The *DN5xx–T Workstations and DSP500–T Server Technical Reference* (009491) manual and the *DN5xx–T Workstations and DSP500–T Server Hardware Architecture Handbook* (009490) describe our implementation of the VMEbus.

The *Domain Binder and Librarian Reference* (004977) manual describes how to use the Domain binder to combine several object modules (for example, a call library and an interrupt library) into one executable object module.

The *Domain C Language Reference* (002093) and *Domain C Library (CLIB) Reference* (005805) manuals describe our implementation of the C language.

The *Domain Distributed Debugging Environment (Domain/DDE) Reference* (011024) manual describes how to use DDE.

The *Domain/OS Calls Reference* manuals, Volume 1 (007196) and Volume 2 (012886) describe the calling syntax for the system services that your driver can call.

The *Domain Pascal Language Reference* (000792) manual describes our implementation of the Pascal language. Appendix C lists our extensions to Standard Pascal.

The *Domain Personal Workstations and Servers Technical Reference* (008778) and the *Domain Personal Workstations and Servers Hardware Architecture Handbook* (007861) describe our implementation of the PC AT compatible bus.

The *IEEE Standard Microcomputer System Bus* (IEEE-796 specification) provides detailed information about the MULTIBUS.

The *Installing Input/Output (I/O) Devices for Domain Nodes* (008268) manual describes the hardware requirements for attaching peripheral devices to the Domain system bus.

The *Managing SysV System Software* (010851) manual describes how to create the SysV network environment, protect network software, and maintain and troubleshoot the network.

The *Microsystem Components Handbook* (230843) is published by Intel.

The *Programming with Domain/OS Calls* (005506) manual describes the general purpose Domain/OS system calls that you can use to perform system services for your driver.

The *Using the OPEN System Toolkit to Extend Your Domain Streams* (008863) manual describes how to extend the Streams facility so that it performs input and output for new types of files and devices.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. To make it easy for you to communicate with us, we provide the Apollo Problem Reporting (APR) system for comments related to hardware, software, and documentation. By using this formal channel, you make it easy for us to respond to your comments.

You can get more information about how to submit an APR by consulting the appropriate Command Reference manual for your environment (Aegis™, BSD, or SysV). Refer to the **mkapr** (make apollo problem report) shell command description. You can view the same description online by typing:

$ **man mkapr** (in the SysV environment)

% **man mkapr** (in the BSD environment)

$ **help mkapr** (in the Aegis environment)

Alternatively, you may use the Reader's Response Form at the back of this manual to submit comments about the manual.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions:

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Pathnames are also in bold. Bold words in text indicate the first use of a new term.
user–supplied values	Italic words or characters in formats and command descriptions represent values that you must supply.
example user input	In examples, information that the user enters appears in bold.
`output`	Information that the system displays appears in `this typeface`.
[ ]	Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square brackets assume their Pascal meanings.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions. In sample Pascal statements, braces assume their Pascal meanings.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/	The notation CTRL/ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the key.
. . .	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.
——— ⊞ ———	This symbol indicates the end of a chapter.

Contents

Figures

Tables

Chapter 1

I/O Bus Structures: the MULTIBUS

This chapter describes MULTIBUS implementations currently available for Domain nodes, the theory of MULTIBUS address translation, how to configure a MULTIBUS controller, and byte swapping. For detailed information about the MULTIBUS, refer to the *IEEE Standard Microcomputer System Bus* (IEEE–796 specification). See the Preface for a complete list of related manuals and their order numbers.

The I/O bus is the network of signal routes through which device controllers and the processor address one another and transfer data. The bus is, therefore, the key hardware component of a computer system's I/O structure. Figure 1–1 shows the relationship of the I/O bus to a Domain node and a set of controllers. The processor, memory, and memory management (address translation) subsystems are linked by an internal bus. Interface hardware connects this internal bus to the I/O bus. User-supplied and Domain system-supplied device controllers attach to the I/O bus and, through the bus, link to the node.

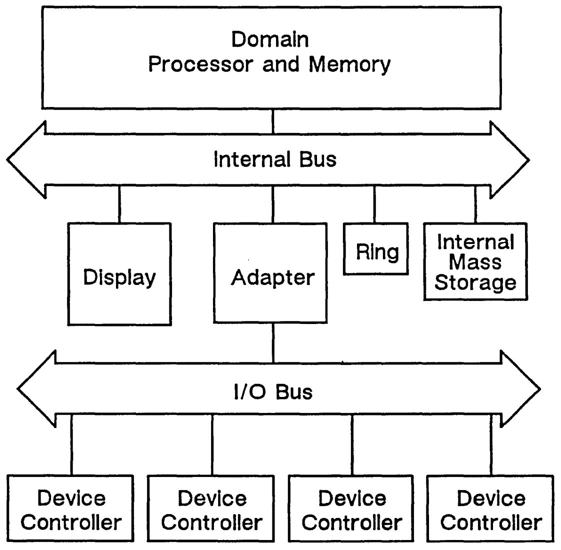


Figure 1–1. Relationship Between a Domain Node and Peripheral Controllers

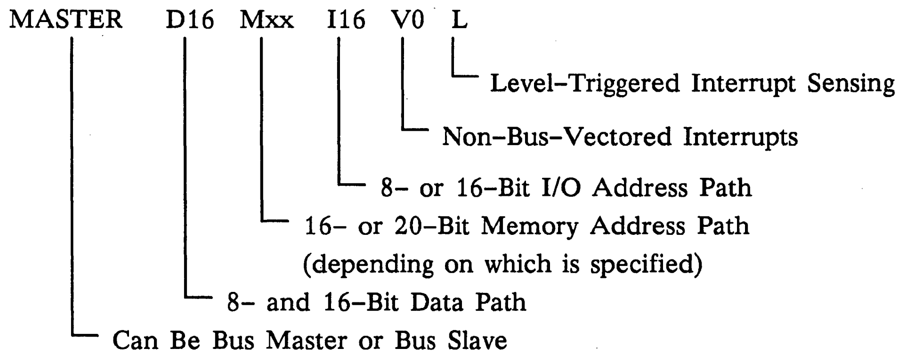
1.1 MULTIBUS Compliance Levels

The MULTIBUS supports compliance levels that allow for the varying capabilities of different computer systems. The levels are described in the *IEEE Standard Microcomputer System Bus* (IEEE–796 specification). To know the implementation available for a particular node model, refer to the section on MULTIBUS interfaces in the peripheral installation instructions or refer to the operating guide for the node model, if one is shipped with the node. If the peripheral installation instructions provide interface information for your node model, you will find the MULTIBUS implementation level available and specific hardware information for that node type. For node models that have an operating guide, you will find the same information in the guide. Table 1–1 lists the MULTIBUS implementation levels that we currently support for various node models.

Table 1–1. MULTIBUS Implementations on Node Models

Node Type	MULTIBUS Implementation	Compliance Level
DN660, DSP160	16–bit MULTIBUS, serial arbitration priority	MASTER D16 M16 I16 V0 L
DSP80 DSP90	20–bit MULTIBUS, parallel arbitration priority	MASTER D16 M20 I16 V0 L
DN5xx, DN5xx–T	20–bit MULTIBUS, serial arbitration priority	MASTER D16 M20 I16 V0 L

The notation used to specify the compliance level is interpreted as follows:


MASTER D16 Mxx I16 V0 L
Level–Triggered Interrupt Sensing
Non–Bus–Vectored Interrupts
8– or 16–Bit I/O Address Path
16– or 20–Bit Memory Address Path (depending on which is specified)
8– and 16–Bit Data Path
Can Be Bus Master or Bus Slave

The following sections explain the compliance levels more fully, particularly the two levels that we currently support:

- MASTER D16 M16 I16 V0 L
- MASTER D16 M20 I16 V0 L

1.1.1 Bus Control

A device controller is bus master when it acquires control of the bus, and bus slave when it carries out commands or decodes addresses presented by another device acting as bus master. Domain nodes with 16-bit MULTIBUS implementation allow both the central processor and any attached controller to act as bus masters:

- When the processor is bus master, it can address 32 KB of MULTIBUS I/O space and 32 KB of MULTIBUS memory space (0–7FFF).
- When a controller is bus master, the processor must be the only slave; it responds to addresses in the range 0–FFFF (64K).

Domain nodes with 20-bit MULTIBUS implementations also allow either the processor or the controllers to act as bus masters:

- When the processor is bus master, it can address 64 KB of MULTIBUS I/O space and 1 MB of MULTIBUS memory space.
- When a controller is bus master, either the processor or another controller on the MULTIBUS may be the slave; up to 1 MB of address space is available.

NOTE: Although the full 64 KB of I/O address space is implemented on nodes with a 20-bit MULTIBUS, user Control and Status Register (CSR) page addresses are restricted to the first 16 KB of MULTIBUS I/O space (see Subsection 1.3.2).

1.1.2 Data Path

For all Domain nodes, the MULTIBUS supports either an 8- or a 16-bit bidirectional data path (D16) for the transfer of data from MULTIBUS memory or I/O addresses. The bus master drives the data lines on a write operation, and the slave drives them on a read operation (memory or I/O).

1.1.3 Memory Address Path

Under compliance level MASTER D16 M16 I16 V0 L, the MULTIBUS supports 16-bit memory addresses on the memory address path; whereas under compliance level MASTER D16 M20 I16 V0 L, the MULTIBUS supports 20-bit memory addresses. We use the terms *16-bit MULTIBUS* or *20-bit MULTIBUS* to describe nodes whose I/O hardware supports 16- or 20-bit memory addresses.

NOTE: If a node with a 20-bit MULTIBUS is fully configured with 3 MB of memory, the upper half (512 KB) of the address space is unavailable for memory-mapped operations.

1.1.4 I/O Address Path

For all Domain nodes, the MULTIBUS I/O address path supports 8-bit or 16-bit I/O addresses (I16).

1.1.5 Interrupt Request Lines

The MULTIBUS provides eight interrupt request lines: line 0 is the highest priority line and line 7 the lowest. A device generates an interrupt by activating its assigned interrupt request line. The MULTIBUS on all Domain nodes uses nonbus-vectored interrupts (V0). With this type of interrupt, the device raises its interrupt line without sending its interrupt vector address over the bus; the I/O hardware generates the interrupt vector address to identify the interrupting device to the processor.

1.1.6 Bus Request Arbitration Resolution

MULTIBUS devices can arbitrate for bus control by using serial or parallel priority resolution. All Domain 16-bit MULTIBUS implementations use a serial scheme. Some 20-bit implementations use a parallel scheme and others use a serial scheme. See the peripheral installation instructions for the priority resolution scheme used by each node type.

With serial resolution, device controllers are daisy-chained together. The first device in the daisy-chain has highest priority. With parallel resolution, arbitration logic in the I/O hardware determines the device that gets highest priority, instead of the device's position relative to other controllers. See the node's operating guide or peripheral installation instructions for the priority assignments supplied by our I/O hardware for nodes that use parallel arbitration resolution.

1.2 MULTIBUS Address Translation

Device drivers on nodes with a 16-bit MULTIBUS can allocate up to 32 pages of processor address space to reference MULTIBUS address space; drivers on nodes with a 20-bit MULTIBUS can allocate up to 1024 pages of processor address space. On any node, the I/O hardware translates addresses between MULTIBUS and processor memory in units of 1024-byte pages. The method of translation depends upon whether processor addresses are to be translated into MULTIBUS addresses (initiated by the processor) or MULTIBUS addresses into processor addresses (initiated by the controller).

1.2.1 Address Translation from Processor to MULTIBUS

When the processor acts as bus master, it initiates a read or write to MULTIBUS address space, and the I/O hardware automatically translates the virtual address that refers to processor address space into a physical address.

This physical address refers to either one of two separate address spaces supported by the MULTIBUS, depending on the kind of I/O operation:

- I/O space: Used for programmed I/O data transfers
- Memory space: Used for memory–mapped data transfers

Much of what follows concerning processor–to–MULTIBUS address translation depends on this concept of two separate MULTIBUS address spaces.

1.2.1.1 Programmed I/O

In programmed I/O, data is transferred as single words or bytes by means of Control and Status Registers (CSRs) on the controller. Device drivers pass or reference data by using these CSRs.

References to the MULTIBUS I/O space are actually references to a controller's CSRs. A page from MULTIBUS I/O space is allocated to them and becomes the controller's CSR page. Section 1.3 describes how to allocate pages of MULTIBUS I/O space for controller CSRs.

When the device is acquired, the GPIO device acquisition routine, **pbu_$acquire**, automatically maps the CSR page to processor address space (that is, establishes a correspondence between MULTIBUS I/O space and processor address space) and passes a pointer to the driver initialization routine. The device driver can then obtain controller status and activate the controller by using the pointer to read and write to the mapped CSRs. Figure 1–2 shows how CSR pages mapped to processor address space correspond to MULTIBUS I/O locations.

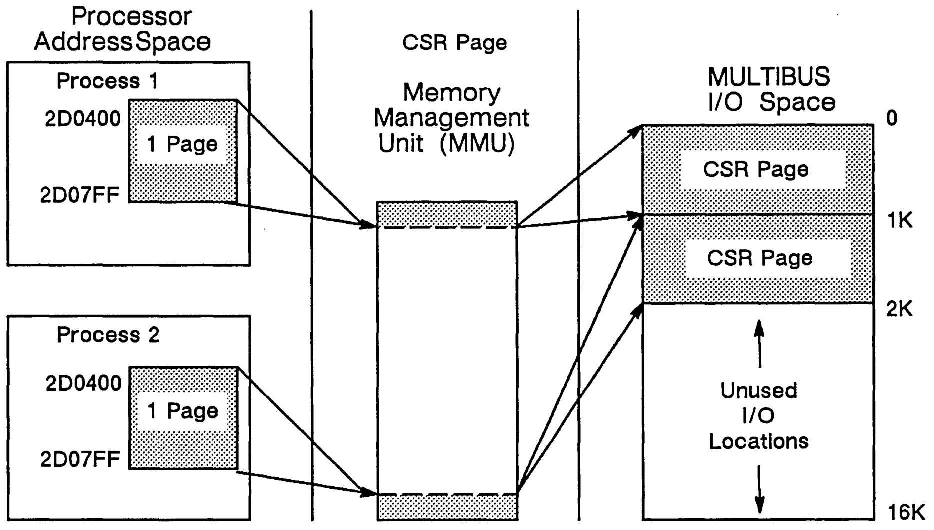


Figure 1–2. Mapping CSR Pages to MULTIBUS I/O Space

1.2.1.2 Memory–Mapped I/O

In memory–mapped I/O, the controller appears to the processor as so many memory locations, and the processor performs I/O operations by storing data to or fetching it from controller memory.

Device drivers gain access to areas of MULTIBUS memory space by calling GPIO routines. These routines map areas of processor address space and particular sections of MULTIBUS memory space. Device drivers next call the GPIO routines that map a controller's memory to processor address space. The drivers can then read and write to controller memory through reads and writes in processor address space. Figure 1–3 illustrates how controller memory is mapped to processor address space.

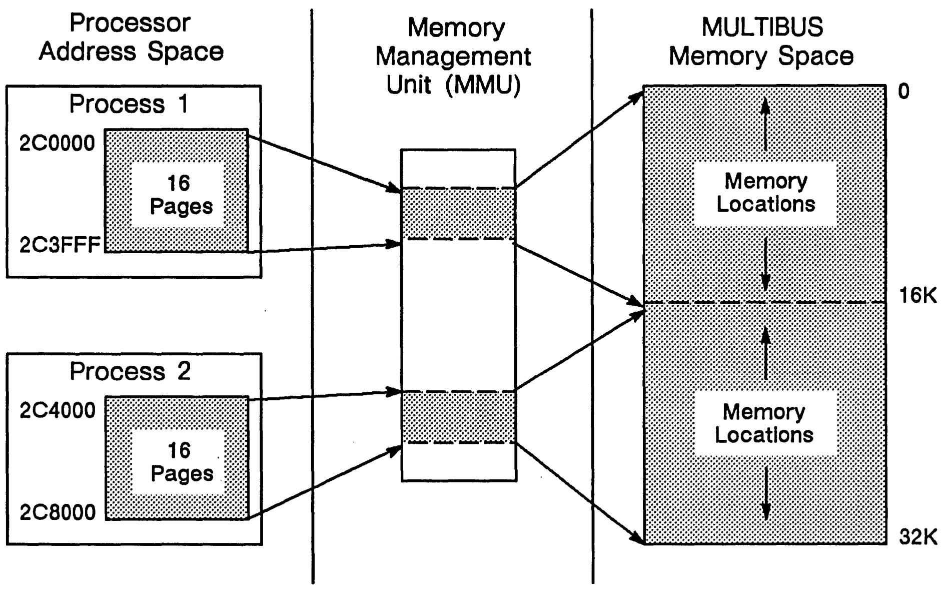


Figure 1–3. Mapping Processor Address Space to MULTIBUS Memory Space

1.2.2 Address Translation from MULTIBUS to Processor: DMA

A Direct Memory Access (DMA) operation contrasts with programmed I/O and memory mapping in the following ways:

- The controller is the bus master.
- Address translation proceeds from the MULTIBUS to the processor.
- A bus address (referred to as an *iova*) is translated into a physical address in processor memory.

Once activated by its device driver, a DMA controller can transfer large amounts of data directly between processor memory and MULTIBUS address space. The job of translating references to MULTIBUS address space into references to processor address space is performed by a data structure called the I/O map. The I/O map contains entries that each map one page of processor memory. The device driver calls GPIO routines to allocate I/O map entries for the DMA. Chapter 7, Section 7.1 describes these GPIO routines in more detail.

For nodes with a 16-bit MULTIBUS, controllers can transfer up to 64 pages of data between the MULTIBUS and the processor at one time. For nodes with a 20-bit MULTIBUS, controllers can transfer up to 1024 pages at one time. Figure 1-4 illustrates a DMA transfer of 64 pages of MULTIBUS address space to two different areas of processor address space.

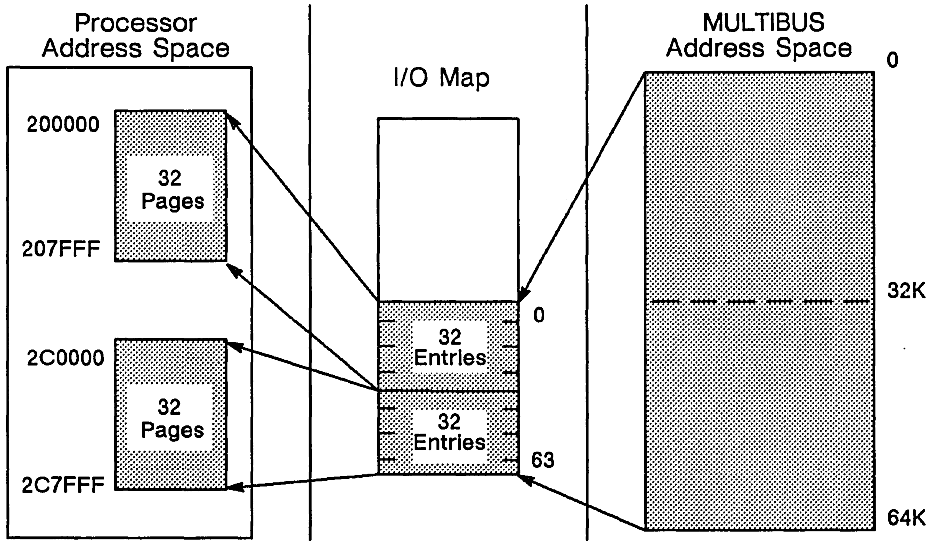


Figure 1-4. Mapping MULTIBUS Address Space to Processor Address Space

1.3 Configuring MULTIBUS Controllers

When you supply your own MULTIBUS controllers for use with a Domain node, you must observe basic configuration rules. The following subsections summarize controller configuration rules for nodes with a 16- or 20-bit MULTIBUS. Table 1-2 lists the address ranges reserved for Domain system-supplied devices.

Table 1-2. MULTIBUS Address Space Used by Domain System-Supplied Devices

Devices	**Addresses Used**
Domain/ComController™	Memory pages 4000 to 7F00 and I/O page 0800 are always in use on a 16-bit MULTIBUS.
ETHERNET* Interlan Board	Uses three dynamically allocated memory pages for DMA I/O address space 080-08F every 256 bytes (180-18F, 280-28F, 380-38F, etc.).
FSD-500	Memory pages F400 and F800 on a 16-bit MULTIBUS or memory pages 6F400 and 6F800 on a 20-bit MULTIBUS are used by the mnemonic debugger, then released during operating initialization. The operating system uses two dynamically allocated memory pages for DMA.
Magtape	Uses 19 dynamically allocated memory pages for DMA, plus memory page FC00 (used during initialization, then released).
Storage Module Device (SMD)	Memory pages F400 and F800 on a 16-bit MULTIBUS or memory pages 6F400 and 6F800 on a 20-bit MULTIBUS are always reserved, whether or not SMD is in use.
VERSATEC** and IMAGEN*** Printers	Uses five dynamically allocated memory pages for DMA; I/O page 400 reserved.
X.25	Pages 7000-7C00 are always in use.

* ETHERNET is a registered trademark of the Xerox Corporation.
** VERSATEC is a registered trademark of VERSATEC, Inc.
*** IMAGEN is a registered trademark of the IMAGEN Corporation.

1.3.1 Nodes With a 16-Bit MULTIBUS

You can connect only one 8-bit controller to a 16-bit MULTIBUS; the others must be 16-bit controllers.

1.3.1.1 Assigning CSR Addresses

Each controller is allocated one page of MULTIBUS I/O space for its set of CSR addresses. MULTIBUS I/O space is divided into two 16-page sections. The lower 16-page section is reserved for the CSR pages of user-supplied controllers; the top 16-page section is reserved for the CSR pages of controllers that Apollo supplies. You can assign the CSR addresses of a 16-bit controller to any page within the 16 pages of MULTIBUS I/O space (0-3FFF hex) allocated to user-supplied controllers. Word (2-byte) and longword (4-byte) registers must reside on even-byte addresses.

If an 8-bit controller is present on your system, its CSR addresses should fall between 80 and FF (hex) on the first page (page 0) of the allocated I/O address space. Of the remaining pages (1-15), 16-bit controllers must occupy only the first 128 bytes (0-7F) of each page. This arrangement is necessary because 8-bit controllers respond to any address in the range 0-FF, modulo 255. For example, an 8-bit controller CSR at address 80 responds to page 0 addresses of 80, 180, 280, 380; page 1 addresses of 480, 580, 680, 780; and so on. By restricting 8-bit controller CSRs to the range 80-FF, all addresses in the range 0-7F become available to 16-bit controllers. Refer to Chapter 11, Section 11.2 for a description of how to set the address of an 8-bit controller CSR.

If you do not have an 8-bit controller on your system and never plan to add one, you can configure a 16-bit controller to respond to any addresses (0-3FF) on its CSR page. Again, word and longword registers must reside on even-byte addresses.

Figure 1-5 illustrates the allocation of CSR addresses when an 8-bit controller is present.

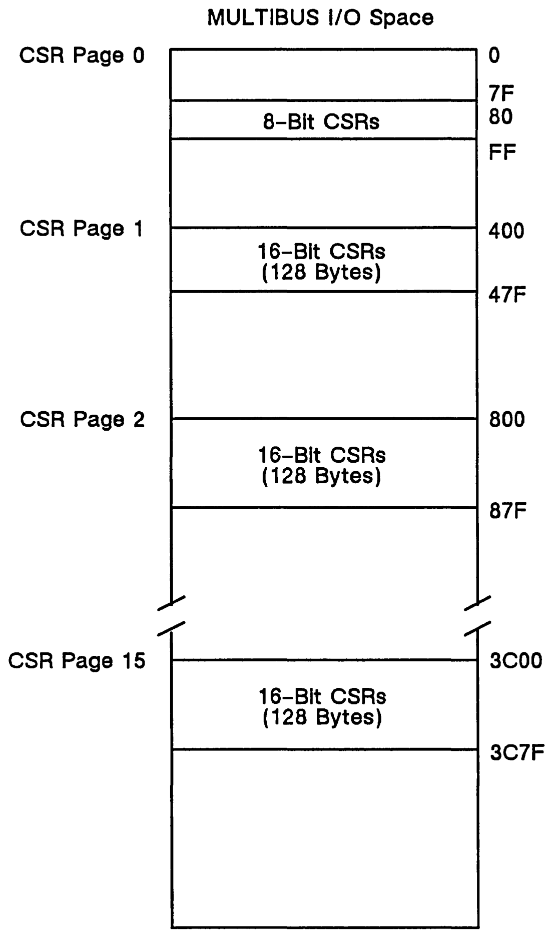


Figure 1-5. 8-Bit Controller CSR Assignment

1.3.1.2 Configuring Controller Memory

Drivers call GPIO routines to map a controller's memory to processor address space so that programs can refer to the controller's memory directly. When configuring controller memory on nodes with a 16-bit MULTIBUS, the following rules apply:

- Controller memory must begin on a page boundary and must reside completely in the first 32 KB (0–7FFF) of MULTIBUS memory space.
- Because of hardware restrictions, the part of the MULTIBUS memory space occupied by controller memory is permanently unavailable for DMA to or from any controller on the bus.
- Programs can access controller memory through the MULTIBUS, but other controllers on the bus cannot do so (see Chapter 7, Subsection 7.2.1).

1.3.1.3 Configuring Controller Address Lines

On a node with a 16-bit MULTIBUS, up to 64 pages of MULTIBUS address space can be mapped (through the I/O map) to processor memory. Controller references to MULTIBUS addresses above 64K are wrapped; the top four bits of addresses on the bus are driven to 0. For example, a controller reference to 65K appears as a reference to 1K. Consequently, when you have the choice of configuring a controller to a 16-bit or a 20-bit address path, configure for a 16-bit address path.

1.3.1.4 Using Interrupt Request Lines

Of the eight interrupt request lines available on the MULTIBUS, the highest priority line (line 0) is reserved for customer devices. The remaining seven interrupt lines are reserved for devices that we supply. Table 1–3 lists the allocation of bus interrupt request lines.

Table 1–3. Allocation of Interrupt Request Lines

Line	Owner
0	Customer devices
1	COM–ETH product controller
2	COM–X.25 product controller and Domain/ComController product
3	Magtape controller
4	Storage module or FSD–500 product controller
5	VERSATEC printer/plotter controller and IMAGEN printer with MULTIBUS option
6	Parallel output/line printer (only on 16-bit MULTIBUS; unused on 20-bit MULTIBUS)
7	Reserved

Because line 6 is used for parallel I/O, it is unavailable for your use. Lines 1 through 5, though reserved for our use, are available to user–supplied controllers. However, if you assign your device to one of lines 1 through 5 and later acquire one of our supported devices assigned to that line, conflicts will result. Line 0 is reserved for customer devices and will never be used by Domain devices.

A single controller can be configured to request interrupts on more than one request line, but each line can handle only one controller.

On nodes with a 16–bit MULTIBUS, the processor is solely responsible for acknowledging peripheral device interrupt requests. Device controllers should never respond to interrupt requests from other peripheral devices on the bus.

1.3.2 Nodes With a 20–Bit MULTIBUS

Nodes with 20–bit MULTIBUS implementations can also handle 8–bit or 16–bit controllers. Of the devices that can be attached to such nodes, only one can be an 8–bit controller; the others must be 16–bit controllers.

1.3.2.1 Assigning CSR Addresses

On nodes with a 20–bit MULTIBUS, 64 pages of MULTIBUS I/O space are available; however, user devices are restricted to the first 16 pages because Domain system–supplied devices occupy the second 16 pages and addresses 8000–FFFF are reserved for future use. Each controller is allocated one page of the first 16 pages of I/O address space for its set of CSRs (if any). You can assign the addresses of a 16–bit controller to any page within the first 16 pages (0–3FFF hex). Word (2–byte) and longword (4–byte) registers must reside on even–byte addresses. If an 8–bit controller is present in your configuration, assign its CSRs according to the rules outlined in Subsection 1.3.1.

1.3.2.2 Configuring Controller Memory

If a node with a 20–bit MULTIBUS is fully configured with 3 MB of memory, the upper half (512 KB) of the address space is available for DMA operations only. Also, if your configuration includes both 16–bit and 20–bit memory–mapped controllers, you must use caution when configuring 20–bit controller memory into MULTIBUS memory space to avoid possible conflicts with 16–bit controller memory. For example, a 16–bit controller configured to respond to memory address C000 will also respond to addresses 1C000, 2C000, ... FC000. In this case, you must ensure that the MULTIBUS addresses assigned to the 20–bit controller do not equal C000, modulo 64K.

1.3.2.3 Configuring Controller Address Lines

Nodes with a 20-bit MULTIBUS implementation can map up to 1024 pages of MULTIBUS address space through the I/O map to processor memory. As in 16-bit MULTIBUS systems, controller references to MULTIBUS addresses above 1 MB are wrapped. Consequently, when you have the choice of configuring a controller to a 24-bit or a 20-bit address path, configure for a 20-bit address path.

1.3.2.4 Using Interrupt Request Lines

Nodes with a 20-bit MULTIBUS allocate interrupt request lines in the same way as nodes with a 16-bit MULTIBUS, except that lines 6 and 7 are also available (although they are reserved for Domain system-supplied devices). Again, the processor is solely responsible for acknowledging peripheral device interrupt requests; device controllers should never respond to interrupt requests from other peripheral devices on the bus. Table 1-3 lists the allocation of bus interrupt lines.

1.4 Byte Swapping

The necessity for byte swapping (exchanging the left and right bytes of a word) arises from the fact that the Domain processor, which is based on the Motorola 68000 family, orders bytes within a word the opposite of the way Intel processors order them on MULTIBUS controllers.

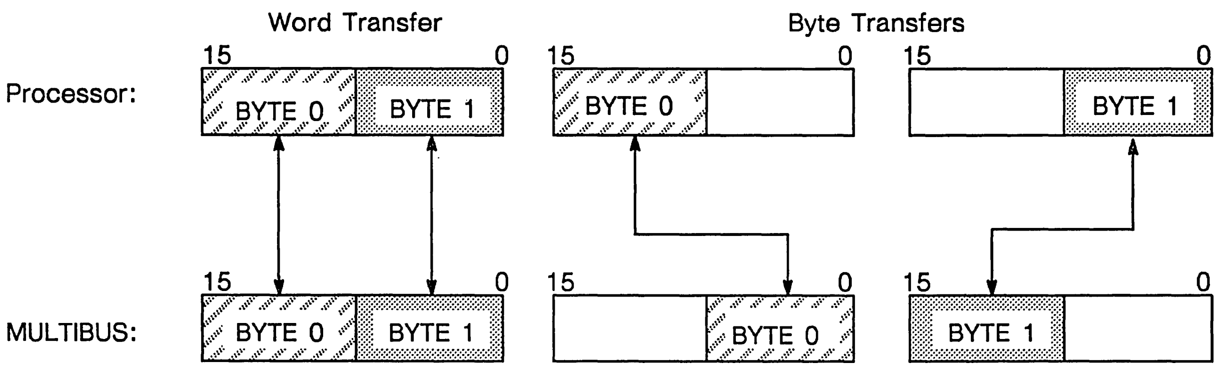
This is how our processor does it:

15 — 8	7 — 0
BYTE 0	BYTE 1

This is how MULTIBUS does it:

15 — 8	7 — 0
BYTE 1	BYTE 0

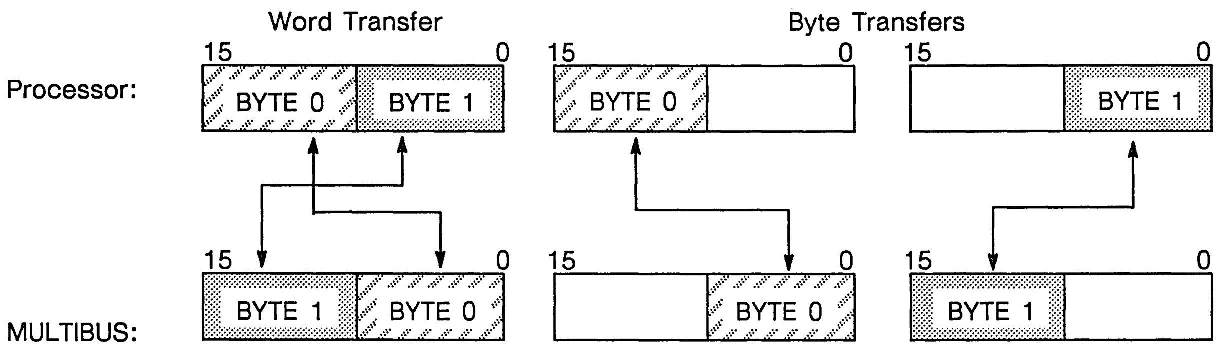
We deal with this incompatibility by swapping bytes in hardware during byte transfers. Effectively, character strings copied as bytes and integers copied as words are preserved, but character strings copied as words (and words copied as bytes) are byte swapped. The following illustrates this strategy:



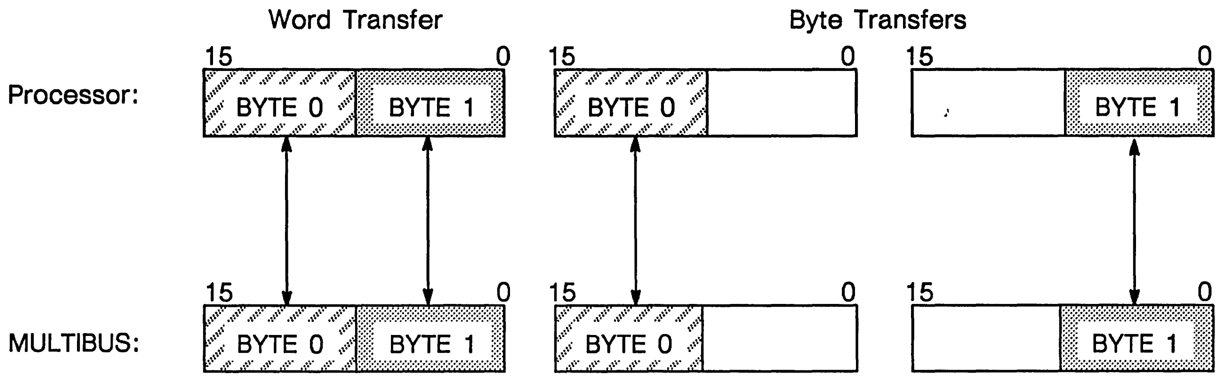
Note that this strategy uses the following byte/word arrangements:

- Pointers to words must be even.
- Pointers to processor left bytes (byte 0) must be even.
- Pointers to processor right bytes (byte 1) must be odd.

The GPIO call **pbu_$control** is available for 20-bit MULTIBUS implementations (refer to Appendix B for a description of the call). This call gives you control over the byte-swapping hardware so that you can specify other byte/word arrangements than those just spelled out (the **pbu_swap_off** option gives you the arrangement described previously). By specifying the **pbu_swap_words** option with this call, you ensure that all character strings have their byte order preserved regardless of whether they are copied as words or bytes and that integers are always byte swapped. The following illustrates byte swapping when **pbu_swap_words** is specified:



By specifying the **pbu_swap_bytes** option with the **pbu_$control** call, you ensure that integers have their byte order preserved regardless of whether they are copied as words or bytes and that character strings are always byte swapped. The following illustrates byte swapping when **pbu_swap_bytes** is specified:



It should be noted that single byte transfers always occur on MULTIBUS data lines 0 through 7 and that word transfers use all 16 data lines.

Chapter 2

I/O Bus Structures: the VMEbus

This chapter presents information you need to know about the VMEbus in order to use GPIO software to write device drivers for VMEbus devices, specifically, address space allocation. grant levels, use of address modifiers, interrupt levels, and software considerations. For additional information about the VMEbus, refer to the *DN5xx-T Workstations and DSP500-T Server Technical Reference* manual and the *Motorola VMEbus Specification Manual,* Rev. C.1 or IEEE P1014/D1.2.

The I/O bus is the network of signal routes through which device controllers and the processor address one another and transfer data. The bus is, therefore, the key hardware component of a computer system's I/O structure. Figure 2-1 shows the relationship of the I/O bus to a Domain node and a set of controllers. The processor, memory, and memory management (address translation) subsystems are linked by an internal bus. Interface hardware connects this internal bus to the I/O bus. User-supplied and Domain system-supplied device controllers attach to the I/O bus and, through the bus, link to the node.

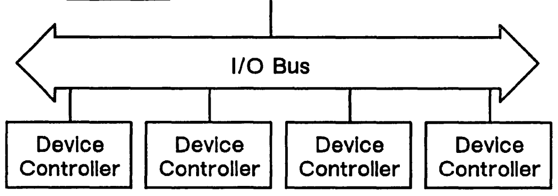


Figure 2-1. Relationship Between a Domain Node and Peripheral Controllers

2.1 Address Space Allocation

Because there is no mapping mechanism between the VMEbus and a customer VMEbus device, there must be agreement as to what VMEbus addresses are reserved for your controllers. In addition, you must be aware that as our allocation of the physical address space on existing and future workstations changes, it may be necessary for you to modify your controllers to respond to different addresses on different workstations.

The address layout for the DN5xx–T is listed in Table 2–1.

Table 2–1. Address Space Allocated for DN5xx–T VMEbus Devices

Physical Addresses	Resource	Address/Data Lines
0000–7FFF	VMEbus CSRs	16–Bit Addressing, 16–Bit Data Path
C000–DFFF	VMEbus CSRs	24–Bit Addressing, 16–Bit Data Path
80000–FFFFF	User VMEbus	24–Bit Addressing, 16–Bit Data Path
200000–2FFFFF	User VMEbus	24–Bit Addressing, 16–Bit Data Path
310000–3FFFFF	User VMEbus	24–Bit Addressing, 16–Bit Data Path
600000–7FFFFF	User VMEbus	24–Bit Addressing, 32–Bit Data Path
800000–FFFFFF	User VMEbus*	24–Bit Addressing, 32–Bit Data Path
3000000–300FFFF	VMEbus I/O (CSRs)	32–Bit Addressing, 32–Bit Data Path
3200000–3FFFFFF	User VMEbus	32–Bit Addressing, 32–Bit Data Path
*Available only on DN570–T workstation.		

2.2 Bus Grant Level

VMEbus devices should use bus grant level 2.

2.3 Address Modifiers

The current DN5xx–T VMEbus interface defines the following address modifiers for all references to VMEbus controllers:

- 2D: 16–Bit Addressing
- 3D: 24–Bit Addressing
- 0D: 32–Bit Addressing

Domain system–supplied controllers also use these address modifiers for DMA activity.

Apollo recommends that the address modifiers that a device uses be held in two program-loadable registers, one for slave responses and the other for master requests. In the initial power-on/reset state of the device, it should be possible to load these registers by using any address modifier.

2.4 Interrupt Level

Customer VMEbus devices are currently assigned to VMEbus interrupt level 5. The VMEbus interrupt level used by a customer device should be jumperable to allow for possible changes in interrupt level allocation on future workstations.

2.5 Status/ID Byte

A VMEbus controller presents a status/ID byte during a VMEbus interrupt acknowledge cycle. The operating system uses this byte to distinguish between multiple VMEbus devices and by GPIO as the unit number identifying the device. Status/ID bytes F8 through FE (corresponding to unit numbers 8 through 14) are available for customer devices; status/ID bytes F0 through F7 and FF are reserved.

The Device Descriptor File (DDF) for a VMEbus device defines the bottom nibble of the status/ID as the device unit number.

2.6 Software Considerations

GPIO software supports memory-mapped I/O, programmed I/O, and DMA operations on the VMEbus.

> **NOTE:** Apollo provides two kinds of calls, **pbu_$** and **pbu2_$**, for several GPIO operations. When referring to either kind interchangeably, we use the term **pbu[2]_$***routine_name*.

There is no DMA address translation hardware (I/O map) for the VMEbus; the following GPIO calls are, therefore, not applicable to drivers that support VMEbus devices:

- **pbu[2]_$allocate_map**
- **pbu[2]_$free_map**
- **pbu[2]_$map**
- **pbu[2]_$unmap**

In addition, the following GPIO calls are not applicable to VMEbus devices and cannot be used in drivers for VMEbus devices:

- **pbu_$device_interrupting**
- **pbu_$disable_device**
- **pbu_$enable_device**
- **pbu_$control**
- **pbu[2]_$dma_start**
- **pbu[2]_$dma_stop**

Otherwise, you use GPIO software when writing drivers for VMEbus devices just as you would for MULTIBUS devices. Extensions to the GPIO package to accommodate the VMEbus in no way limit the current facilities of GPIO.

2.6.1 Wiring for DMA: pbu_$wire_special

Since there is no mapping hardware between the customer's device and the VMEbus, device drivers should call **pbu_$wire_special** (instead of **pbu[2]_$wire**) to wire buffers for DMA operations. This call returns a list of physical (VMEbus) addresses at which the buffer is located. The customer's driver or controller hardware uses the addresses to perform the necessary scatter–gather operations. Refer to Appendix B for a full description of this call.

2.6.2 Creating a DDF for a VMEbus Device

To create a DDF for a VMEbus device, you must specify the **–vme** option with the **crddf** command. This option indicates to GPIO that the device in question resides on the VMEbus. It is recommended that this option be the first specified when building a new DDF. Valid unit numbers when the **–vme** option is specified are in the range 8 to 14 (pbu_$min_vme_unit to pbu_$max_vme_unit).

If the **–vme** option is specified, the specification of a CSR page is optional. If a CSR page is specified, it must be page–aligned and in the range 0000–7C00 (A16) or C000–DC00 (A24).

Refer to Appendix A for a full description of the **crddf** command and the **–vme** option and to Chapter 11, Subsection 11.3.2 for an example of the **crddf** command with the **–vme** option.

Chapter 3

I/O Bus Structures: the IBM PC AT Compatible Bus

This chapter presents information about the IBM PC AT compatible bus in order to use GPIO software to write device drivers for PC AT compatible devices, specifically: I/O address and memory space allocation, unit numbering, testing for device presence, DMA and interrupt lines, byte swapping, and software considerations. For additional information about the PC AT compatible bus, refer to the *Domain Personal Workstations and Servers Technical Reference* manual.

NOTE: Apollo provides two kinds of calls, **pbu_$** and **pbu2_$**, for several GPIO operations. When referring to either kind interchangeably, we use the term **pbu[2]_$***routine_name*.

The I/O bus is the network of signal routes through which device controllers and the processor address one another and transfer data. The bus is the key hardware component of a computer system's I/O structure. Figure 3-1 shows the relationship of the I/O bus to a Domain node and a set of controllers. The processor, memory, and memory management (address translation) subsystems are linked by an internal bus. Interface hardware connects this internal bus to the I/O bus. User-supplied and Domain system-supplied device controllers attach to the I/O bus and, through the bus, link to the node.

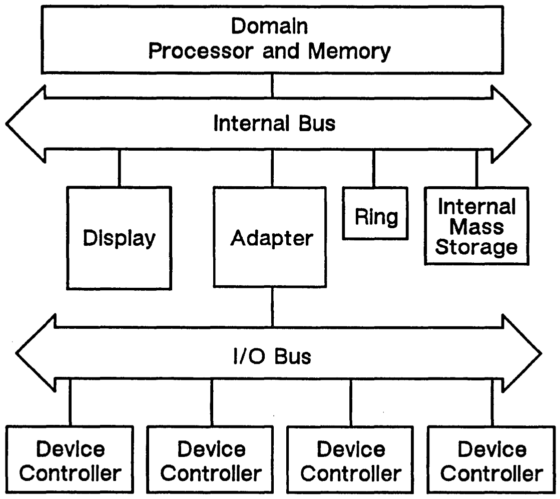


Figure 3-1. Relationship Between a Domain Node and Peripheral Controllers

3.1 PC AT Compatible Address Space

The physical address space on the PC AT compatible bus that is available to the user consists of I/O address space (reserved for device CSRs), and memory address space (reserved for memory-mapped controllers). The following subsections describe these two address spaces in detail. For additional information on the PC AT compatible bus address space, refer to the *Domain Personal Workstations and Servers Hardware Architecture Handbook.*

3.1.1 I/O Address Space

The I/O address space (0-3FF) is reserved for device CSRs. Table 3-1 lists the address ranges within this area that are reserved for Domain system-supplied devices and those that are available for customer devices. If your system is not configured with the system-supplied device that occupies a particular address range, then you may use that range for your own device.

Table 3-1. I/O Address Space Allocated for Domain System-Supplied Devices

Bus Address (Hex)	**Device**
000–0FF	Reserved
100–19F	Customer Devices
1A0–1A7	Disk Controller
1A8–1FF	Customer Devices
200–207	Tape Controller
208–21F	Customer Devices
220–23F	Apollo Token Ring Network Controller–AT
240–2F7	Customer Devices
2F8–2FF	Serial–Parallel Expansion (SPE) option — Serial Line 2
300–307	802.3 Network Controller–AT
310–317	802.3 Network Controller–AT (Alternate)
320–33F	Apollo Token Ring Network Controller–AT
340–377	Customer Devices
378–37F	SPE option — Parallel Port
380–3AF	Customer Devices
3B0–3BF	Monochrome Graphics (Alternate Color)
3C0–3CF	Customer Devices
3D0–3DF	Color Graphics (Alternate Monochrome)
3E0–3EF	Customer Devices
3F0–3F7	Disk Controller
3F8–3FF	SPE — Serial Line 1

To provide protection for system devices and virtual memory support, addresses in the PC AT compatible I/O address space are mapped differently from addresses in MULTIBUS and VMEbus address spaces. Ten-bit consecutive addresses in the I/O address space are mapped into processor address space in groups of eight bytes, and each group is assigned the first eight bytes of a different, but consecutive, page (1024 bytes). Thus, the first 1024 addresses in PC AT compatible address space (0–3FF) map to 128 physical pages (40000–5FFFF) in processor address space.

A PC AT compatible controller using three 8-byte CSR addresses might have the following type declaration:

```
typedef struct csr_page_t {
    char first_eight[7];
    char next_eight[7];
    char last_eight[7];
} csr_page_t #attribute[device];
```

In Apollo systems, however, the type declaration should be as follows:

```
typedef struct csr_page_t {
    char first_eight[8];
    char pad1[bytes_per_page-8];
    char next_eight[8];
    char pad2[bytes_per_page-8];
    char last_eight[8];
    char pad3[bytes_per_page-8];
} csr_page_t #attribute[device];
```

Figure 3-2 illustrates the mapping scheme for the preceding example (csr_ptr is the pointer that **pbu_$acquire** passes to the device initialization routine after mapping the CSR page(s) into driver address space).

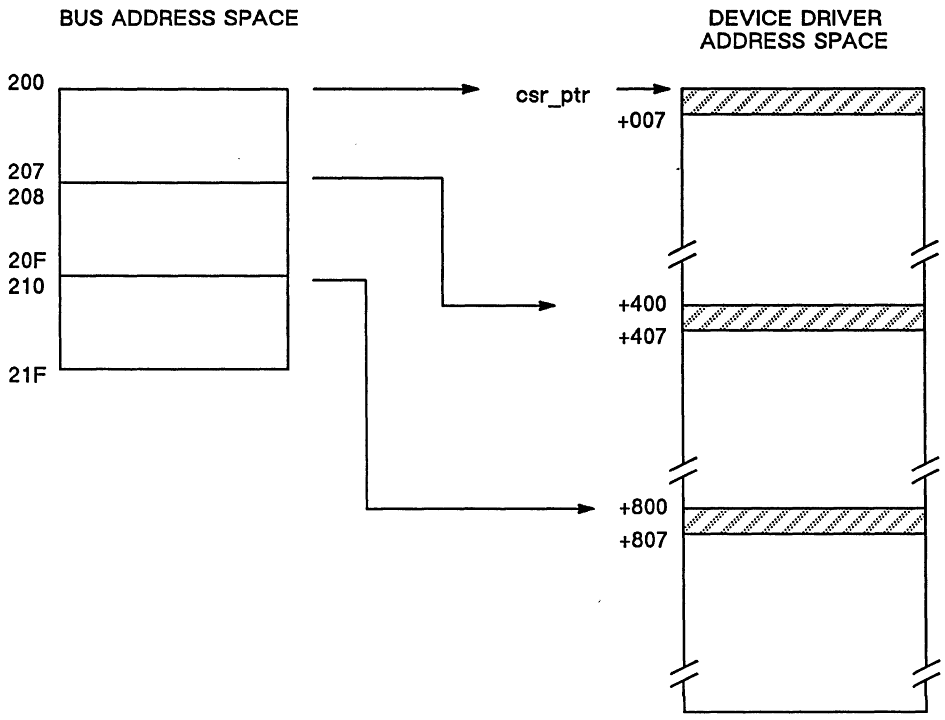


Figure 3-2. CSR Mapping Scheme for PC AT Compatible Devices

Sixteen-bit addresses (so-called PC AT addresses, which are not supported on the PC AT compatible bus) extend the address range beyond the 1 KB (0-3FF) range of 10-bit addresses up to 64 KB (0-FFFF). Such addresses are "folded" and mapped to different locations on the same set of 128 physical pages as are occupied by 10-bit addresses.

Figure 3–3 shows how the 16 bits of an PC AT compatible I/O address are translated to a processor physical address.

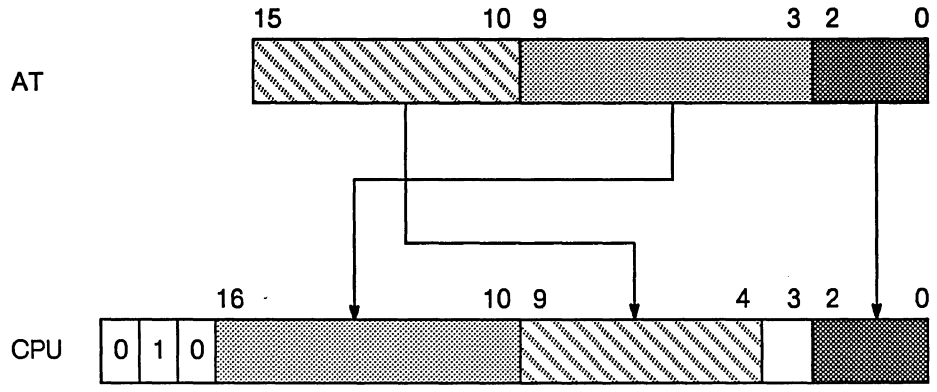


Figure 3–3. Mapping a 16–Bit PC AT Address to Processor Address Space

Apollo provides the **cvt_at** command to return the iova for 10– and 16–bit addresses. The **cvt_at** command also reports any conflict between the address you specify for your device and the address of any system–supplied devices. Refer to Appendix A for the syntax and usage.

3.1.2 Memory Space

The PC AT compatible memory space is used for memory–mapped and bus–master devices. Addresses are mapped one–to–one to processor physical address space. Controllers are mapped and unmapped using the GPIO routines **pbu2_$map_controller** and **pbu2_$unmap_controller**.

Table 3–2 lists the address ranges that are reserved for Domain system–supplied devices as well as those that are available for customer devices. If your system is not configured with the system–supplied device that occupies a particular address range, then you may use that range for your own device. For a more detailed map of memory space usage, refer to the *Domain Personal Workstations and Servers Hardware Architecture Handbook.*

Table 3-2. DN3000/DN4000 Physical Memory Allocated for Domain System-Supplied Devices

Physical Address (Hex)	Device
000000-03FFFF	Reserved for the System
040000-05FFFF	I/O Address Space (see Table 3-1)
080000-09FFFF	Available for Customer Devices
0A0000-0BFFFF	Color or Alternate Monochrome Graphics
0C0000-0DFFFF	Alternate Monochrome Graphics or Apollo Token Ring (0D0000-0DFFFF)
0E0000-0FFFFF	Alternate Color Graphics or Alternate or Second Single-Board Ring (0E0000-0EFFFF)
100000-8FFFFF*	Main Memory
900000-BFFFFF	Available for Customer Devices
C00000-CFFFFF	PC Coprocessor
D00000-DFFFFF	PC Coprocessor Alternate
E00000-F9FFFF	Available for Customer Devices
FA0000-FDFFFF	Monochrome Graphics
FE0000-FFFFFF	Available for Customer Devices

*On the DN4000, this area is not occupied by main memory and is available for customer devices.

3.2 Unit Numbering

The unit number of an PC AT compatible device is identical with the Interrupt Request (IRQ) line. There are 16 possible unit numbers; unit number 0 has the highest priority. However, since Domain system-supplied devices also use this range, not all unit numbers are available for customer devices. The current allocation of unit numbers as well as the interrupt priority (from highest to lowest) assigned to each unit number are listed in Table 3-3 for the DN3000 and in Table 3-4 for the DN4000.

NOTE: In Table 3-3 and Table 3-4 the phrase "or User Device" means that the IRQ reserved for a device that is not present on the system can be used for another device.

Table 3-3. Allocation of Unit Numbers on the DN3000

Unit No. and IRQ	Interrupt Priority	Device
0*	1	Timer
1*	2	Keyboard
2*	-	Reserved
3	3	Apollo Token Ring Network Controller–AT
4	12	SPE — Serial Line 1 or User Device
5	13	Tape Controller
6	14	Disk Controller or User Device
7	15	SPE — Parallel Line or User Device
8*	4	Calendar — Serial Lines 1 and 2
9	5	ETHERNET 2, SPE — Serial Line 2 or User Device
10	6	ETHERNET 1 or User Device
11	7	PC Coprocessor or User Device
12	8	User Device
13*	9	Reserved
14	10	Disk Controller
15	11	PC Coprocessor Alternate or User Device
*This IRQ line is used by the processor and is not available on the bus.		

Table 3-4. Allocation of Unit Numbers on the DN4000

Unit No. and IRQ	Interrupt Priority	Device
0*	1	Timer
1*	2	Keyboard
2*	-	Reserved
3	3	Apollo Token Ring Network Controller–AT
4	12	SPE — Serial Line 3 or User Device
5	13	Tape Controller
6	14	Disk Controller or User Device
7	15	SPE — Parallel Line or User Device
8*	4	Calendar — Serial Lines 1 and 2
9	5	ETHERNET 2, SPE — Serial Line 4 or User Device
10	6	ETHERNET 1 or User Device
11	7	PC Coprocessor or User Device
12	8	User Device
13*	9	Reserved
14	10	Disk Controller
15	11	PC Coprocessor Alternate or User Device
*This IRQ line is used by the processor and is not available on the bus.		

3.3 Testing for Controller Presence

The PC AT compatible bus does not generate bus time–outs. Therefore, you cannot use the GPIO calls **pbu_$read_csr** or **pbu_$write_csr** to test for controller presence on the bus. Instead, you must write to an I/O register control bit and check if the appropriate status bit(s) react as you would expect if the controller were present on the bus.

3.4 DMA and IRQ Lines

DMA and IRQ lines typically float on PC AT compatible controllers. Refer to the device documentation for specific information on enabling these lines. Generally, however, you should do the following:

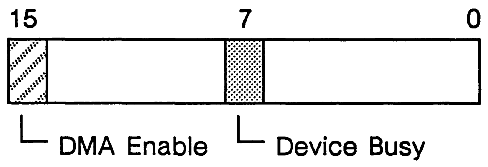
- Call **pbu[2]_$dma_start** after enabling the DMA lines and **pbu[2]_$dma_stop** before disabling them (refer to Appendix B for information on these GPIO calls). If the device only does DMA at your command, you can set a "DMA enable" bit in the driver's initialization routine and then do **pbu[2]_$dma_start** followed by the data transfer command to the device in your driver.

- Call **pbu_$enable_device** after you have set up the controller to have some interrupts enabled. Call **pbu_$disable_device** before you clear all interrupt enables from the controller. Refer to Appendix B for more information on these GPIO calls.

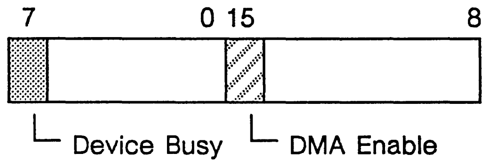
3.5 Byte Swapping

The necessity for byte swapping (transposing the order of the bytes in a word) arises from the fact that the Domain processor orders bytes differently from the way that a PC AT compatible controller does. To compensate for this, I/O hardware performs byte swapping during data transfers according to the following rules:

- I/O hardware transposes the bytes of words transferred between the processor and the bus. Thus, integers and CSRs defined as 16 bits are byte swapped. For example, a CSR that has the following internal representation on the PC AT compatible controller:

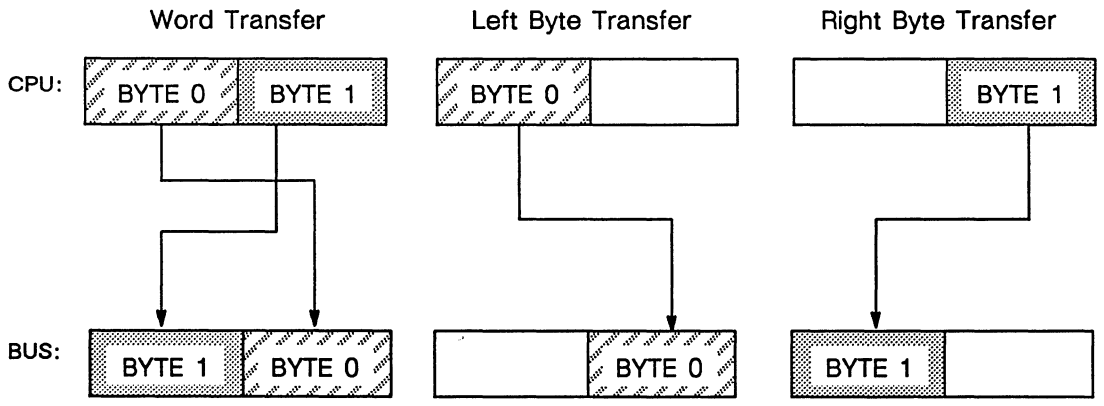


would look like this on our processor:



- Byte swapping does not occur during byte transfers. Thus, characters are transferred correctly.

The following illustration shows byte swapping between the processor and the PC AT compatible bus:



3.6 Software Considerations

GPIO software supports four kinds of I/O operations on the PC AT compatible bus:

- Memory-mapped I/O
- Programmed I/O
- DMA by nonbus-master devices
- DMA by bus-master devices

Memory-mapped I/O and programmed I/O work on the PC AT compatible bus the same way as on the MULTIBUS or VMEbus and do not require any special GPIO routines. DMA operations by both bus-master and nonbus-master devices, however, do require two special GPIO routines: **pbu[2]_$dma_start** and **pbu[2]_$dma_stop**. (If your driver is to run on the DN4000, use **pbu2_$dma_start** and **pbu2_$dma_stop**; if on the DN3000, use **pbu_$dma_start** and **pbu_$dma_stop**.)

If you are writing a driver for a device that cannot request external bus mastership, your driver must surround each DMA operation with **pbu[2]_$dma_start** and **pbu[2]_$dma_stop**, no matter whether the operation was successful or not. The **pbu[2]_$dma_start** routine prepares the processor's DMA hardware for the DMA operation. After the driver calls **pbu[2]_$dma_start**, the controller can begin its operation. When the controller indicates that the operation is completed, the driver next calls **pbu[2]_$dma_stop** to get status from DMA hardware to ensure that the hardware has completed its share of the operation as well. The driver must call **pbu[2]_$dma_stop** even if the controller reports an error. The driver may ignore the status returned by **pbu[2]_$dma_stop**; however, if the controller had a problem, it is likely that the DMA operation did not run to completion. The call to **pbu[2]_$dma_stop** must be made so that software can reset its knowledge of who is using the DMA channel.

If you are writing a driver for a device that can request external bus mastership, your driver must call **pbu[2]_$dma_start** once, specifying the **pbu_dma_cascade** option. This option reserves the DMA channel and provides bus arbitration. The **pbu[2]_$dma_stop** routine must be called when the device is released.

For more detailed information on what GPIO routines to call and how to use them when performing DMA on the PC AT compatible bus, refer to Chapter 7, Subsections 7.1.1 and 7.1.2.

3.7 Creating a DDF for a PC AT Compatible Device

To create a Device Descriptor File (DDF) for a PC AT compatible device, you must specify the **-at** option with the **crddf** command. This option indicates to GPIO software that the device in question resides on the PC AT compatible bus. We recommend that this option be the first specified when building a new DDF. Valid unit numbers when **-at** is specified are in the range 0–15, except for those assigned to Domain system-supplied devices (see Table 3-3 and Table 3-4).

The **-dma_channel** option must be used with PC AT compatible devices to specify the DMA channel number that a controller will use.

Refer to Appendix A for a full description of the **crddf** command and the **-at** and **-dma_channel** options and to Chapter 11, Subsection 11.3.1 for an example of the **crddf** command with the **-at** option.

Chapter 4

Overview of I/O Software

The major components of I/O software are

- One or more application programs (user written)
- GPIO routines and commands (supplied by Apollo)
- Device driver routines (user written)

Section 4.1 through Subsection 4.4.3.4 briefly describe these components and show the relationships among them. Figure 4–1 shows the relationships among the application program, the device driver, and the GPIO routines and commands. Subsection 4.4.4 provides a driver component checklist for your use when writing a driver.

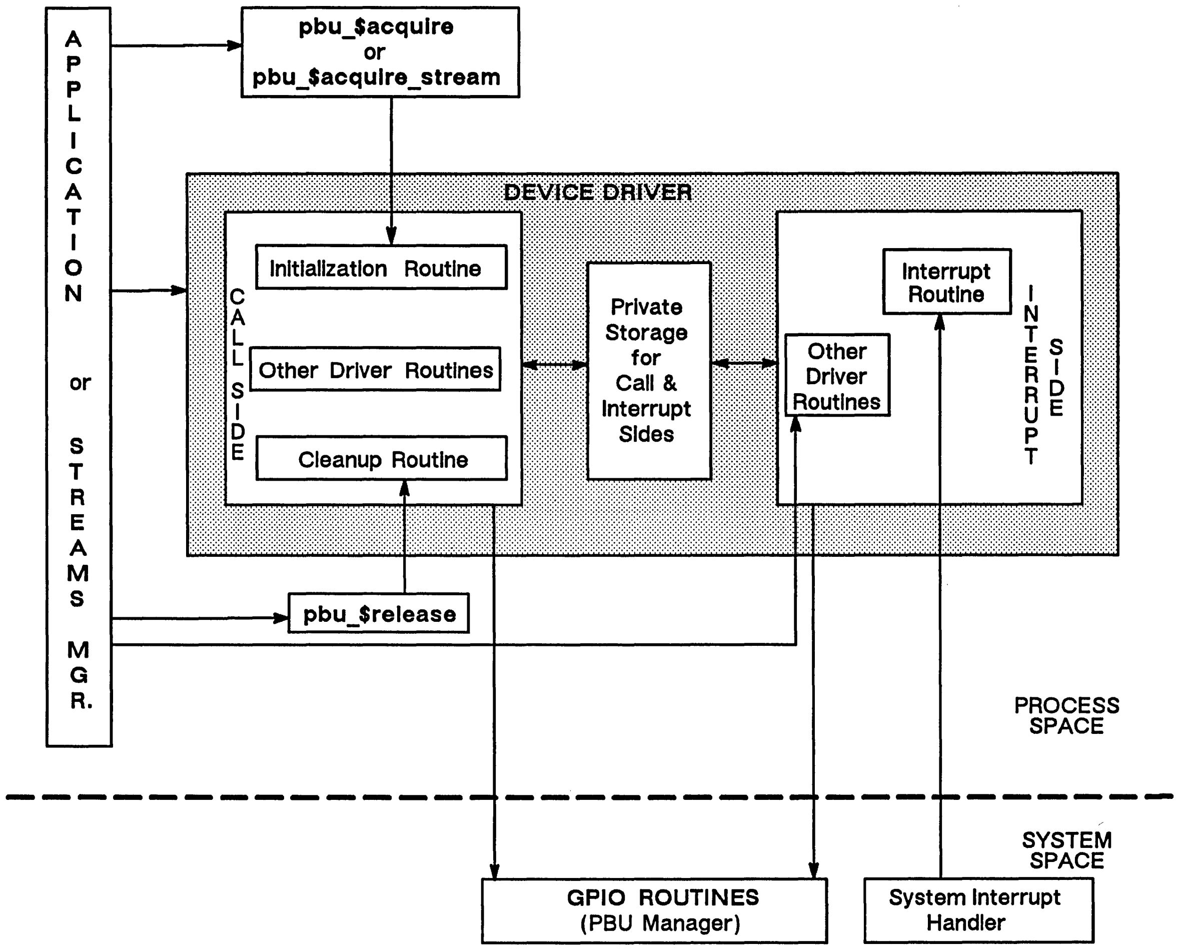


Figure 4–1. Interaction of I/O Software

4.1 Application Program

The application program can consist of one or more programs. For example, application programs can call a device server, which is a collection of programs that perform device-specific processing before calling the device driver to perform an I/O operation. In other cases, the application program is the device driver itself.

4.2 Streams Manager

For information on how to write and use streams, refer to the *Using the OPEN System Toolkit to Extend the Streams Facility* manual, and to Chapter 12, Subsection 12.1.3 in this manual for examples of how to acquire a device with a streams manager using the **pbu_$acquire_stream** routine.

4.3 GPIO Commands and Routines

The GPIO commands and routines create the environment in which a device driver runs by performing the following:

- Controlling the acquisition and release of the device
- Creating and deleting the mapping between a device's memory or registers and processor address space
- Setting up the mechanisms to facilitate data transfers to and from a device

Table 4–1 lists the files associated with the GPIO software product. The individual commands are described in Appendix A, the routines in Appendix B.

Table 4–1. GPIO Software

File	Contents
/lib/pbu_int_lib	Library to be bound with user–written interrupt routines
/lib/pbulib	GPIO routines and interface to internal GPIO manager, automatically installed at system startup
/com/aqdev	**aqdev** (acquire_device) command for users in the Domain/Aegis environment (note that the **aqdev** command is not available to users in the Domain/BSD4.3 or Domain/SysV environments)
/com/rldev	**rldev** (release_device) command for users in the Domain/Aegis environment (note that the **rldev** command is not available to users in the Domain/BSD4.3 or Domain/SysV environments)
/com/crddf	**crddf** (create_ddf) command for users in the Domain/Aegis environment
/com/cvt_at	**cvt_at** (convert_at_addresses) command
/usr/apollo/bin/crddf	**crddf** (create_ddf) command for users in the Domain/BSD4.3 and Domain/SysV environments
/usr/apollo/bin/cvt_at	**cvt_at** (convert_at_addresses) command for users in the Domain/BSD4.3 and Domain/SysV environments
/sys/ins/pbu.ins.pas	Insert file for Pascal programs using GPIO routines
/usr/include/apollo/pbu.h	Insert files for C programs using GPIO routines
/sys/help/pbu.hlp	Help file for GPIO routines and command index to GPIO commands
/sys/help/aqdev.hlp	Help file for the **aqdev** command
/sys/help/rldev.hlp	Help file for the **rldev** command
/sys/help/crddf.hlp	Help file for the **crddf** command
/sys/help/cvt_at.hlp	Help file for the **cvt_at** command
/domain_examples/gpio_examples	Directory containing sample drivers

4.4 Device Driver

The device driver is a user–written program, or set of programs, that controls a peripheral device on behalf of an application program.

4.4.1 Driver Functions

In general, a device driver performs the following functions:

- Ensures that the device is physically present on the bus
- Initializes the driver control block
- Allocates resources required for data transfers
- Processes I/O requests from the application into device–specific commands
- Reads controller status registers
- Responds to device interrupts
- Responds to device time–out conditions
- Responds to requests to cancel an I/O operation
- Performs status checking and error logging
- Returns status from the device to the application that made the I/O request

4.4.2 Major Components of a Driver

To carry out these functions, a device driver may include the following routines:

- An initialization routine called during device acquisition. This routine creates controller data structures and readies the device for I/O operations. You must include this routine in your driver, using the calling sequence described in Chapter 6, Subsection 6.1.1.

- One or more interrupt routines called by the System Interrupt Handler to respond to device interrupts. This routine is optional. If you decide to write an interrupt routine, use the calling sequence described in Chapter 8, Subsection 8.2.1.

- A cleanup routine called during device release (by **pbu_$release**). This routine ensures that no I/O is in progress to or from the device and that the device will not generate any more interrupts. Write the cleanup routine according to the calling sequence in Chapter 6, Section 6.4. Although this routine is optional, we strongly recommend that you include it in your device driver.

In addition, a driver may include one or more of the following routines:

- A validation routine that checks device–specific parameters of an I/O request
- I/O preprocessing routines that allocate the needed I/O data structures, depending upon the type of transfer and the type of bus
- A data transfer routine
- A wait routine that waits for an interrupt or device time–out while the I/O operation is in progress
- Command handling routines that process commands from the application

Which of these routines you decide to include in your driver and how you implement them depends on the requirements of the device and the application. To help you with the design of your driver, Part 2 of this manual "Writing a Driver" describes the driver components in detail and explains how to construct them by using GPIO routines. Part 3 "Reference Information" provides information, such as the format and syntax of GPIO commands and routines, performance information, and so on. You may also find it helpful to refer to the following online sample drivers, located in subdirectories of **/domain_examples/gpio_examples**:

- Versions in C and Pascal of a device driver for a hypothetical "bulk memory" device (see subdirectories **bm_example_c** and **bm_example**; see also the program listings in Appendixes E [C] and F [Pascal])
- A device driver for an Interlan controller (see subdirectory **interlan_example**)
- A device driver for a 3Com* controller (see subdirectory **threecom_example**)
- A shared driver for the SPE board (see subdirectory **global_example**)

To make the device driver accessible to user programs, you must bind the routines as described in Chapter 10, Subsection 10.1.2. If your driver includes one or more interrupt routines, you must bind them separately from the other routines.

You specify the pathname(s) of the device driver and the entry points of the initialization, interrupt, and clean-up routines using the **crddf** (create_ddf) command. This command establishes a DDF that describes the device to the system and allows GPIO routines to call driver routines. See Chapter 11 and Appendix A for information about the purpose of the DDF, how to build the DDF with the **crddf** command, and the options available with the **crddf** command.

When a user process acquires the device (see Chapter 12), the driver routines are loaded into its address space so that application programs can call them. The set of driver routines that programs can actively call constitutes the call side of the driver, whereas the interrupt routine(s) and associated data structures make up the interrupt side of the driver.

* 3Com is a registered trademark of 3Com Corporation.

4.4.3 Operation of a Driver: A Dry Run of bm_example

You may find the online sample driver in **bm_example** a good place to begin familiarizing yourself with a driver. In order to give you a feel for how it functions, the following paragraphs explain a typical DMA operation. The driver was written for a hypothetical bulk memory MULTIBUS device in order to illustrate the general design of a driver and to demonstrate the use of GPIO routines. For these reasons, the driver and the fictitious controller for which it was written were kept simple: the controller has five 8-bit registers and can perform read and write DMA operations. However, **bm_example** is a compilable functioning driver and includes all the major components. Figure 4-2 illustrates how these components relate to each other as well as to the application and GPIO routines. A slightly reorganized version of the **bm_example** driver appears in Appendixes E (C) and F (Pascal).

Note that names of driver routines begin with **bm** (Bulk Memory), whereas names of GPIO routines all begin with **pbu** (Peripheral Bus Unit). Also, names of driver routines that do not include a dollar ($) sign (for example, **bm_command**) are internal subroutines that are not referenced outside the module in which they are defined.

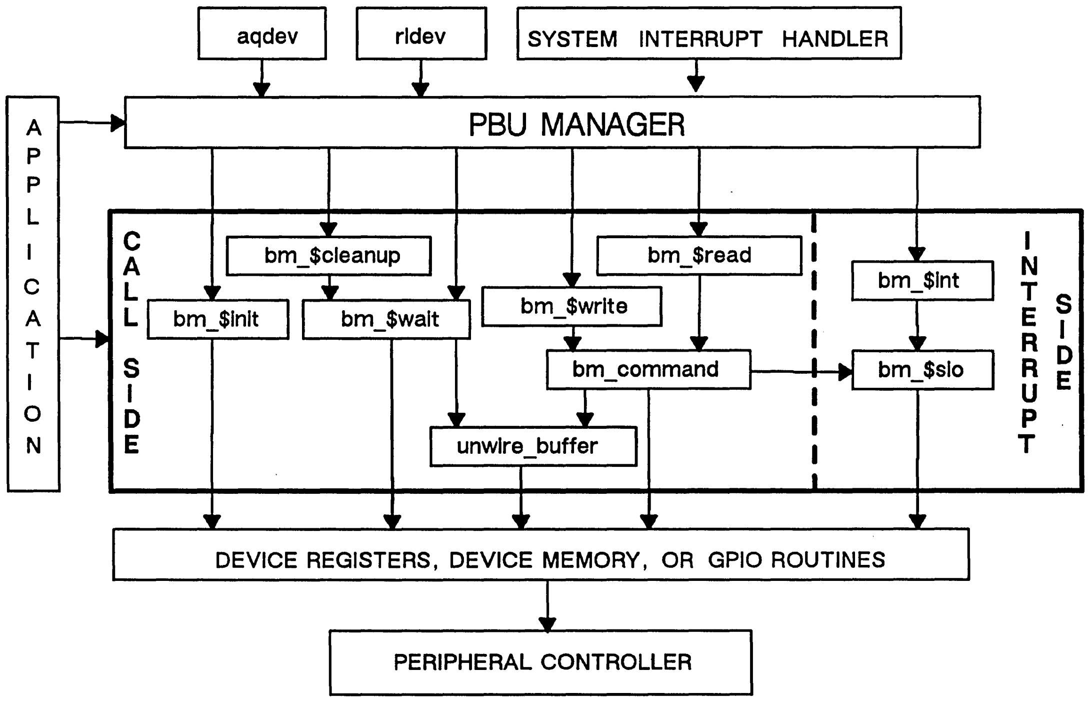


Figure 4-2. Driver Routines in bm_example

4.4.3.1 Initialization

After the device has been acquired, the PBU Manager (a collection of routines that are internal to the operating system and manage GPIO resources) activates the driver's initialization routine, **bm_$init**. This routine does the following:

- Initializes the driver control block (bmcb)
- Calls **pbu_$write_csr** to determine if the device is physically present on the bus
- Calls **pbu_$allocate_map** to allocate an area of the I/O map for mapping buffers to MULTIBUS address space

The **bm_$init** routine then returns control to the PBU Manager. The driver is now ready to accept I/O commands from the application.

4.4.3.2 Command Processing

The application calls one of the command–handling routines, **bm_$read** or **bm_$write,** depending on the type of I/O operation. Either routine immediately calls an internal routine, **bm_command,** which in turn calls the following GPIO routines:

- **pbu_$wire,** to make the I/O buffer permanently resident in processor address space so that it is unavailable to the operating system's page–replacement mechanisms
- **bm_$sio,** to start up the DMA operation
- **pbu_$enable_device,** to allow the controller to issue interrupts

When the driver's data transfer routine, bm_$sio (the start I/O routine), is called, it does the following:

- Calls **pbu_$map,** which maps the I/O buffer into MULTIBUS address space
- Issues the read or write command to the controller via the CSR page

Program control then passes from **bm_$sio** through **bm_command** and **bm_$read/write** to the application. The application calls the driver's wait routine, **bm_$wait,** which in turn calls the following GPIO routines:

- **pbu_$wait,** to wait either for the eventcount to advance (for information about eventcounts, refer to Chapter 6, Section 6.3) or for a specified interval to pass, whichever comes first
- **pbu_$unmap,** to unmap the I/O buffer from MULTIBUS address space
- **pbu_$unwire** (called via an internal routine, **unwire_buffer**), to unwire the I/O buffer

The **bm_$wait** routine then returns a status code to the application that indicates whether or not the I/O operation was complete.

4.4.3.3 Interrupt Handling

When the I/O operation is complete, the device issues an interrupt that is intercepted by the System Interrupt Handler. The System Interrupt Handler then transfers program control to the driver's interrupt routine, **bm_$int**. This routine first determines whether any more data remains to be transferred. If there is, **bm_$int** calls **bm_$sio** to start the next data transfer and enables the controller interrupt logic. Once all data has been transferred, **bm_$int** advances the eventcount and returns program control to the PBU Manager.

4.4.3.4 Cleanup

The PBU Manager calls the driver's cleanup routine, **bm_$cleanup**, when either the application calls **pbu_$release** or the user inserts the End-Of-File (EOF) mark (under the DM, this is usually done by pressing CTRL/Z or CTRL/D). Initially, **bm_$cleanup** determines if an I/O operation is still in progress. If so, it either resets the controller or calls **bm_$wait**, depending on what the application specifies. Regardless of whether an I/O operation is still in progress, **bm_$cleanup** calls the following GPIO routines:

- **pbu_$free_map**, to release the area of the I/O map previously allocated by **pbu_$allocate_map**
- **pbu_$disable_device**, to prevent the controller from issuing any more interrupts

The **bm_$cleanup** routine then returns program control to the PBU Manager, thus concluding operation of the driver.

4.4.4 Driver Checklist

Following is a checklist of components that can be included in a driver. *Italicized* items must be included. Whether or not you decide to include any of the other items depends on the device you are supporting, the application, and your convenience.

- ❑ Insert files (Chapter 5)
  - ❍ *System Insert Files* (Section 5.1)
  - ❍ CSR Page (Subsection 5.2.1.1)
  - ❍ Driver Control Block (Subsection 5.2.1.2)
- ❑ *Call–Side Library* (Chapter 6)
  - ❍ *Initialization Routine* (Section 6.1)
  - ❍ Command–Processing Routine (Section 6.2): Required if the device is to be under the control of the application
  - ❍ Wait Routine (Section 6.3): Necessary if your driver has an interrupt routine
  - ❍ Cleanup Routine (Section 6.4): Highly recommended
  - ❍ *Data–Transfer Routine* (Chapter 7): Can be installed in either the call–side library or (if one exists) the interrupt–side library
- ❑ Interrupt Library (Chapter 8): Required only if your driver has an interrupt routine
  - ❍ Interrupt Routine (Section 8.2): Required if your device handles interrupts and performs asynchronous transfers
  - ❍ Start I/O Routine (SIO) (Section 8.3): Must be installed in the interrupt–side library if called by any interrupt–side routine; otherwise, can be included as part of the data–transfer routine in the call–side library
- ❑ *Device Descriptor File*: (Chapter 11)

Chapter 5

Insert Files

Insert files are included in the driver to enable it to reference certain resources: either system calls that reside outside the driver (GPIO routines) or routines and data structures that exist within the driver and which both call–side and interrupt–side routines can reference. To reference any of these resources, you must specify the pathname of the insert file (using the #include keyword in C or the %INCLUDE directive in Pascal) in the module where the calling routine resides. This chapter describes which system insert files to include in the driver and explains how to set up driver–specific insert files. For a description of insert files in general and available system calls, see the *Programming with Domain/OS Calls* manual.

NOTE: Unlike Pascal, the C programming language is case–sensitive; therefore, all system procedure names (such as GPIO routines) must be lowercase, which is consistent with their appearance in the system insert files. However, any global names in C that are accessed by GPIO routines are case–sensitive.

5.1 System Insert Files

Table 5-1 shows the pathnames for the required and optional system insert files.

Table 5-1. System Insert Files

Language	Required Insert Files	Optional Insert Files
Pascal	**/sys/ins/base.ins.pas** (base definitions)	**/sys/ins/vfmt.ins.pas** (variable formatting calls)
	/sys/ins/pbu.ins.pas (GPIO routines)	**/sys/ins/error.ins.pas** (error reporting calls)
C	**<apollo/base.h>** (base definitions)	**<apollo/vfmt.h>** (variable formating calls)
	<apollo/pbu.h> (GPIO routines)	**<apollo/error.h>** (error reporting calls)

5.2 Driver-Specific Insert Files

Driver-specific insert files serve as links between the call side and the interrupt side of the driver and between the driver and the application. They fall into two categories:

- Public Insert Files: Declare data structures and driver routines that the application can use

- Private Insert Files: Declare the structures and routines to which the driver alone refers

This division between public and private is admittedly an artificial distinction, and you may wish to ignore it by creating only one driver-specific insert file, especially if your driver is simple and straightforward. However, creating two insert files does have the advantage of presenting to the user, who may not care to know the inner workings of the driver, only what is pertinent to interfacing the application with the driver. At any rate, we have followed the distinction here, and Subsections 5.2.1 and 5.2.2 describe private and public insert files separately.

Examples of public and private insert files appear in the **bm_example** in Appendix F, Sections F.1 and F.2.

5.2.1 Private Insert File

The private insert file connects the call and interrupt sides of the driver. It is where you declare those internal components (flags, pointers, records, etc.) that are common to both sides. The three most important of these components (the CSR page, the driver control block, and internal driver routines) are described in Subsections 5.2.1.1 through 5.2.1.3.

5.2.1.1 CSR Page

The CSR page is a record structure that defines the controller's internal registers which the driver needs to access, such as the command, status, and address registers. It is through the CSR page that the driver reads and writes to those registers. For this reason, it is important to set up each field in the CSR page so that it exactly matches the position of the corresponding register in controller memory. This procedure ensures against, for example, the driver writing to what it takes to be a write–only command register when in fact it is a read–only status register.

An example of a CSR page as declared in a private insert file follows:

```
typedef union mm_csr_page_t {
    struct {
        unsigned char command;
        unsigned char status;
        unsigned char pad_1;
        unsigned char r_data;
        unsigned char pad_2;
        unsigned char int_status;
        unsigned char pad_3;
        unsigned char pad_4;
        unsigned char d_data;
        unsigned char int_enable;
        unsigned char pad_5;
        unsigned char pad_6;
        unsigned char pad_7;
        unsigned char pad_8;
        unsigned char pad_9;
    } c;

    char all[bytes_per_page];
} mm_csr_page_t #attribute[device];
```

As you examine the previous example, note the following points:

- The #attribute [device] directive in this example is designed for use in a device driver to protect against any undesired compiler optimization. Its function is explained more fully in Appendix C, Section C.3.
- The record structure itself is of the union type so that, in this case, the CSR page can be accessed either as a whole or register by register; it could, however, have been constructed of fixed parts only, depending upon the requirements of the driver.
- Each structure member is of the char data type because each register consists of eight bits; that is, the space allocated to the char data type. (Use of the char data type, or arrays of chars, to specify structure members ensures that the compiler does not perform improper compressions.)
- The field "all" is declared as an array of bytes_per_page chars because that is the space allocated to any CSR page.
- Finally, pads are used where appropriate to maintain proper spacing between registers. Note that pad_5 through pad_9 could also have been coded as an array:

  ```
  char pads[9-5+1];
  ```

- In this CSR page, the interrupt enable register (int_enable), a write-only register, is offset at 09 hex from the base address. If we were to remove the pads from the CSR page record, int_enable would then be offset at 05 hex. Any attempt to write to this register would result in a bus time-out error since we would actually be trying to write to a read-only register, the interrupt status register (int_status), which is offset at 05 hex. If you are in any doubt about the positioning of fields within the CSR page, you should use the compiler's **-map** option so that you can check the field displacements within the CSR page definition.
- The record structure that defines the CSR page is referenced as a pointer; for this reason, a declaration such as the following also appears in the private insert file:

  ```
  typedef union mm_csr_page_ptr_t {
      mm_csr_page_t        *c;
      pbu_$csr_page_ptr_t p;
  }  mm_csr_page_ptr_t;
  ```

- The pointer in this example is declared as a union so that it can be used in two different contexts, either in the driver or in a GPIO routine.

For tips on setting up the CSR page, refer to Appendix C, Section C.1.

5.2.1.2 Driver Control Block

Although the driver control block is optional, you may find it useful to include one in your driver as a storage area to be used for communications between the call and interrupt sides. It contains information that is shared by different driver routines and continuously updated, such as status flags, buffer address and length, and so on. The nature and layout of this information depend upon the requirements of the driver and the convenience of the programmer. In the following example, because the control block is referenced by the interrupt handler, it must be part of the interrupt library.

It should be noted that, for drivers written in Pascal, if the control block is referenced by the interrupt side, it must be allocated (using the DEFINE clause) in the interrupt library; for more information on defining globals in drivers written in Pascal or C, refer to Appendix C, Subsection C.2.5.

The driver control block in **bm_example_c** is declared in **bm.h** as follows:

```
typedef union {
    struct {
       unsigned int   init: 1;       /* set to true when controller
                                        initialized */
       unsigned int   buffer_wired : 1; /* set when a buffer is wired */
       unsigned int   busy : 1;      /* set when an operation is in
                                        progress */
       unsigned int   done : 1;      /* set by interrupt routine when
                                        transfer completes */
       unsigned int   pad : 4;       /* fill out to byte ? */
    } b;
    char    all;
} bm_$flags_t;

/* status register definition */

typedef union {
    struct {
       unsigned int   attention: 1; /* 1 => change in controller
                                            status */
       unsigned int   status_modifier : 1; /* 1 => current
                                                  status unavailable */
       unsigned int   control_unit_end : 1;/* 1 => busy condition
                                                  cleared */
       unsigned int   busy : 1;     /* 1 => controller currently busy */
       unsigned int   channel_end : 1;     /* 1 => end of operation */
       unsigned int   device_end : 1;      /* 1 => end of operation */
       unsigned int   unit_check : 1;      /* 1 => parity error in bm */
       unsigned int   unit_exception : 1;  /* 1 => illegal bm address */
        } b;
        unsigned char    all;
} bm_$status_t;
```

```
typedef struct {                               /* define communications area */
    pbu_$unit_t        pbu_unit_number;  /* number of this pbu device */
    bm_$flags_t          flags;
    char                 pad;        /* a byte of padding */
    pbu_$ddf_ptr_t       ddf_ptr     /* pointer to mapped ddf */
    bm_$csr_page_ptr_t   csr_ptr;    /* pointer to mapped csr page */
    pbu_$iova_t          bm_iova;    /* start of our area of i/o address
                                        space */
    bm_$buf_ptr_t        bufaddr;    /* address of start of buffer */
    bm_$buf_len_t        buflen;     /* total length of buffer */
    bm_$bm_address_t     bm_address; /* address of start of bm area */
    unsigned char        command;    /* current command (read or
                                        write) */
    bm_$buf_len_t        rem_len;    /* length remaining to read or
                                        write */
    bm_$status_t         status;     /* status from last interrupt */
    status_$t            sio_status; /* status from bm_$sio (start I/O
                                        routine) called from int side */
    bm_$buf_ptr_t        io_addr;    /* address of last i/o transfer */
    bm_$buf_len_t        io_len;     /* length of last i/o transfer */
    unsigned char        init_cmd;   /* initialization command (see
                                        bm_$init!) */
    unsigned char        read_cmd;   /* read command */
    unsigned char        write_cmd;  /* write command */
} bm_$bmcb_t;

extern bm_$bmcb_t bmcb;              /* main control block */
```

5.2.1.3 Internal Driver Routines

The only routines that must be referenced (using the EXTERN clause) in the private insert file are those functions and procedures that are shared by the call and interrupt sides, but not by the application. These routines must be allocated in the interrupt side. In **bm_example**, there is only one such routine: **bm_$sio** (the start I/O routine). However, you may wish to list all external routines (except those already referenced in the public insert file; see Subsection 5.2.2) for documentation purposes.

5.2.2 Public Insert File

The public insert file is a convenience for the user who wants to know only what is necessary to interface the driver with the application. It therefore typically contains device status codes that the user may want to access and any user–callable routines within the driver, such as status–checking routines and user–visible entry points. The three user–callable routines listed in the **bm_example_c** public insert file, **bm.h**, are **bm_$read**, **bm_$wait**, and **bm_$write**.

Chapter 6

Call–Side Routines

This chapter describes the following call–side routines:

- Initialization
- Command Processing
- Wait
- Cleanup

The data–transfer routine, which may be included in either the call–side library or the interrupt–side library, is treated separately in Chapter 7.

For information on fault handling, refer to the description of the PFM calls in the *Domain/OS Calls Reference* manual.

> **NOTE:** Unlike Pascal, the C programming language is case–sensitive; therefore, all system procedure names (such as GPIO routines) must be lowercase, which is consistent with their appearance in the system insert files. However, any global names in C that are accessed by GPIO routines are case–sensitive.

6.1 Initialization

The device acquisition routine, **pbu_$acquire**, calls the driver initialization routine to perform the functions necessary to ready a controller for I/O operations. Typically, these functions include:

- Initializing any internal storage for the device driver and writing to it the device unit number and pointers to the CSR page and the DDF.
- Accessing the DDF (if necessary) to determine how the controller is configured on the system.
- Ensuring that the controller is present on the bus.
- Allocating I/O resources and saving pointers to these resources within the driver's control block. The resources allocated depend upon the method of data transfer used by the controller and the type of bus.
- Performing controller-specific initialization. This step can include setting up any initialization control blocks or data structures that the controller requires.
- Enabling device interrupts.

It should be noted that the initialization routine need not return after it initializes the device; it can perform all required device I/O, service requests from other processes, and so on.

Chapter 7 describes resource allocation for DMA and memory-mapped I/O, and Chapter 8, Subsection 8.2.2 describes device enabling and disabling. Subsections 6.1.1 through 6.1.4 give more information about the required calling format for the initialization routine, initializing driver storage, testing for controller presence, and setting up controller-specific data structures. For an example of an initialization routine, see the **bm_$init** routine in Appendix E, Section E.2 (C) and Appendix F, Section F.3 (Pascal).

6.1.1 Initialization Routine Format

The initialization routine is called by GPIO software and must, therefore, conform to the following calling sequence (shown in C and in Pascal):

Synopsis (C)

```
void initialization_routine(
        pbu_$unit_t         &unit,
        pbu_$ddf_ptr_t      &ddf_ptr,
        pbu_$csr_page_ptr_t &csr_ptr,
        status_$t           *status
        );
```

Synopsis (Pascal)

```
procedure initialization_routine(
        in  unit:    pbu_$unit_t;
        in  ddf_ptr: pbu_$ddf_ptr_t;
        in  csr_ptr: pbu_$csr_page_ptr_t;
        out status:  status_$t
        );
```

Description

If the initialization routine returns a nonzero status, **pbu_$acquire** unloads the driver, releases the device, and returns an error status to its caller.

The input and output parameters are described as follows:

csr_ptr	The virtual address of the device's CSR page in pbu_$csr_page_ptr_t format.
ddf_ptr	A virtual address of the DDF in pbu_$ddf_ptr_t format. This data type is described in Appendix B, Section B.1.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format.

6.1.2 Initializing Driver Internal Storage

Some device drivers may require an internal storage area, such as a driver control block, to be used for communications between their call and interrupt sides. (The interrupt side of the driver allocates this storage area, using the DEFINE clause, see Appendix C, Subsection C.2.5.) If a storage area has been defined, it should be initialized by the initialization routine. (When **pbu_$acquire** maps the page that contains the device's CSRs into user–process address space, it passes a pointer to the CSR page to the initialization routine. If the initialization routine has stored the pointer, your program can refer to the CSR page as necessary.) The routine can then optionally store pointers to the mapped CSR page and DDF within it. During an I/O transfer, the call and interrupt routines can read and write to it such information as I/O buffer location and length, current transfer status (read or write), interrupt status, and other statistics.

In **bm_example_c**, the initialization routine, bm_$init, initializes the control block bmcb with the following assignments:

```
bmcb.pbu_unit_number = unit;    /* unit number to pass pbu
                                   manager */
bmcb.ddf_ptr = ddf_ptr;         /* pointer to mapped ddf */
bmcb.csr_ptr.p = csr_ptr;       /* pointer to mapped controller
                                   page */
```

6.1.3 Testing for Device Presence

If a device is not present on the bus (MULTIBUS or VMEbus only) or if the driver attempts to reference a nonexistent CSR, the system generates a bus time–out error and returns the application program to the shell command level (unless it has specified a fault handler; see Chapter 7, Subsection 7.1.4).

The initialization routine can test for a device's presence by reading or writing to its CSR with the routines **pbu_$read_csr** or **pbu_$write_csr**. If the read or write request causes a bus time–out error, the routines suppress normal bus time–out handling and instead return an error status to the driver. In this way, the driver can retain control even if the device is not responding or does not exist. (Device drivers can also use **pbu_$read_csr** and **pbu_$write_csr** to refer to addresses on a memory–mapped controller; see Chapter 7, Subsection 7.2.2.)

NOTE: The PC AT compatible bus does not generate bus time–out errors, which means that you cannot use **pbu_$read_csr** or **pbu_$write_csr** to test for device presence; instead, you must tweak the appropriate device register and see if it responds as you would expect if the device were present.

In the following segment from **bm_example_c**, **bm_$init** calls **pbu_$write_csr** in order to test for device presence and to initialize it. After **pbu_$write_csr** returns, **bm_$init** checks status for a nonzero value, indicating that the device was not present; if the status is nonzero, program control returns to **pbu_$acquire**.

```
pbu_$write_csr(bmcb.pbu_unit_number,    /* number of this pbu
                                           device */
        (char)bmcb.csr_ptr.c->command, /* the command
                                           register */
        BM_INIT_CMD,                    /* initialization
                                           command */
        false,                          /* do a byte, not word
                                           write to command
                                           reg */
        status);                        /* returned status */

if (status->all == pbu_$bus_timeout) { /* controller probably
                                           not there if error */
        status->all = bm_$no_controller;
        return;
}
else if (status->all != status_$ok) {
        status->s.fail = 1;
        return;
}
```

This next example (taken from **/domain_examples/gpio_examples/global_example**), tests for the presence of the device by issuing several device-specific commands.

```
{ To test for the presence of the controller and that we are handling}
{ the CSR page pointer correctly we will write to the controller and }
{ check if the appropriate status bit(s) react as you would expect   }
{ if the controller were present on the bus                          }

        with cb.device_csr_ptr.c^ : csr do begin

{ Initialize controller to test for it's presence }

            cb.line_cntrl_copy.wrd_len := 3;        { 8-bits/char }
            cb.line_cntrl_copy.dlab := false;       { latch/off }
            csr.line_cntrl := cb.line_cntrl_copy.all;
            time_$wait(time_$relative, device_wait_time, st);

            csr.int_enable := no_interrupts_byte;      { disallow all }
                                                       { interrupts }
            time_$wait(time_$relative, device_wait_time, st);

            cb.modem_cntrl_copy.loop := true;          { loopback }
            csr.modem_cntrl := cb.modem_cntrl_copy.all;
            time_$wait(time_$relative, device_wait_time, st);
            cb.line_cntrl_copy.dlab := true;             { latch/on }
            csr.line_cntrl := cb.line_cntrl_copy.all;
            time_$wait(time_$relative, device_wait_time, st);

            csr.data := chr(12);             { set baud rate to 9600 }
            time_$wait(time_$relative, device_wait_time, st);

            csr.int_enable := chr(16#00);  { set second baud rate byte }
            time_$wait(time_$relative, device_wait_time, st);

            cb.line_cntrl_copy.dlab := false; { done with setting baud }
                                              { rate, latch/off }
            csr.line_cntrl := cb.line_cntrl_copy.all;
            time_$wait(time_$relative, device_wait_time, st);

{ To test for the presence of the controller we will write to the}
{ transmit register. This should cause the 'data ready' bit in the line}
{ status register to become true. Next we will read the receive}
{ register. This should cause the 'data ready' bit in the line status}
{ register to become false. }

            csr.data := chr(16#5a); { write dummy data to transmit reg }
            time_$wait(time_$relative, device_wait_time, st);

            cb.line_st_copy.all := csr.line_st;
            if not cb.line_st_copy.data_rdy then begin
                if dbg in cb.flags then
                    vfmt_$write2('data ready NOT true, should be
                                                            true%.',0,0);
                status.all  := device_no_controller;
                return;
                end;
```

6.1.4 Initializing Controller Data Structures

Certain controllers, particularly those based on Intel 8089 I/O processors, may need to use initialization control blocks or other data structures that are located at preset, or hard-wired, memory addresses. During initialization, the controller makes DMA references to these control blocks that are indistinguishable from normal DMA transfers to and from processor memory. If a controller uses hard-wired addresses during initialization, the initialization routine must first allocate memory for these addresses.

6.1.4.1 Allocating Hard–Wired Control Blocks on the MULTIBUS

The initialization routine allocates hard–wired addresses by

- Calling the routine **pbu_$allocate_map**
- Specifying the memory's starting address within MULTIBUS memory
- Giving a length, which must be in 1024–byte increments

As stated in Chapter 1, Subsection 1.2.2, each I/O map entry maps one page of MULTIBUS memory address space. The **pbu_$allocate_map** routine allocates the I/O map entries that correspond to the MULTIBUS address specified in the call, thereby reserving the addresses occupied by the control blocks.

For example, if a controller refers to MULTIBUS address FFF6 for an initialization control block, the initialization routine calls **pbu_$allocate_map** and specifies MULTIBUS address FC00 (because it is a page–aligned address) and a length of 1024. Because the routine specifies a particular address, the *force_flag* parameter must be set to "true" (see Appendix B for a syntactic description of the GPIO call **pbu_$allocate_map**). If the driver needs to allocate two pages of address space in addition to the page required during initialization, it specifies a MULTIBUS address of F400 (FC00–800) and a length of 3072.

Controllers that use hard–wired control blocks during initialization greatly reduce the flexibility with which the I/O map can be allocated. Moreover, if several peripheral devices are simultaneously in use, the MULTIBUS address that the controller requires might already be allocated to another controller. Since most controllers allow you to specify hard–wired MULTIBUS addresses by setting switches on the controller, you should refer to the information in Table 1–2 to avoid setting MULTIBUS addresses that Domain controllers are likely to use.

NOTE: We make no guarantee that the addresses currently used by Domain controllers will not change.

6.1.4.2 Defining Page–Aligned Control Blocks

Device drivers for controllers using hard–wired initialization control blocks or PC AT compatible and VMEbus controllers that need to align a 1–KB buffer must also ensure that the data area used to define the control blocks is page aligned by allocating a buffer at least one page larger than the required size.

The following program allocates a page–aligned buffer for a data area less than or equal to one page, and then sets the sixth byte in the page to 0 ("bytes_per_page" is defined in pbu.ins.pas):

```
#nolist;
#include "<apollo/base.h>"
#include "<apollo/pbu.h>"
#list;

typedef struct buffer_t {                    /* define buffer page */
    char page[bytes_per_page];
} buffer_t;

char buffer[2*bytes_per_page-1];

main(argc, argv)
int argc;
char *argv[];
{
   buffer_t *p;

   p = (buffer_t *) buffer;                  /* point to start of buffer */

   p = (buffer_t *) (((unsigned long)p + bytes_per_page-1)
         & ~(bytes_per_page-1));             /* round up to page boundary */

   printf("buffer = %lx, p = %lx\n", buffer, p);

   exit(0);
}
```

You can also page align control blocks and data buffers when you bind the driver by using the **–align** option; refer to Chapter 10, Subsection 10.1.2.1.

6.2 Command Processing

The driver's command–processing routine (or any other driver routine that performs command processing) is the application's entry point into the driver. The command–processing routine receives I/O requests from the application and, on the basis of those requests, passes the appropriate command to the device. There are several ways to set up command processing in the driver. The driver may include routines for each kind of I/O request that the application may issue; one routine may handle all requests, or the initialization routine may do all command processing (it all depends upon the requirements of the application and the kinds of I/O that the peripheral device services).

Command processing in **bm_example** is performed by two types of routines (see Appendix E, Section E.2 [C]) and Appendix F, Section F.3 [Pascal]:

- Command-specific routines, **bm_$read** and **bm_$write**, that the application can call
- An internal routine, **bm_command**, that is called by the command-specific routines to perform any common processing before passing control to the routine that starts the I/O operation

Depending on whether the application wants the controller to do a read or write operation:

- The application calls either **bm_$read** or **bm_$write**, passing as parameters:
  - The data buffer to be transferred
  - Its address
  - The bulk memory address
- The **bm_$read** or **bm_$write** routine passes the same parameters, along with the specific controller commands, to **bm_command**
- The **bm_command** routine:
  - Takes care of any processing common to both read and write commands, such as checking to see that the controller has been initialized and is not busy and validating buffer length and address
  - Wires down the buffer by calling **pbu_$wire** (wiring ensures that no buffers are removed from memory, or "paged out," during the I/O operation)
  - Calls **bm_$sio** to start the I/O operation

    The following program segment from **bm_command** shows how it prepares for the call to **bm_$sio** (the expressions in the assignment statements were passed to **bm_command** as parameters by one of the command-specific routines):

```
bmcb.command = command;      /* command to perform */
bmcb.io_addr = bmcb.bufaddr; /* first address to transfer */
bmcb.rem_len = len;          /* length "remaining" to transfer */
bmcb.bm_address = bm_address; /* where to start in the bm */
bm_$sio(status);             /* start up the operation */
```

  - Just before returning, **bm_command** enables interrupts by calling **pbu_$enable_device**.

6.3 Waiting for Device Interrupts

The function of a wait routine is to defer any driver activity until either an interrupt occurs (usually indicating the end of an I/O operation) or a specified time–out value elapses. Wait routines, or for that matter any other driver routine, can wait for interrupts from a device by calling either **pbu_$wait** alone or both **pbu_$get_ec** and **ec2_$wait**. The wait routine in **bm_example** is **bm_$wait**; see Appendix E, Section E.2 (C) and Appendix F, Section F.3 (Pascal).

6.3.1 Using pbu_$wait

Drivers (and their applications) use **pbu_$wait** if they need to wait for only three events:

- Device interrupt
- Device time–out
- Quit fault (asynchronous fault) from the terminal user

The **pbu_$wait** routine waits for any or all of the these events by checking for either of the following conditions:

- The System Interrupt Handler has advanced the device's eventcount since the last call to **pbu_$wait**. If the eventcount is advanced, **pbu_$wait** returns immediately. Eventcounts are described in Chapter 8, Subsection 8.2.3.
- A positive time–out value. If the time–out value is less than or equal to 0, **pbu_$wait** returns. Otherwise, the routine waits for the specified interval or until the System Interrupt Handler requests an eventcount advance.

The **pbu_$wait** routine contains an internal flag that indicates whether or not the System Interrupt Handler has advanced the device's eventcount. When **pbu_$wait** returns, it resets this flag to indicate an eventcount advance.

The caller can also permit quit faults (CTRL/Qs or CTRL/Ds) to terminate the wait state by specifying a parameter to **pbu_$wait**; refer to Appendix B for a description of **pbu_$wait** calling format.

The **bm_$wait** routine in **bm_example_c** specifies "index" as the output parameter of **pbu_$wait**. Depending on whether the value of index is 0, 1, or 2, **bm_$wait** then determines which of the three events occurred and acts accordingly.

The following segment illustrates how **bm_$wait** handles this task:

```
if (!bmcb.flags.done) {
        pbu_timeout = timeout;    /* value in seconds */
        pbu_timeout = (pbu_timeout == 0) ? (3600 * 1000) :
                      (pbu_timeout * 1000);

        /*
         * We want the ability to handle any faults through the
         * return value of pbu_$wait, when we enable again, we
         * will get the fault.  If we did not inhibit before the
         * pbu_$wait call, and we received a fault, we would not
         * be able to cleanup (unmap and unwire buffer) since we
         * would be blasted back to the shell or the last fault
         * handler.
         */

        pfm_$inhibit();                    /* inhibit faults */
        index = pbu_$wait(bmcb.pbu_unit_number,
                          pbu_timeout /* number of
                                         milliseconds to wait */
                          true,       /* true means allow quits
                                         while waiting */
                          status);

        if (status->all != status_$ok) {   /* he didn't like
                                              something */
                status->s.fail = 1;
                return;
        }
}
else index = 0;                     /* transfer already complete */

switch (index) {
case 0:
        /* The operation completed. Get the ending status and
         * length transferred for the caller.
         */

        bm_status->all = bmcb.status.all;
        if (bmcb.status.all == bm_$sio_error)
                status->all = bmcb.sio_status.all;
        else if (bmcb.status.all != bm_$status_ok)
                status->all = bm_$io_error;
        *rem_len = bmcb.rem_len;          /* residual count */
        break;
case 1:
        /* the operation did not complete in time. */
        status->all = bm_$timeout;
        break;
case 2:
```

```
                /*
                 * the user typed control-q while we were waiting. Note:
                 * the standard system fault catcher will blast us
                 * directly back to shell command level, so we'd never
                 * get here. But just in case the fault catcher chooses
                 * to ignore the quit, we'll handle it.
                 */
                status->all = bm_$quit_during_wait;
                break;
        default:
                printf("Invalid pbu_$wait index value, %d\n", index);
        }
```

Table 6-1 shows how **pbu_$wait** responds to asynchronous faults (quit faults), depending on whether asynchronous faults are inhibited or enabled and whether errors are handled by the cleanup handler or by the fault handler.

Table 6-1. pbu_$wait Actions When Asynchronous Faults Are Inhibited/Enabled

Handler Response	**Asynchronous Faults Inhibited**	**Asynchronous Faults Enabled**
Cleanup Handler Response	Does not handle fault, but returns indication that quit fault did occur	Executes cleanup handler
Fault Handler Response	Does not handle fault, but returns indication that quit fault did occur	Executes fault handler; if fault handler returns control to the interrupted code, **pbu_$wait** returns an indication that a quit fault occurred

6.3.2 Using pbu_$get_ec and ec2_$wait

A device driver or one of its applications may want to wait for more events than device interrupt, time-out, or quit fault. For example, an application may be simultaneously handling a peripheral device and fielding commands from the terminal. In this case, the application uses system routines **pbu_$get_ec** and **ec2_$wait** to wait for a variety of events, including device interrupt.

The driver routine or application specifies the following as arguments to **pbu_$get_ec**:

- The unit number of the device.
- A key that indicates which eventcount to get. Currently, the key must be **pbu_$get_device_ec**.

The **pbu_$get_ec** routine returns a value that identifies the device's eventcount. Drivers need to call **pbu_$get_ec** only once during the time the device is acquired; they should store the returned pointer for subsequent use. However, no errors occur if **pbu_$get_ec** is called more than once.

Next, the application or driver routine constructs two lists:

- A list of identifiers for any eventcounts to be waited on, including the identifier returned by **pbu_$get_ec**
- A list of satisfaction values for each eventcount

The routine (or application) specifies these lists as parameters to **ec2_$wait**. This system routine waits until one of the eventcounts reaches its corresponding satisfaction value and returns an index value that indicates which eventcount was satisfied.

The following example shows how to wait for device interrupt with **ec2_$wait**. (For a description of **ec2_$wait** and the other eventcount routines, refer to the *Domain/OS Calls Reference* manual.)

```
#nolist;
#include <apollo/base.h>
#include <apollo/pbu.h>
#list;

boolean                               /* true => device advance */
dev_$wait(uec, uecval, st)            /* false => user's ec advanced */
ec2_$ptr_t *uecp;                     /* user's eventcount */
long       &uecval;                   /* user's eventcount value */
status_$t  *st;
{
    int        i;
    ec2_$ptr_t ecp;

    ec2_$ptr_t ec_ptr_list[2];        /* ec ptr list and value arrays */
    long       ec_val_list[2];

    /* get the device ec */
    pbu_$get_ec(unit, pbu_$get_device_ec, &ecp, st);
    if (st->all != status_$ok)
        return(false);                /* no eventcount */
    /* wait for either the device or the user's ec to be advanced */
    ec_ptr_list[0] = ecp;
    ec_val_list[0] = ec2_$read(*ecp) + 1;
    ec_ptr_list[1] = uecp;
    ec_val_list[1] = uecval;

    /* if the operation is already done, don't wait, just
       return success */
    if (op_already_done)
        return(true);

    i = ec2_$wait(ec_ptr_list, ec_val_list, st);
    if (st->all != status_$ok)
        return(false);              /* no eventcount */

    return(i-1 == 0);               /* ec2_$wait returns 1..n */
}
```

In the example, op_already_done is a flag that the user-written interrupt routine sets when an interrupt is received from the device. The example procedure checks the flag after it calculates the eventcount value to wait for. In general, whenever a program waits for an eventcount, it must provide a method (other than the eventcount itself) by which it can identify whether or not the desired event has already occurred.

> NOTE: The variable returned by **pbu_$get_ec** is an **ec2_$ptr_t**, which is *not* a normal pointer. Do not assume that it contains a virtual address.

The driver can go about other business while an I/O operation is in progress. In this case, the driver should return an eventcount for the application to wait upon while the driver is off doing something else.

6.4 Performing Cleanup Functions

User-written device drivers can optionally supply a cleanup routine to perform device-specific cleanup functions before a device is released. The routine **pbu_$release** obtains the entry point of the cleanup routine from the DDF and calls the routine during device release. The cleanup routine in **bm_example** is called **bm_$cleanup**; refer to Appendix E, Section E.2 (C) and Appendix F, Section F.3 (Pascal).

Functions performed by the cleanup routine include:

- Ensuring that no I/O is in progress when the device is released. The routine can perform this function either by waiting for any outstanding device I/O to complete or aborting any outstanding I/O.
- Clearing any pending interrupts from the device.
- Deciding whether or not to cancel the release process.
- For PC AT compatible device drivers, ensuring that the last call to **pbu[2]_$dma_start** had a corresponding call to **pbu[2]_$dma_stop**.
- Releasing any acquired I/O resources.

The cleanup routine is bound with the other call-side routines.

The cleanup routine is called by GPIO software and must, therefore, conform to the following calling sequence (shown in C and Pascal):

Synopsis (C)

```
void cleanup_routine(
        pbu_$unit_t &unit,
        boolean     &force_flag,
        status_$t   *status
        );
```

Synopsis (Pascal)

```
procedure cleanup_routine(
        in  unit:       pbu_$unit_t;
        in  force_flag: boolean;
        out status:     status_$t
        );
```

Description

force_flag	A Boolean value that indicates whether or not the cleanup routine can abort the device release operation. If this parameter is set to "true", the device is released regardless of the status returned by the cleanup routine. If this flag is set to "false", the cleanup routine can abort the release procedure by returning a nonzero status code. Upon receipt of the status, **pbu_$release** aborts device release and returns to its caller. This flag is the same as the *force_flag* parameter for **pbu_$release**.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format.

Chapter 7

Transferring Data

Data can be transferred between the application and the device by means of DMA, memory–mapped I/O, or programmed I/O. The method you use depends on the kind of controller your driver supports. This chapter describes how to implement each method in your driver, using GPIO routines.

> **NOTE:** Apollo provides two kinds of calls, **pbu_$** and **pbu2_$**, for several GPIO operations. The **pbu2_$** routines take addresses and lengths specified as 4–byte integers rather than 2–byte integers. When referring to either kind interchangeably, we use the term **pbu[2]_$***routine_name*.

MULTIBUS users should take note that drivers running on nodes equipped with a 16–bit MULTIBUS can also use **pbu2_$** routines; however, on nodes with a 20–bit MULTIBUS, the driver must *not* call a **pbu_$** routine for which there is a **pbu2_$** counterpart. For this reason, it may be convenient always to use the **pbu2_$** routine, where one is available, so that the same driver can run on either 16–bit or 20–bit MULTIBUS nodes. Also, note that if your driver specifies a 20–bit MULTIBUS address and is running on a node with a 16–bit MULTIBUS, the GPIO routines will return an error indication because the 16–bit MULTIBUS supports only 16–bit MULTIBUS addresses.

If you are writing a driver for an PC AT compatible and VMEbus device, you must use **pbu2_$** routines where they are available. The one exception to this rule concerns **pbu[2]_$dma_start** and **pbu[2]_$dma_stop** routines. Drivers running on the DN3000 use **pbu_$dma_start** and **pbu_$dma_stop**. Drivers running on the DN4000 use **pbu2_$dma_start** and **pbu2_$dma_stop**. The exception to the exception is that if you are writing a driver for a device that can exert bus mastership on a DN4000, you must use **pbu2_$dma_start** and **pbu2_$dma_stop**.

NOTE: Unlike Pascal, the C programming language is case-sensitive; therefore, all system procedure names (such as GPIO routines) must be lowercase, which is consistent with their appearance in the system insert files. However, any global names in C that are accessed by GPIO routines are case-sensitive.

7.1 DMA Transfers

A DMA transfer to or from processor memory occurs when a DMA controller makes memory references to bus address space. Apollo supports DMA transfers on the MULTIBUS, PC AT compatible bus, and VMEbus. Differences in the way you implement a DMA transfer in your driver depend not so much on the kind of bus as on whether your node's I/O hardware includes an I/O map. All workstations that support the MULTIBUS are equipped with the I/O map. For the PC AT compatible bus, the DN3000 doesn't have the I/O map and the DN4000 does. If your workstation has the I/O map, refer to Subsection 7.1.1; otherwise, refer to Subsection. 7.1.2.1. You should also refer to Chapter 1, Chapter 2, and Chapter 3 of this manual for additional bus-specific information.

7.1.1 Using the I/O Map to Perform DMA Transfers

The I/O map translates memory references to bus address space into processor memory references. Before the controller can initiate memory references, the device driver must establish an association between the pages of processor memory and the pages of bus address space. This is referred to as *mapping an I/O buffer*.

The process of mapping an I/O buffer consists of the following:

- Allocating bus address space for the controller
- Wiring the pages of the I/O buffer
- Setting up the I/O map to establish mapping between processor memory and bus address space

7.1.1.1 Allocating Bus Address Space

All controllers use the same bus address space to access processor memory.

NOTE: The address space is 64 KB for 16-bit controllers, 1024 KB for 20-bit controllers, and 16 MB for 24-bit controllers.

Since I/O buffers concurrently in use by controllers must not overlap in bus address space, the device driver must ensure against overlap by allocating a section of bus address space for the controller. You use the GPIO routine **pbu[2]_$allocate_map** to allocate the section for the controller. The driver specifies the length of the I/O buffer to **pbu[2]_$allocate_map**; the routine locates a portion of the I/O map that matches the given length and returns the address of the first page of bus memory allocated to the buffer.

If another device is active when the driver calls **pbu[2]_$allocate_map**, either the requested amount of I/O map space may be unavailable or a hard-wired bus address may already be in use (see Chapter 6, Subsection 6.1.4). In this case, the driver has several choices:

- Wait for an interval and then retry the operation
- Request a smaller amount
- Report the error to the application program
- Inform the interactive user that the requested system resources are unavailable

The following call to **pbu_$allocate_map** (from the initialization routine of **bm_example_c**) allocates an area of the I/O map that corresponds to the largest block (32 KB) the driver ever reads or writes. The constant bm_$block_len is declared as having a value of 32768; bmcb.bm_iova contains the start of the allocated area of bus address space.

```
bmcb.bm_iova = pbu_$allocate_map(
           bmcb.pbu_unit_number, /* number of this pbu device */
           bm_$block_len,     /* maximum block size we'll use */
           false,               /* don't need a specific iova */
           0,                   /* forced iova would go here */
           status);             /* returned status */
```

7.1.1.2 Wiring I/O Buffers

A buffer is wired when it is permanently resident in processor memory and is, therefore, unavailable to the MMU's paging operations. Device drivers must wire their I/O buffers because the I/O map cannot handle the movement or absence of pages during an I/O operation.

A device driver wires an I/O buffer by calling the routine **pbu[2]_$wire**, specifying the buffer to be wired and its length. A page that is part of a wired buffer cannot be wired again. If a page of the requested buffer is already wired, **pbu[2]_$wire** returns an error indication to the driver.

The bm_command routine in **bm_example_c** calls **pbu_$wire** just before sending the read or write command to the routine, as follows:

```
bmcb.bufaddr = buffer;      /* save address of buffer */
bmcb.buflen = len;          /* save length of buffer */

pbu_$wire(bmcb.pbu_unit_number, /* number of this pbu unit */
          (void *)buffer,       /* buffer to wire */
          bmcb.buflen,          /* length to wire (in bytes) */
          status);              /* returned status */

if (status->all != 0) {         /* give up if something goes
                                   wrong */
        status->fail = 1;
        return;
}

bmcb.flags.buffer_wired = 1; /* remember we wired the buffer */
```

The size of a node's main memory determines the maximum number of 1024–byte pages that can be wired by all drivers in the system. To determine the approximate maximum number of wired pages for your node, subtract 256 from the number of pages of memory that the node has. For example, for a node with 1 MB of main memory, 1024 pages minus 256 (pages) equals 768, so drivers must wire fewer than 768 pages.

The driver can also wire an I/O buffer by defining a static storage area in the interrupt routine and copying data to it or from it for I/O. If the storage area is allocated in the interrupt module, it is wired by virtue of being allocated in the interrupt side, which is itself wired; therefore, no call to **pbu[2]_$wire** need ever be made.

For timing considerations in wiring and unwiring an I/O buffer, refer to Appendix D, Section D.3.

7.1.1.3 Setting Up the I/O Map

After the driver has allocated pages of bus address space for the buffer and wired the buffer into processor memory, it must establish the mapping between the buffer and the pages of bus address space by calling the GPIO routine **pbu[2]_$map**. This routine takes three arguments:

- The I/O buffer
- The I/O buffer's length
- A bus address within any page of the area allocated by **pbu[2]_$allocate_map**

The **pbu[2]_$map** routine establishes the displacement within bus address space for the buffer and returns an address that corresponds to the start of the buffer.

If the buffer you want to map is permanently wired, you can call **pbu[2]_$map** in the initialization routine, just after calling **pbu[2]_$allocate_map**; otherwise, you should call it in one of the command-processing routines or in the start I/O routine. In the following example (from **bm_example_c**), **pbu_$map** is called in the start I/O routine (bm_$sio), just before touching the controller's command register. The return value (bmcb.csr_ptr.c) is the buffer's address, which is written to the controller's address register:

```
bmcb.csr_ptr.c->iova = pbu_$map(bmcb.pbu_unit_number, /* number
                                    of this pbu unit */
        (void *)bmcb.bufaddr, /* virtual address of buffer */
         bmcb.io_len,         /* length of buffer */
         bmcb.bm_iova,  /* iova we got from pbu_$allocate_map */
         status);             /* returned status */

if (status->all != 0)
        return;
```

7.1.1.4 Preallocating I/O Resources

A device driver does not need to allocate and deallocate I/O map entries for each I/O operation. Instead, when it initializes the device, the driver can allocate a portion of the I/O map that corresponds to the largest buffer that will be used during I/O transfers. The driver can map buffers via the allocated I/O map entries until the device is released.

Similarly, the device driver can "permanently" wire and map an I/O buffer at device initialization for the duration of driver execution. During device initialization, the initialization routine can call the routines **pbu[2]_$allocate_map**, **pbu[2]_$wire**, and **pbu[2]_$map** to establish a correspondence between this preallocated buffer and a section of bus address space. The routine saves the address returned by **pbu[2]_$map**. To perform a DMA transfer, the driver copies data into the preallocated buffer, loads the address returned by **pbu[2]_$map** into the controller's DMA registers, and initiates the transfer. Appendix D, Section D.4 discusses some performance advantages of a permanently wired buffer.

Another way to preallocate I/O resources is to define a preallocated buffer in the interrupt side of the driver, as described in Subsection 7.1.1.2, "Wiring I/O Buffers."

7.1.1.5 Dynamic Resource Allocation

Drivers for applications that move data directly to or from a file–system object mapped into processor address space usually wire and unwire a buffer for each I/O operation. For example,

```
map file into address space;
i := 0;
WHILE i < number_of_pages_in_file DO BEGIN
           wire pages i to i+n-1;
           do i/o;
           unwire pages i to i+n-1;
           i := i+n;
           END;
```

Note that the driver need not wire any pages used by the interrupt routine; they are wired when the driver is installed into user–process address space during device acquisition. Sometimes, however, the device driver may attempt to wire a buffer in the data$ section of an application program that shares a page with the data$ section of the interrupt routine. Because this page has already been wired, **pbu[2]_$wire** returns an error. To prevent this error, the driver can

- Place the buffer in dynamic storage (the stack)
- Place the buffer in a mapped object (which will always be page aligned)
- Declare a dummy array of one page immediately following the buffer declaration

7.1.1.6 Scatter–Gather Operations

A scatter–gather I/O operation consists of reading (scattering) or writing (gathering) a single block of data in bus address space to or from discontiguous buffers in processor address space. For example, when the operating system reads a Domain disk block, it places the 32–byte header in supervisor memory and the 1024 bytes of data elsewhere in memory.

The **pbu[2]_$map** routine can be used to implement limited forms of scatter–gather by observing Rules 1, 2, and 3:

1. The *end* of the first section of data to be read or written must fall on a page–aligned boundary.
2. The driver should map each subsequent section to a bus address that is one page higher than the page address of the previous section.
3. All blocks of data following the first section must be an integral number of pages in length and must *start* on page–aligned boundaries. (The last section need not end on a page boundary.)

The following example shows how to apply the rules when mapping a block of data to discontiguous buffers. In this example, the block has a 5C–byte header and 1A0 bytes of data.

First, the driver calls **pbu[2]_$allocate_map**, which reserves an area of the I/O map and returns the address of the first available page in bus memory (in this example, 3000).

Next, the driver calls **pbu[2]_$map**, specifying iova 3000, the length 5C, and buffer address 2A9FA4 (that is, the start of the area in processor address space where the header is to be transferred). The buffer address is obtained by subtracting the length of 5C from a page–aligned address in processor address space (2AA000), giving the starting address 2A9FA4. This procedure satisfies Rule 1 by ensuring that the first section ends on a page–aligned boundary. The **pbu[2]_$map** routine returns the header's starting address (33A4) in bus address space.

The 1A0 bytes of data are to be transferred to a buffer at address 2E4400, thus satisfying Rule 3, which requires each subsequent section to start on a page boundary. The driver calls **pbu[2]_$map**, specifying iova 3400, the length of the data 1A0, and the address 2E4400. The **pbu[2]_$map** routine returns a bus address 3400 for the data, in accordance with Rule 2, which requires the driver to map each subsequent block to a bus address that is one page higher than the bus address of the previous block.

Figure 7–1 illustrates this example of mapping to discontiguous buffers.

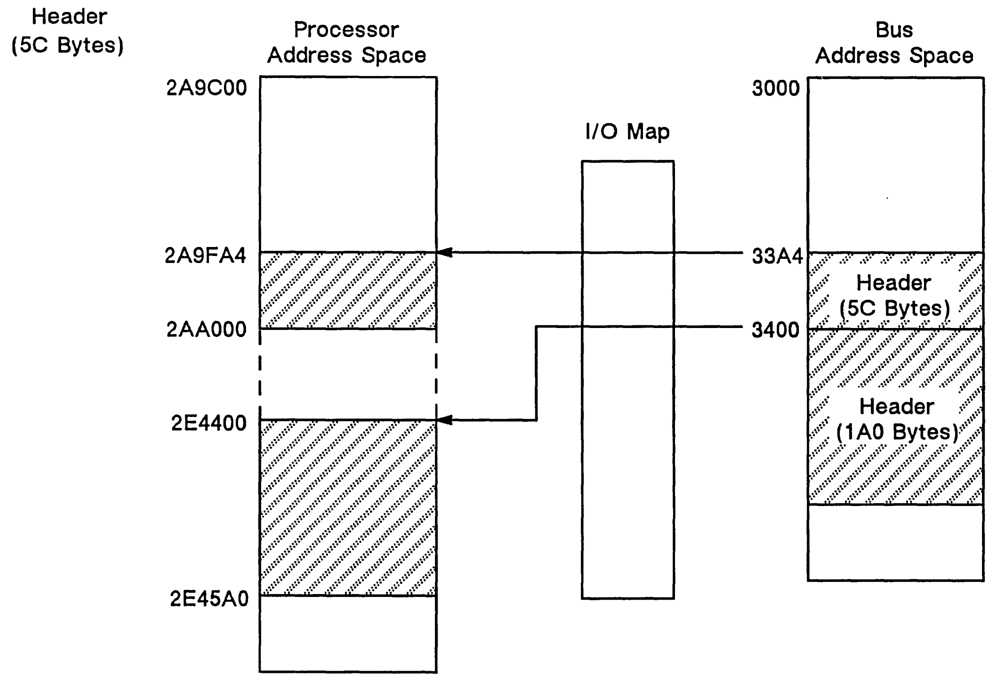


Figure 7–1. Mapping Discontiguous Buffers

7.1.2 Starting and Stopping a DMA Operation on the PC AT Compatible Bus

You must use the **pbu2_$** DMA calls for bus-master PC AT compatible devices with an I/O map. For bus-master PC AT compatible bus devices without an I/O map, we prefer that you also use the **pbu2_$** DMA calls. Although the nonbus-master device code may use either **pbu2_$** or **pbu_$** DMA calls to perform DMA operations, we recommend that the **pbu2_$** calls be used for the following reasons:

- **pbu2_$** DMA calls will work on machines with or without I/O map hardware.
- Even if I/O map hardware is present, drivers that use the **pbu_$** DMA calls are still restricted to DMA operations with a maximum length of one page.
- If I/O map hardware is present, drivers that make **pbu2_$** DMA calls have better control over and can more efficiently use the I/O map resources, and may perform DMA operations of more than one page.

For drivers that wish to use **pbu_$** DMA calls on machines without I/O map hardware and **pbu2_$** DMA calls on machines with I/O map hardware, use the **pbu_$get_info** call to determine the presence of an I/O map.

```
#nolist
#include <apollo/base.h>
#include <apollo/pbu.h>
#list

void device_$init(
            pbu_$unit_t        &unit,          /* unit number */
            pbu_$ddf_ptr_t     &ddf_ptr,       /* pointer to ddf */
            pbu_$csr_page_ptr_t &csr_ptr,      /* pointer to csr page */
            status_$t          *status)        /* returned status */
{

   pbu_$info_t     info;

   .......

   /* determine configuration */
   pbu_$get_info(sizeof(info), &info, status);
   if (status->all != status_$ok) return;

   /* check for iomap existence for bus my device is on */

   if (info.iomap_types & pbu_atbus_iomap) {    /* is there an iomap for
                                                   atbus? */

       /* yes, machine has an iomap */
       .......
   }
   else {
        /* machine has no iomap */
        .....
    }
    .......
}
```

7.1.2.1 DMA Transfers Without the I/O Map

In drivers for nonbus–master devices, **pbu_$dma_start** and **pbu_$dma_stop** must surround each DMA operation, whether successful or not. The **pbu_$dma_start** routine prepares DMA hardware for the controller's operation. After the driver calls **pbu_$dma_start**, the controller can begin its operation. When the controller indicates that the operation is completed, the driver calls **pbu_$dma_stop** to get status from DMA hardware to ensure that the hardware completed its share of the operation as well. The driver must call **pbu_$dma_stop** even if the controller reports an error. The driver may ignore the status returned by **pbu_$dma_stop**, but if the controller had a problem, it is likely that the DMA operation did not run to completion. The call to **pbu_$dma_stop** must, in any case, be made so that software can reset its knowledge of who is using the DMA channel.

It is important that these two calls surround each DMA operation. If you make a call to **pbu_$dma_start** without a subsequent call to **pbu_$dma_stop**, the channel you specified in **pbu_$dma_start** becomes unavailable for any additional DMA activity; the next time you attempt to call **pbu_$dma_start**, you will get a REQUESTED DMA CHANNEL IN USE error message. If you get this message, however, you can call **pbu_$dma_stop** to release the channel.

Use the following calls in the sequence in which they appear. The sequence of calls made by a driver using **pbu2_$dma_start/stop** for nonbus–masters follows:

- **pbu2_$wire** Wires the buffer
- **pbu_$dma_start** Sets up and starts DMA operation
- Device–specific code to activate DMA operation
- **pbu_$dma_stop** Stops DMA operation
- **pbu2_$unwire** Unwires the buffer

Drivers for bus-master devices must call **pbu[2]_$dma_start** once, specifying the **pbu_dma_cascade** option. This option reserves the DMA channel and provides bus arbitration. The **pbu[2]_$dma_stop** routine must be called when the device is released.

Use the following calls in the sequence in which they appear. The sequence of calls made by a driver using **pbu_$dma_start/stop** for bus-masters follows:

- **pbu_$dma_start** Sets processor's DMA hardware to cascade mode so that the device can use its own DMA hardware
- **pbu2_$wire** Wires the buffer
- Device specific code to activate DMA operation
- **pbu2_$unwire** Unwires the buffer
- **pbu_$dma_stop** Takes DMA processor's DMA hardware out of cascade mode

Unless the device itself supports scatter-gather operations, DMA transfers without the I/O map are limited to 1024 bytes of data per operation and must not cross page boundaries. (Methods of aligning a buffer on a page boundary are discussed in Chapter 6, Subsection 6.1.4, and Chapter 10, Subsection 10.1.2.1.) If your device has its own scatter-gather hardware, your driver must wire its I/O buffer by calling **pbu_$wire_special**, specifying as arguments the buffer to be wired and its length. The routine returns a list of physical addresses, which the driver sends to the device. Refer to Appendix B, Section B.2 for a description of this GPIO routine.

PC AT compatible devices that can request bus mastership must also call **pbu_$wire_special**, specifying as arguments the buffer to be wired and its length. (Drivers for nonbus-master devices must call **pbu2_$wire**.)

Drivers designed to run only on the DN3000 should call **pbu_$dma_start** and **pbu_$dma_stop**. How you use these calls depends on whether your driver supports a bus-master or nonbus-master device.

> NOTE: If you are designing your driver to run on the DN4000 and you wish to take advantage of its I/O map, you must use **pbu2_$dma_start** and **pbu2_$dma_stop** (see Subsection 7.1.1).

The following program segments are from a DN3000 driver for a nonbus-master device. Included here are parts of the call-side transfer routine (dma_data), which initiates the DMA operation, and the interrupt routine (dev_$int), which services device interrupts and stops the DMA operation.

The driver assumes that the data to be transferred is page aligned, but it does include a check to determine if the amount of data to be transferred exceeds the 1–KB limit per DMA operation. If the amount of data exceeds 1 KB the interrupt routine restarts the DMA operation for the next 1–KB block of data and continues to do so until all of the data is transferred.

First, the transfer routine:

```
PROCEDURE dma_data (                            { DMA data to/from the
                                                  controller }
                 IN  cb_ptr:    dev_cb_ptr_t;{ control block pointer }
                 IN  dir_read: boolean;     { a flag:
                                                  True  = read data
                                                          from device
                                                  False = write data to
                                                          device }


                 IN  va:        univ_ptr;     { virtual address
                                                  (pointer) to the
                                                  buffer to read/write }
                 IN  len:       pinteger;     { length to dma in bytes}
                 OUT status:    status_$t     { return status }
                 );


VAR
      dma_buf_ptr:     ^string;
      dma_dir:   pbu_$dma_direction_t;
      st:        status_$t;
      cnt:       pinteger;
begin
      with cb_ptr^:cb, cb.csr_ptr^:csr do begin
         cb.dma_complete := false;          { no DMA started yet }

         { Enable the DMA request on the device before calling
           start_dma.  This must be done because the DMA line will float
           unless the dma enable bit is set. }

         cb.dev_control:= cb.dev_control +
                   [dma_ienable, dma_enable]; { DMA interrupt enable,
                                                DMA enable }
         csr.dev_control := cb.dev_control;   { write driver's copy to
                                                csr page }

         if dir_read then cb.dma_dir := pbu_dma_read;  { if true, DMA
                                                         read}
         else cb,dma_dir := pbu_dma_write;            { if false, DMA
                                                         write}
        { Check that that the data to DMA is in bytes_per_page chunks. }
```

```
         cb.dma_buf_ptr := va;
         if cnt > bytes_per_page then begin;
              cb.dma_remainder := cnt - bytes_per_page;
              cnt := bytes_per_page;
              end
         else cb.dma_remainder := 0;

         { Call PBU routine to setup and enable DMA controller on CPU
           board. }

         pbu_$dma_start (cb.pbu_unit, cb.dma_chan, cb.dma_dir,
                         cb.dma_buf_ptr^, cnt, [], status);
         if status.code <> status_$ok then goto dma_fail;

         { Wait for the DMA to complete. The interrupt routine will call
           pbu_$dma_stop if DMA goes to completion. }

         while not cb.dma_complete do
             if (pbu_$wait (cb.pbu_unit, dev_timeout, true, status)<>0)
                 then exit;
         if not cb.dma_complete then begin {interrupt did not happen...}
             status.all := dev_$dma_timeout; { ... DMA timed out, so
                                                abort DMA. }
  dma_fail:  discard(pbu_$dma_stop (cb.pbu_unit, cb.dma_chan, st));

             cb.dev_control := cb.dev_control -
                      [dma_ienable, dma_enable]; { turn off device's DMA
                                                   enables }
              csr.dev_control := cb.dev_control; { write driver's copy
                                                   out to csr page }
         end;  { if not cb.dma_complete }
      end;   { with cb_ptr^, cb.csr_ptr^ }
      return;
  end { dma_data };
```

Next, the interrupt routine (some device-specific code is omitted at the beginning of the routine that checks for a command-complete interrupt):

```
FUNCTION dev_$int (in unit: pbu_$unit_t): pbu_$interrupt_return_t;  {
device interrupt
                                                          routine }

  var
      st:         status_$t;
  begin
      dev_$int := [pbu_$interrupt_advance,
                   pbu_$interrupt_enable];       { default return }
      with dev_$cb[0]:cb, cb.csr_ptr^:csr do begin
       .
       .
       .
      { Check for DMA-complete interrupt.  It is necessary to disable
        the DMA channel before disabling DMA on the device, because as
        soon as DMA is disabled on the device, the DMA request lines
        will float, causing spurious DMA cycles if the DMA channel were
        still enabled. }

           if csr.dev_status.dma_done then begin
               discard(pbu_$dma_stop(cb.pbu_unit, cb.dma_chan,
                       cb.dma_stop_stat));



           { Make sure we don't try to DMA more than 1K at a time.
             cb.dma_remainder is initialized in dma_data and is updated
             here. }

                if cb.dma_remainder <> 0 then begin      { more to do }
                     dev_$int := [pbu_$interrupt_enable];

                     { adjust the buffer pointer to the
                       bytes_per_page block }

                     cb.dma_buf_ptr :=
                            univ_ptr(integer32(cb.dma_buf_ptr) +
                                                 bytes_per_page);
                     { check to see if we have more than bytes_per_page
                       left to transfer }

                     if cb.dma_remainder > bytes_per_page then begin
                          cnt := bytes_per_page;
                          cb.dma_remainder := cb.dma_remainder -
                                                  bytes_per_page;
                          end
                     else begin
                          cnt := cb.dma_remainder;
                          cb.dma_remainder := 0;
                          end;

                   { start up the DMA channel for the next
                     bytes_per_page block }
                    cb.dev_control  := cb.dev_control -
                           [dma_enable,dma_ienable];  { disable DMA
                                                        interrupt and
                                                        DMA lines }
                     csr.dev_control := cb.dev_control;  { copy to CSR
                                                           page }

                     cb.dma_complete := true;     { flag dma complete }
                     end; { if - then - else cb.dma_remainder <> 0 }
                end; { if csr.dev_status.dma_done }
      end;    { with dev_$cb[0], cb.csr_ptr^ }
end; { dev_$int }
```

7.1.2.2 DMA Transfers With the I/O Map

Drivers that are designed to run on a PC AT compatible bus machine with an I/O map and take advantage of its I/O map use the same calls as drivers for MULTIBUS devices. In addition, such drivers also call **pbu2_$dma_start** and **pbu2_$dma_stop**. How you make these calls depends on whether your driver supports a bus-master or nonbus-master device.

In drivers for nonbus-master devices, **pbu2_$dma_start** and **pbu2_$dma_stop** must surround each DMA operation, whether successful or not. The **pbu2_$dma_start** routine prepares DMA hardware for the controller's operation. After the driver calls **pbu2_$dma_start**, the controller can begin its operation. When the controller indicates that the operation is completed, the driver calls **pbu2_$dma_stop** to get status from the DMA hardware to ensure that the hardware completed its share of the operation as well. The driver must call **pbu2_$dma_stop** even if the controller reports an error. The driver may ignore the status returned by **pbu2_$dma_stop**, but if the controller had a problem, it is likely that the DMA operation did not run to completion. The call to **pbu2_$dma_stop** must, in any case, be made so that software can reset its knowledge of who is using the DMA channel.

It is important that these two calls surround each DMA operation. If your driver makes a call to **pbu2_$dma_start** without a subsequent call to **pbu2_$dma_stop**, the channel you specified in **pbu2_$dma_start** becomes unavailable for any additional DMA activity; the next time the driver attempts to call **pbu2_$dma_start**, you will get a REQUESTED DMA CHANNEL IN USE error message. If you get this message, however, you can call **pbu2_$dma_stop** to release the channel.

Use the following calls in the sequence in which they appear. The sequence of calls made by a driver for a nonbus-master device follows:

- **pbu2_$allocate_map** Allocates an area of the I/O map
- **pbu2_$wire** Wires the buffer
- **pbu2_$map** Maps the buffer to bus memory space
- **pbu2_$dma_start** Sets up and starts the DMA operation
- Device-specific code to activate DMA operation
- **pbu2_$dma_stop** Stops the DMA operation
- **pbu2_$unmap** Unmaps the buffer
- **pbu2_$unwire** Unwires the buffer
- **pbu2_$free_map** Releases the previously allocated area of the I/O map

Drivers for bus–master devices must call **pbu2_$dma_start** once, specifying the **pbu_dma_cascade** option. This option reserves the DMA channel and provides bus arbitration. The **pbu2_$dma_stop** routine must be called when the device is released.

Use the following calls in the sequence in which they appear. The sequence of calls made by a driver for a bus–master device follows:

- **pbu2_$dma_start** Sets the processor's DMA hardware to cascade mode so that the device can use its own DMA hardware
- **pbu2_$allocate_map** Allocates an area of the I/O map
- **pbu2_$wire** Wires the buffer
- **pbu2_$map** Maps the buffer to bus memory space
- Device–specific code to activate DMA operation
- **pbu2_$unmap** Unmaps the buffer
- **pbu2_$unwire** Unwires the buffer
- **pbu2_dma_stop** Takes the DMA hardware out of cascade mode
- **pbu2_$free_map** Releases the previously allocated area of the I/O map

7.1.3 Releasing I/O Resources After a DMA Transfer

The driver uses GPIO routines to release I/O resources following the completion of a DMA transfer. Subsections 7.1.3.1 and 7.1.3.2 describe what routines to call and how to use them.

> NOTE: If you are not designing your driver to run only on the DN4000, only Subsection 7.1.3.2 is pertinent to you.

7.1.3.1 Deallocating the I/O Map

Because each device can have only one area of the I/O map allocated to it at a time, the device driver must call **pbu[2]_$free_map** to deallocate I/O map entries before it can call **pbu[2]_$allocate_map** again. However, the driver need not allocate the I/O map dynamically (see Subsection 7.1.1 for more information about I/O resource allocation).

7.1.3.2 Unwiring the I/O Buffer

Device drivers that have wired their buffers by using **pbu[2]_$wire** or **pbu_$wire_special** must unwire them with **pbu[2]_$unwire** unless they are going to use them again for another I/O operation. If the buffer is a file-system object into which data has been read, the driver should ensure that the data is saved when the file is closed by

- Copying the buffer to another area in memory before unwiring it, or
- Setting to "true" the modify_flag argument to **pbu[2]_$unwire** so that **pbu[2]_$unwire** marks each page of the buffer as having been modified before unwiring it

7.1.4 Releasing I/O Resources During Faults

If a device driver has allocated I/O resources and a synchronous or asynchronous fault occurs, the allocated resources (I/O map entries, wired buffers, or mapped memory) are not deallocated unless the application program or driver establishes a cleanup handler or the process terminates.

The application or driver uses the system function **pfm_$cleanup** to establish its own fault handling routine. The device driver should also contain a cleanup routine that deallocates I/O resources and disables the device. The driver should monitor the allocation of the following I/O resources:

- The area of the I/O map that is allocated
- Locations and sizes of wired buffers
- Bus memory addresses and sizes of mapped buffers

When a fault occurs, the application's fault handler, as one of its functions, calls the driver cleanup routine to release any allocated I/O resources.

If the initialization routine contains the entire application, the application need not establish a fault handler. The **pbu_$acquire** routine establishes a fault handler before calling the initialization routine, so that any fault during initialization causes the device to be released, thereby releasing any allocated resources.

7.2 Memory–Mapped Transfers

A memory–mapped controller contains on–board memory that can store data received from external devices. However, the controller itself does not transfer the blocks of data to processor address space, as it would if it performed DMA; instead, the device driver moves the data to or from controller memory.

The driver in **/domain_examples/gpio_examples/threecom_example** supports a memory–mapped controller.

Before a device driver can refer to controller memory, it must associate the area of controller memory with an area of processor address space. The way this mapping is accomplished depends on the node's bus.

- Device drivers running on a node equipped with a 16–bit MULTIBUS call GPIO routines **pbu_$map_controller** and **pbu_$unmap_controller** to map and unmap controller memory to and from processor address space.
- Drivers for 20–bit controllers running on nodes with a 20–bit MULTIBUS call GPIO routines **pbu2_$map_controller** and **pbu2_$unmap_controller** to map and unmap controller memory to and from processor address space.
- Drivers for VMEbus and PC AT compatible devices call GPIO routines **pbu2_$map_controller** and **pbu2_$unmap_controller** to map and unmap controller memory to and from processor address space.

NOTE: If a DN5xx workstation, DSP80, or DSP90 with a 20–bit MULTIBUS is fully configured with 3 MB of memory, only 512 KB of the MULTIBUS address space is available for memory–mapped operations. This restriction does not apply to the DN5xx–T family.

7.2.1 Referencing Controller Memory

Certain restrictions apply when referencing controller memory on the MULTIBUS, VMEbus, and PC AT compatible bus.

For the MULTIBUS:

- Controller memory must be page aligned and must occupy only the first 32 KB of MULTIBUS memory space on nodes with a 16-bit MULTIBUS and 1 MB on nodes with a 20-bit MULTIBUS. (For more controller configuration information, see Chapter 1, Section 1.3.)

- The area of MULTIBUS memory space occupied by the controller memory is permanently unavailable to DMA operations by any controller.

- On the 16-bit MULTIBUS, neither the memory-mapped controller nor any other controller can use the MULTIBUS to read or write to memory on the memory-mapped controller.

  The reason for this restriction is that the I/O hardware interprets memory references on the bus as DMA references to processor memory. If the reference is a memory write, the data is transferred to both controller memory and processor memory, causing a bus time-out error if the I/O map was not set up correctly. If the reference is a memory read, the I/O hardware and the controller simultaneously become bus masters, resulting in corrupted data.

  This restriction does not apply to the 20-bit MULTIBUS.

For the VMEbus:

- Controller memory must be page aligned.

- Controller memory must lie within the area reserved for it in processor physical address space (see Table 2-1).

- The area of memory space occupied by controller memory is permanently unavailable to DMA operations by any controller.

For the PC AT compatible bus:

- Controller memory must be page aligned.

- Controller memory must occupy user-available locations in processor physical address space (see Table 3-2).

7.2.2 Mapping Controller Memory

The device driver calls **pbu[2]_$map_controller** to map controller memory to processor address space. The **pbu[2]_$map_controller** routine returns a virtual address that represents the start of the mapped area in processor address space. Any subsequent reads or writes to this area will read or write directly to controller memory. The driver can use **pbu_$read_csr** and **pbu_$write_csr** to reference the mapped memory. These routines suppress normal bus time-out generation if part of the memory is not responding.

> NOTE: The PC AT compatible bus does not generate bus time-outs, which means that you cannot use **pbu_$read/write_csr** to test for controller presence; instead, you must tweak the appropriate device register and see if it responds in a predictable fashion to determine if the device is present.

The following segment is from the initialization routine for a driver supporting a memory-mapped controller. The routine calls **pbu_$map_controller** and **pbu_$read_csr** to test if the controller is present on the bus and, if it is, to initialize it; cbp has been declared as a pointer to the driver control block.

```
with cbp^ do begin

       mem_ptr := pbu_$map_controller (pbu_unit, mem_base, mem_len,
                                           status);

      if status.all <> status_$ok then
        begin
            status.fail := true;
             return;
        end;

      { Read the status register with read_csr to see if the
        controller is really there.  }

      pbu_$read_csr (pbu_unit, mem_ptr^.csr, i, false, status);

      if status.all = pbu_$bus_timeout then
        begin
            status.all := dev_$no_controller;
             return;
        end;

      if status.all <> status_$ok then
        begin
            status.fail := true;
             return;
        end;

      flags := flags + [init];   { tell everyone we're initialized }
```

```
        { Issue a reset command to the controller, then go online.
          From here on in, we depend on dev_$cleanup to clean up if we
          get an error. }

        dev_$set_mode (unit, dev_$reset, [], status);
        if status.all <> status_$ok then return;
        dev_$set_mode (unit, dev_$online, [], status);
        if status.all <> status_$ok then return;
  end;
```

MULTIBUS users should take the following precautions when performing memory-mapped I/O:

- The **pbu[2]_$map_controller** routine makes the area of bus memory space allocated to the controller unavailable for any subsequent DMA operations. Note that the bus addresses required for the controller may already be allocated for a DMA transfer. To prevent this situation from occurring, application programs should acquire memory-mapped devices before DMA devices.

- Because the hardware has no indication that a memory-mapped controller is present until **pbu[2]_$map_controller** is called, the I/O map allocation routines may allocate, for the memory-mapped controller or for another controller, an I/O map area that overlaps the area allocated to the memory-mapped controller. As a precaution, you should configure the controller memory to occupy the high end of bus memory space, because the I/O map allocation routines allocate I/O map areas from low addresses to high addresses.

- If the driver of a memory-mapped controller needs to perform a DMA transfer, it can call **pbu[2]_$allocate_map** to allocate another area of the I/O map. However, the device driver must call **pbu[2]_$map_controller** *before* calling **pbu[2]_$allocate_map**.

7.2.3 Unmapping Controller Memory

Drivers *must* call **pbu[2]_$unmap_controller** to unmap controller memory. If the driver needs to retain an image of the controller memory, it must copy the memory to another area of processor address space before calling **pbu[2]_$unmap_controller**.

7.3 Programmed I/O

In programmed I/O, the processor transfers the data one word (or byte) at a time, testing a device register following each transfer to determine if it was complete. A device for any bus may perform programmed I/O, provided it is equipped with the necessary interface.

Writing a data–transfer routine using programmed I/O is the simplest of the three methods because there are no buffers to allocate and wire (and deallocate and unwire), no I/O map to set up, and no DMA hardware to turn on and off. However, programmed I/O is also generally the slowest because

- The rate of transfer is limited to one word or byte at a time.
- The transfer itself is under the control of software rather than hardware.
- The device must inform the processor after each transfer.

In the case of the DN3000, however, programmed I/O is appreciably faster than DMA because

- The MC68020 programmed I/O transfer rate is 12 MHz versus the specification for the PC AT compatible bus DMA transfer rate of 6 MHz.
- DMA transfers for nonbus–master devices on the DN3000 are limited to 1 KB.

Thus, given the choice, you may wish to opt for programmed I/O, especially in drivers for slow (serial lines) or fast (hard disk) buffered devices, and reserve DMA for devices of intermediate speed (floppy disk).

Chapter 8

Interrupt–Side Routines

The interrupt side differs from the call side in that all memory on the interrupt side is wired to prevent paging. How this affects what you can and cannot do with the interrupt side is the subject of Section 8.1. Not all drivers require an interrupt side. Whether or not you include one in your driver depends on whether you want the driver or the System Interrupt Handler to handle interrupts. Refer to Subsection 8.2.3 for a comparison of the way that the System Interrupt Handler processes interrupts with the way a user–written interrupt routine does. Also, refer to Appendix D, Section D.4 for interrupt–processing times. If you decide to include an interrupt routine in your driver, then the interrupt side must be bound separately from the call side (see Chapter 10, Section 10.1.2).

Included in this chapter is a description of the Start I/O (SIO) function. Although an I/O operation may be started in the call side of the driver, it must be started in the interrupt side if the interrupt routine is going to call it.

NOTE: Unlike Pascal, the C programming language is case sensitive; therefore, all system procedure names (such as GPIO routines) must be lowercase, which is consistent with their appearance in the system insert files. However, any global names in C that are accessed by GPIO routines are case–sensitive.

8.1 Interrupt Side Restrictions

The interrupt side differs from the call side because it is wired to protect the address space occupied by the interrupt routine from memory management paging operations. This means that, for drivers written in Pascal, any routine or data structure referenced by the interrupt routine must be installed and DEFINEd in the same module as the interrupt routine. As a result, the interrupt side is set up somewhat differently from the call side. (For more information about defining globals, see Appendix C, Subsection C.2.5.)

No interrupt–side routine must ever reference unwired memory, shared nonglobal memory, or global memory. This restriction applies to referencing library routines such as PGM and VFMT calls and doing reads or writes in Pascal or C. Such references could cause a page fault, thus aborting interrupt processing and generating a fault in the driver process (see Subsection 8.2.4). The only GPIO routines that an interrupt–side routine can call are **pbu[2]_$map, pbu[2]_$unmap, pbu_$device_interrupting** (which determines whether an interrupt occurred), **pbu_$advance_ec, pbu[2]_$dma_start,** and **pbu[2]_$dma_stop.**

Because any reference that an interrupt–side routine makes to globals must be resolved internally to the interrupt library, all routines and data structures referenced in the interrupt side must be allocated there. Thus, for example, you must allocate the driver control block (using the DEFINE clause, if your driver is written in Pascal) within the interrupt side in order to reference it there. The same holds true for routines. To ensure that the interrupt side makes no unresolved references, we recommend that you specify the –sys option when you bind the interrupt library. This option produces a listing of all system globals that cannot be resolved within the input object module; a successful binding should result in the message, "All globals are resolved" (see Chapter 10, Subsection 10.1.2.2).

NOTE: **pbu_$acquire, pbu_$acquire_stream,** and **aqdev** will refuse to load an interrupt library with unresolved globals.

A driver can contain several interrupt routines to handle a device that interrupts on more than one request line. However, the size of the interrupt module (the interrupt routine(s) and any other procedures bound with it) must not exceed 32 KB, including procedure, data, and debug information.

8.2 Interrupt Routine

Drivers handle interrupts by performing the following functions:

- Enabling and disabling interrupts from the device
- Waiting for interrupts from the device
- Processing (optionally) device interrupts with one or more interrupt routines

Subsections 8.2.1 through 8.2.5 discuss these functions as well as other aspects of interrupt routines.

8.2.1 Interrupt Routine Format

The interrupt routine is called by GPIO software and must, therefore, conform to the following formats:

For C:

```
pbu_$interrupt_return_t interrupt_routine (pbu_$unit_t &unit);
```

For Pascal:

```
function interrupt_routine(in unit:pbu_$unit_t):pbu_$interrupt_return_t;
```

The input parameter, *unit*, is optional (for more information, see Chapter 9, Section 9.6). The function returns a set of flags in pbu_$interrupt_return_t format that specify actions that the System Interrupt Handler is to perform. Possible values are

- **pbu_$interrupt_advance**, which directs the System Interrupt Handler to advance the device's eventcount
- **pbu_$interrupt_enable**, which directs the System Interrupt Handler to re-enable interrupts from the device

8.2.2 Enabling and Disabling Device Interrupts

On all buses except the VMEbus, a hardware interrupt mask register controls the processor's receipt of interrupts. Each bit within the register corresponds to one of the interrupt lines. Resetting the bit prevents the processor from receiving interrupts from the device. If the device requests an interrupt and the interrupt mask bit is reset, the interrupt is taken when the bit is set.

Device interrupts are automatically disabled under the following conditions:

- At system initialization (all device interrupts disabled)
- After the device is acquired
- When the System Interrupt Handler intercepts an interrupt from the device, regardless of whether the driver includes a user-written interrupt routine
- When the device is released
- During system shutdown

When the device driver requires that the processor receive interrupts from the device, it enables interrupts by calling the routine **pbu_$enable_device**. This routine clears the device's interrupt mask bit, permitting the processor to receive interrupts from the device. Calling the routine **pbu_$disable_device** sets the interrupt mask bit, which prevents receipt of device interrupts.

Any of the routines that make up the call side of the driver can call **pbu_$enable_device** and **pbu_$disable_device** to prevent the interrupt routine from running during the execution of critical sections of code. The interrupt routine can optionally enable interrupts by setting the appropriate return value, but it cannot call **pbu_$enable_device** or **pbu_$disable_device**. In **bm_example_c**, **bm_command** calls **pbu_$enable_device** just after it calls **bm_$sio** to start the I/O operation, and **bm_$cleanup** calls **pbu_$disable_device** as part of the release routine.

Of course, the controller itself may provide its own means of enabling and disabling interrupts that the driver can directly access. Refer to the controller documentation.

> **NOTE:** Interrupt lines typically float on the PC AT compatible bus (see Chapter 3, Section 3.4 for important information on enabling and disabling interrupts).

8.2.3 Processing Device Interrupts

Processing a device interrupt proceeds through three stages:

1. When an interrupt occurs, control is transferred to the System Interrupt Handler.
2. If a user-written interrupt routine exists, the System Interrupt Handler transfers control to this routine for further interrupt processing.
3. The user-written interrupt routine returns control to the System Interrupt Handler, which returns from the interrupt.

The System Interrupt Handler synchronizes operations with driver routines using eventcounts. An eventcount is an ec2_$eventcount type that programs can define to count the occurrence of a specific event. The eventcount may be shared among two or more processes, any of which can increment the eventcount to mark the passing of an event.

Each device has an associated eventcount. The System Interrupt Handler can advance this eventcount to indicate that an interrupt occurred. The driver's call side waits for an interrupt to occur by waiting for this eventcount to advance, as does the **bm_$wait** routine in **bm_example_c**. Thus, the device's eventcount provides the method by which the interrupt handler can signal to the driver's call side that an interrupt is completed. The *Programming with Domain/OS Calls* manual describes eventcounts in detail.

Depending on the requirements of the device and your driver, you may decide to let the System Interrupt Handler do all of the interrupt processing and not include an interrupt side in your driver. The advantage of not including an interrupt side is that you decrease the time it takes for program control to return from the System Interrupt Handler to the call side. (For information about interrupt processing overhead, see Appendix D, Section D.2.)

8.2.3.1 Processing by the System Interrupt Handler

When the System Interrupt Handler gains control, it performs the following functions:

- After determining which device has requested the interrupt, it disables further interrupts from the device by resetting the appropriate bit in the interrupt mask register.

- If a user–written interrupt routine exists, the System Interrupt Handler transfers control to it. Otherwise, the handler advances the eventcount associated with the device and exits. Note that in the latter case the handler does not enable interrupts from the device when it exits, and the driver must make another call to **pbu_$enable_device** if it wants to re–enable interrupts.

8.2.3.2 Processing by the User–Written Interrupt Routine

The user–written interrupt routine performs device–specific interrupt processing. Typically, these functions include

- Reading the device's status register(s) by referencing offsets into the CSR page
- Writing to the device's CSRs to acknowledge the interrupt
- Saving information about the interrupt for use by other driver functions
- Determining whether or not the device must perform more I/O, and restarting the device or calling an SIO routine
- Calling **pbu[2]_$map** to map a new I/O buffer
- Determining whether any other driver functions should be notified of the interrupt
- Determining whether or not to re–enable interrupts from the device
- Determining whether or not to advance the eventcount associated with the device

For an example of a user–written interrupt, refer to Appendix E, Section E.3 (C) and Appendix F, Section F.4 (Pascal).

8.2.4 Faults in User-Written Interrupt Routines

As noted in Section 8.1, a user-written interrupt routine is not allowed to generate any faults. If a fault does occur during interrupt processing, the operating system takes the following actions:

1. It locates the process owning the device, and saves fault diagnostic information at the low end of the interrupt routine's stack.

2. It generates an asynchronous fault for the owner process. The fault status is **fault_$pbu_user_int_fault** (in **/sys/ins/fault.ins.lan**).

3. It discontinues processing of the interrupt, advances the eventcount for the device, and resumes the interrupted process.

4. When the owning process next gains control, it receives the fault status that the system generated in Step 2.

Information about the fault can be obtained by using the **tb** (traceback) command with the **-u** option. The **-u** option dumps the pbu unit fault information, as shown in the following example. (Or you can supply a specific unit number for **tb** to dump, by using the **-u** *<unit number>* option to the **tb** command.)

```
$ tb -u

Process         86 (parent 85, group 0)
Time            88/03/15.10:08(EST)
Program         //cray/dmb/pbu_test.new/pbutest
Status          00120017: fault in user-space interrupt handler for
                pbu device (OS/fault handler)
In routine      "pbutest" line 872
Called from     "PM_$CALL" line 151
Called from     "pgm_$load_run" line 605

Fault frame for pbu unit 4
Fault Status     00120003: integer divide by zero (OS/fault handler)
User Fault PC    031F0D30
D0-D3:           00000001 00000000 00000000 0000FFFF
D4-D7:           0000FFFF 000002DC 00000000 0002331C
A0-A3:           03150B8C 03C89E18 03C89E18 02D48400
A4-A7:           031CFDF8 031F15CC 03150B7C 03150B64
Supervisor ECB   00000000
Supervisor SR    0000
Supervisor PC    00000000
```

The "User Fault PC", along with a map of the interrupt library and the information printed by the **aqdev** command (available only in the Aegis environment) with the **-d[b]** option, or by the **pbu_$acquire** routine with debug set to "true", can be used to isolate the logic that caused the fault.

8.2.5 Mapping Buffers from the Interrupt Routine

Drivers for devices that need to queue more data buffers than they can transfer at one time can facilitate transfers by calling **pbu[2]_$map** (and **pbu[2]_$unmap**) from their interrupt routines. An outline of this sequence of events follows:

1. The driver's resource allocation routines obtain the data to be transferred and wire down the needed buffers until they reach the limit set by **pbu[2]_$wire** (see Chapter 7, Subsection 7.1.1).
2. The driver calls **pbu[2]_$map** to map the first buffer and starts the I/O transfer.
3. When the interrupt routine gains control at the end of the first transfer, it saves the ending status. If there is another buffer waiting to be transferred, the interrupt routine calls **pbu[2]_$map** and starts another I/O transfer.

Mapping buffers from the interrupt routine ensures a minimal delay between data transfer startups because the interrupt routine need not reactivate the call side of the driver until an entire sequence of I/O has finished.

To use this same technique in a driver for an PC AT compatible device, you would make the following changes, depending on the machine type:

- Drivers running on the DN4000 would call **pbu2_$dma_start** and **pbu2_$dma_stop** *in addition to* **pbu2_$map** and **pbu2_$unmap**.
- Drivers running on the DN3000 would call **pbu_$dma_start** and **pbu_$dma_stop** *instead of* **pbu2_$map** and **pbu2_$unmap**.

8.3 Starting an I/O Operation

The Start I/O (SIO) routine is that part of the driver which actually performs the data transfer. The mechanics of the data transfer have already been described in Chapter 7. You might want to include an SIO routine in the interrupt side because the driver may have more data to transfer than can be handled in one I/O operation, and the interval between I/O operations is shorter when the interrupt side interacts directly with the SIO routine rather than going through the call side. In any case, if the interrupt routine (or any routine installed in the interrupt-side library) calls the SIO routine, it must be installed in the interrupt-side library.

In the sample driver in **bm_example**, the SIO routine (**bm_$sio**) is called by both call and interrupt sides and is, therefore, included in the interrupt side; refer to Appendix E, Section E.3 (C) and Appendix F, Section F.4 (Pascal).

Chapter 9

Global Drivers

This chapter describes how to design and write global device drivers. A global driver allows different processes to multiplex different operations on various devices such as the ETHERNET controller.

The general organization of a global driver is the same as for a private driver, consisting of a call side, interrupt side, and insert files. Likewise, the program **crddf** creates a DDF for a global driver in the same way as it does for a private driver: arguments to the program specify the unit number, call and interrupt libraries, initialization and cleanup entry points, interrupt entry points, and other useful information.

Whereas the private driver resides in user private address space where it is accessible only to the process assigned to that address space, the global driver resides in global address space where it is accessible to any process that wants it. This difference impacts the design of the global driver, which must be capable of handling calls from multiple processes and keeping them separate from each other.

> **NOTE:** Writers of global device drivers must not use variable names that conflict with names of system–defined symbols. Use the **esa** (**external_symbol_address**) command to determine if a name belongs to a system–defined symbol.

See **/domain_examples/gpio_examples/global_example** for an example of a global driver.

9.1 Controlling Multiple Processes

The major design consideration of a global driver is how to control multiple processes that are attempting to access the same procedure or data structure. Specifically, a global driver must be designed to perform these functions:

- Mutual exclusion; that is, preventing two or more processes from getting into the call library at the same time and tripping over each other
- Synchronization among client processes where one may be controlling resources on which others need to wait

9.1.1 Mutual Exclusion

Any routines in the call–side library that update shared data structures, including those that actually control the device, must be protected with mutual exclusion (MUTEX) locks; that is, surrounded by calls to mutex_$lock and mutex_$unlock. This precaution ensures that only one process can be executing in the body of a procedure at a time. A procedure designed for mutual exclusion would typically look like the following:

```
mutex_lock_rec_t lock;
void p (parameters)
 .
 .
 .
{
    if (mutex_$lock(lock, wait_time)){
       /* body of procedure */
    .
    .
    .
    mutex_$unlock (lock);
    }
 }
```

It should be noted that prior to releasing the lock (either for the purpose of waiting or upon exiting) the procedure must restore the state of all shared data structures to something that is “safe” for any other process.

If in the body of the procedure a process needs to wait on an event, the procedure must provide a means of releasing the lock so that another process can begin execution and satisfy the wait condition, as in the following:

```
mutex_$unlock (lock);
ec2_$wait ( ... );
(void) mutex_$lock (lock, wait_time);
```

9.1.2 Synchronization

As described in Chapter 8, GPIO software provides one built–in eventcount per device as a means of synchronizing device operations with driver routines. However, a global driver typically needs multiple eventcounts (for example, per client process, per socket, or per queue). The driver's interrupt handler must also be able to advance one or more of these eventcounts selectively. The following GPIO calls provide this functionality:

- **pbu_$allocate_ec**
- **pbu_$release_ec**
- **pbu_$advance_ec**

The first two are paired calls that manage the allocation from a special pool of eventcounts in wired space in the nucleus. The third enables an interrupt handler to selectively advance a particular eventcount based on the type of interrupt, data received, etc. All three routines use ordinary ec2_$ptr_t eventcount pointers; thus, the ordinary **ec2_$...** routines can be used. (Note, however, that only eventcounts from the special pool can be advanced by an interrupt handler.) For a description of these calls, refer to Appendix B.

The interrupt handler decides which eventcount to advance based on status or the results of the device, then advances that particular eventcount, awakening whatever process is waiting for that particular event. For example, a network device supports multiple devices, each waiting on an eventcount for a particular packet. When a packet comes in, the interrupt handler decides which process it is destined for by checking the packet type or other information in the packet. It then advances the appropriate eventcount, which notifies the process that its packet has arrived.

The procedures **pbu_$wait** and **pbu_$get_ec** work as they do for private drivers. The **pbu_$get_ec** procedure returns the pointer to the built–in eventcount in the device control table entry. This is advanced under control of the return value from the interrupt handler. The procedure **pbu_$wait** can be used to wait on this eventcount and a time–out. However, it should only be used in a global driver under the protection of a MUTEX lock. It is subject to a race condition so that, if two processes try to call it at approximately the same time, one waits while the other does not. The behavior is likely to appear unpredictable to the developer of a device driver.

9.2 Global Memory

Because global drivers reside in global memory, they are like global libraries in that they must be loaded at system initialization and unloaded at system shutdown. However, a global driver differs from a global library in that a global driver has read–write "state" and its data sections are loaded into writeable global virtual memory, making it accessible to all processes. Read–write data structures for global drivers can be declared in a data section of the call or interrupt library, or allocated dynamically by calling the routines rws_$alloc_rw_pool and rws_$alloc_heap_pool. If you call either procedure in a global driver, you must specify rws_$global_pool as an input parameter (for private drivers, you must specify rws_$std_pool).

There is only one copy of the data for the entire system, not one per process (as with the ..._impure_data$ sections for ordinary global libraries) or one read–only section per system (as with data$ and ..._pure_data$ sections). Any routines and variables that are exported by both the call–side and interrupt–side libraries are entered in the system–wide Known Global Table (KGT) so that they are visible and accessible to all processes and, therefore, corruptible by all processes.

If you wish to avoid filling up the KGT and generating long, unique variable names, you should put all variables in a named common section (overlay section) in the insert file; only one entry will be stored in the KGT rather than one for each variable. You should be forewarned, however, that if an overlay section contains initialization data, it is reinitialized each time a program containing that section is loaded.

9.3 Initialization and Cleanup

All driver initialization occurs when the driver is loaded (at system initialization), and all cleanup occurs when the driver is unloaded (at system shutdown). In other words, there is no per–process initialization or cleanup for global drivers. Each procedure in a global driver must be so designed that it restores the module invariant (doesn't leave the procedure in an inconsistent state) before releasing the lock and allowing another process to begin execution.

9.4 Fault Handling

If the interrupt handler in a private driver takes a fault, the fault is reflected back to the process that owns the driver. In a global driver, however, the fault is reflected back to the process that last touched the driver. The reason for this difference is that in a global driver you don't want the fault to reflect back to the owning process, which is the DM, the SPM, or the init process. As a result, if an interrupt handler generates a fault, the fault may not be sent back to the offending process.

9.5 Loading and Unloading

Unlike private drivers, which are dynamically loaded, global drivers must be loaded at system initialization. To load a global driver, you place the DDF for the global device in the directory **/dev/global_devices**. Immediately after loading the global libraries, the system searches the directory **/dev/global_devices** for global device drivers and then calls **pbu_$acquire** for each DDF it finds. If it finds non–DDF objects, it writes a message into the **/dev/sio** file for display on the screen or terminal, identifying them and the fact that they were not loaded. The list of global devices is recorded (by unit number) in **pbu_$global_units**. This read–only variable is initialized during system initialization and is readable by all processes. Thus, a driver can discover if it is loaded globally by testing whether its unit number is in that set. Devices are initialized in ascending order of unit number.

A status code is returned for any DDF that cannot be loaded, and the DDF is ignored. Files in the directory that are not DDFs are also ignored.

During system initialization for the DM, SPM, or init process and immediately after all libraries are initialized, the driver initialization routine is called for each global device. As mentioned, devices are initialized in ascending order of unit number. If a driver initialization routine returns bad status, system initialization is immediately suspended and an error message is displayed. The system cannot be restarted until either the problem is corrected or the device's DDF is removed from the directory **/dev/global_devices**. Note that DDFs can be removed with the delete_file (**dlf**) command to the phase II shell (the boot shell).

When the system exits, it calls the cleanup routine of each global driver to gracefully release each device. Devices are called in descending order of unit number so that they are released in Last In, First Out (LIFO) order.

9.6 Multiple–Device Drivers

The GPIO software package allows the same driver (either global or private) to support more than one device. A node configured with two ETHERNET controllers, for example, can be supported either by two independent drivers or by the same driver. In the latter case, the same call and interrupt libraries service both devices, using common data structures to control them. This holds true, whether or not the devices are shared.

Each device is specified by its own DDF. The DDF specifies the interrupt level, CSR page, entry points for the initialization and cleanup routines, and other vital information for the device. Different DDFs may point to the same call and interrupt modules. Specifying the multiple option with the **crddf** command ensures that **pbu_$acquire** doesn't load multiple copies of the same library. Note, however, that the initialization and cleanup entry points are called individually for each device.

The interrupt handler has an input parameter, in pbu_$unit_t format, that identifies the unit which this handler services so that it knows which registers to read, which data structures to work on, and so on. Thus, one interrupt routine can support multiple devices at different interrupt levels and decide dynamically which one has interrupted. This parameter is passed to the interrupt handler at interrupt time. The procedure signature of an interrupt handler is as follows:

For C:

```
pbu_$interrupt_return_t interrupt_handler (pbu_$unit_t &unit);
```

For Pascal:

```
function interrupt_handler(in unit:pbu_$unit_t):pbu_$interrupt_return_t;
```

Chapter 10

Building and Debugging

The final steps in creating your device driver are

- Building a single output file by compiling and binding the modules that make up your driver
- Debugging the driver

10.1 Building the Device Driver

The purpose of *building* is to create a single output object file by compiling and binding the several modules that make up your driver.

10.1.1 Compiling the Device Driver

A sample compile line from a build script from **/domain_examples/gpio_examples/bm_example/build_lib.sh** follows. Notice the **-pic** option to create a relocatable executable library.

```
.
.
.
pas bm_lib -pic -opt -b -l -map ^1 ^2 ^3 ^4
.
.
.
```

NOTE: You must use the **-pic** option to the compiler in order to create a relocatable executable library.

10.1.2 Binding the Device Driver

As input, the bind operations take the call–side and the interrupt–side (if one exists) routines. The output of the bind becomes the input for the DDF's **call_library** and **interrupt_library** parameters. Follow the instructions in this section to produce the proper input for the DDF. (Chapter 11 and Appendix A describe how to build the DDF and the DDF parameters.)

During device acquisition, **pbu_$acquire** reads the DDF to find the pathname in **call_library** and uses the pathname to install the device driver into user–process address space, making it accessible to user programs. Specification of **interrupt_library** is optional, depending on whether you have written interrupt routines for the driver.

If the driver does support one or more interrupt routines, use two bind operations to produce two separate executable modules:

- The call–side module (input for **call_library** in the DDF)
- The interrupt–side module (input for **interrupt_library** in the DDF)

For convenience, you can write a shell script to perform the two bind operations. This section provides a sample shell script.

The call–side module contains the call–side routines. For input to the bind, use the binary file produced in a successful compilation of the module(s) that contain the call–side routines, including

- The device initialization routine
- The driver routines
- An optional cleanup routine

The interrupt–side module contains the interrupt–side routine(s), bound with the GPIO source library **/lib/pbu_int_lib**. The interrupt–side module also contains any communications areas (a driver control block) to be shared between the interrupt routine(s) and the call–side routines. For input to the bind, use

- The system binary file **/lib/pbu_int_lib.**
- The binary file produced in a successful compilation of the interrupt–side module. In the sample shell script, this module is named **interrupt_side.bin**.
- Any other areas that the driver's interrupt routine references.

If you've written a device acquisition program (see Chapter 12, Subsection 12.1.2), you should not bind it with the driver.

When binding a driver that contains variables that are globally visible, we recommend using the **–mark** option to specifically mark each variable, rather than the **–allmark** option. Such variables include anything you want to share between the call side and the interrupt side as well as routines that are entry points for the application or GPIO software. If you are writing a shared driver you must not use the **–allmark** option. Refer to the *Domain Binder and Librarian Reference* manual for information on the **–mark** option.

Sample bind lines from a build script from **/domain_examples/gpio_examples/bm_example/build_lib.sh** follow. Notice the use of the **–mark** option, and that only symbols use the **–mark** option.

```
.
.
.
bind -b bm.lib -map >bm_lib.map - <<!
bm_lib.bin
-allunmark
-mark bm_$init
-mark bm_$cleanup
-mark bm_$read
-mark bm_$write
-mark bm_$wait
-end
!
cpf bm.lib /lib/bm.lib -chn
dlf bm_lib.bin
```

10.1.2.1 Using Bind to Page Align Buffers

If you have to page align a buffer, you may want to consider using the **–align** option. To use this option, you must declare the area of memory you want page aligned in a specially marked data section and then specify (in this order) **–align**, the name of that section, and the word **page** when entering the **bind** command line. For example, to page align a 1–KB area of memory called dma_buffer, first you would declare the following area of memory:

```
var (buffer_sec)
     dma_buffer : array[0..bytes_per_page-1]of char;
```

then you would enter the following command line:

```
$ bind -allmark my_call_side.bin -align buffer_sec page -b mycall_side.lib
  -m my_call_side.map
```

> **NOTE:** Arguments to the **–align** option must all appear on the same line with **–align**.

For additional information, refer to the *Domain Binder and Librarian Reference* manual. For information about placing variables in sections, refer to the *Domain Pascal Language Reference* manual and to the discussion of C's #section command in the *Domain C Language Reference* manual.

10.1.2.2 System Globals

Specifying **-sys** causes the binder to list all interrupt routine references to system globals. This list must be empty, as **pbu_$acquire** will not install an interrupt library with any unresolved globals (see Chapter 8, Section 8.1). The pathnames specified as the **-b** arguments are those you use for **call_library** and **interrupt_library** when you build the DDF (see Chapter 11). If you specify **-sys** when binding the call-side module, you'll probably notice that several unresolved globals are listed. These are external references to globals defined in the interrupt side and will be resolved at run time.

For information about the binder, refer to the *Domain Binder and Librarian Reference* manual. For information about shell scripts, refer to the *Aegis Command Reference* manual.

10.2 Debugging the Device Driver

You can use the high-level language debugging tool Domain Distributed Debugging Environment(DDE) on the call-side library by following the procedure outlined in this section, but you must not use it on the interrupt-side library. By its very nature, an interrupt routine cannot take faults, but must run to completion without interruption.

To make it possible to debug your interrupt routine, follow these guidelines:

- Debug the interrupt routines as call-side routines, *before* installing them in the interrupt side. That is, write your interrupt routines as you normally would, but for debugging purposes, install them in the call-side library, just after the call is made to the wait routine. Then, after you have debugged them with DDE, you can copy them into the interrupt-side library where they belong.
- There is no way to set break points in an interrupt-side routine. The best way to debug it is to make it leave a trail of data and flags about where it has been and then examine the data to see if it is what you would expect it to be.
- Store as many statistics as possible in a control block that is shared by the call and interrupt sides. In this way, you can read the control block to determine what the interrupt routine is doing.

Although you may use DDE on global drivers, there are special considerations when debugging in global space. These are discussed in Section 10.3.

Using DDE on call-side routines that are accessible from the application is simple and straightforward.

The following example of using DDE on an initialization routine is annotated, showing the descriptions of the debugging activity in *italic* print.

> NOTE: The line:
> dde> property lib <*call_library_executable_pathname*>
> tells the debugger to stop after loading the call library so that you can set break points before the pbu initialization routine gets called.

Debugging Example: *Description of Activity:*

```
$ dde fast                                  debug the application program

Initializing image"//acme/abe/backup/fast/fast"...
Initializing block"\\tape_mt_mgr".
Stopped at: \\tape_mt_mgr\main\71           debugger stops at program start
dde> prop lib mt_lib.lib                    stop when mt_lib.lib is loaded
dde> go                                     continue application program
Initializing image"//acme/abe/backup/driver/mt_lib.lib"...
The target program has loaded    //acme/abe/backup/driver/mt_lib.lib
Stopped at: \\tape_mt_mgr\main\71           debugger stops at program start
dde> prop lib mt_lib.lib                    stop when mt_lib.lib is loaded
dde> go                                     continue application program
Initializing image"//acme/abe/backup/driver/mt_lib.lib"...
The target program has loaded    //acme/abe/backup/driver/mt_lib.lib
Stopped at: unkown location (0E82117E)      debugger has loaded mt_lib.lib
dde> break \\mt_user\umt_$init              set a breakpoint in init routine
Initializing block"\\mt_user".
dde> break \\mt_user\umt_$release           and release routine
dde> go                                     continue
Break at: \\mt_user\umt_$init\395           here we are at the init routine
```

For additional information on using DDE, refer to the *Domain Distributed Debugging Environment (Domain/DDE) Reference* manual.

10.3 Debugging the Global Driver

A device driver that has been designed as globally shareable can always be loaded as a private nonshared driver, which means that it can be debugged in the privacy of its own address space, like an ordinary nonshared GPIO driver. Thus, you can use the debugging procedure outlined in Section 10.2.

To debug the driver in global space, work with just one process at a time. Getting the driver to work properly for a single process in global space should not be much more difficult to do than in private space. You need

- The **debug 4** switch to the phase II shell. This causes system initialization to tell where it has loaded the driver and interrupt libraries, how many pages are wired, where the entry points are, etc. (**debug 4** switch is the same as **aqdev's -db** switch and **crddf's -debug** switch; it generates the same information for a private driver as for a shared driver.)

- The **property system -on** command allows you to step right into the driver in global space.

To debug the driver as a shared driver, check that it has already been loaded from the directory **/dev/global_devices**. Start the debugger on any number of applications that use the device concurrently. Among other things, you will be able to track down deadlocks, observe synchronization problems, and notice shared access to data unprotected by locks.

The most reliable approach to debugging shared libraries with a state that is common to many processes is to test it with a random-number-driven diagnostic application. This application exercises the interface to the library, calling the different procedures at random with different values, then comparing the actual results with the expected ones. The random aspect is important because after enough time, you start to flush out synchronization problems. The first round of problems typically shows up within seconds, the second round within minutes, but the subtle bugs sometimes take hours or days before they happen. If your driver can stand up to a weekend of exercising by a dozen randomly driven processes without revealing any bugs, it has a good chance of surviving a number of real applications concurrently.

Chapter 11

Device Descriptor File

The Device Descriptor File (DDF) is a character special device file that stores static configuration information about a device, as well as information about the driver, that GPIO software needs to know. Each device connected to a node has one associated DDF. You create the DDF by invoking the **crddf** command (see Appendix A) and specifying a pathname for the DDF, normally in the **/dev** directory on the node to which the device is physically attached. The information stored in the DDF comes from the options you specify with the **crddf** command.

The DDF is mapped into user–process address space when the device is acquired. The DDF format is completely defined by the type pbu_$ddf_t (see Appendix B, Section B.1).

The DDF contains the following information:

- The device's unit number and the ID of the node to which the device is attached. The device's unit number is equal to its lowest assigned interrupt request line number.
- The pathname of the module that contains the user-written call-side routines. The **aqdev** command uses the pathname to install the device driver in the address space of the user process from which the call to **pbu_$acquire** was made.
- The entry point of the device initialization routine.
- The entry point of the cleanup routine, if one exists.
- The pathname of a library that contains one or more interrupt routines, if they exist.
- The stack size required by the interrupt routine(s).
- The address of the device's CSR page.
- The interrupt request line number for the device.

DDFs exist in three versions, which differ from each other according to the options you specify when invoking the **crddf** command. If you specify a Version 3 option (**-at**), then the system creates a Version 3 DDF. Table 11-1 lists the required options for each version. For a full description of all **crddf** options, refer to Appendix A.

Table 11-1. Required Options for Different DDF Versions

Version 1 Options*	**Version 2 Options**	**Version 3 Options**
-unit	Version 1 options plus any of the following:	Version 1 and 2 options plus any of the following:
-node		
-call_library	-csr_offset	-at
-initialization_routine	-memory_base (< 64 KB)	-vme
	-memory_size (< 64 KB)	-memory_base (> 64 KB)
		-memory_size (>64 KB)
*All Version 1 options are required for a Version 1 DDF (see Appendix A).		

11.1 Building a DDF in a Shell Script

One way to build the DDF is to create a shell script so that if you need to change the DDF, you can simply change the shell script and rebuild the DDF.

A shell script called **build_bm_ddf.sh** for the sample driver in **bm_example_c** follows; it also appears in the subdirectory **/domain_examples/gpio_examples/bm_example**. A brief explanation follows the example. As you read the script, note that it consists mainly of the **crddf** command and appropriate options read from standard input (this shell script builds a Version 1 DDF):

```
#!von
dlf /dev/bm > ? /dev/null
crddf /dev/bm - <<!
-unit 2
-node *
-csr_page 400
-call_library /lib/bm.lib
-interrupt_library /lib/bm_int.lib
-initialization_routine bm_$init
-cleanup_routine bm_$cleanup
-interrupt_routine 2 bm_$int
-serial_number 01234567
-user_info ddf_for_bulk_memory_device
-display
-end
!
```

The pathnames specified for call_library and interrupt_library are the call–side and interrupt–side modules generated by two of the **build_...** shell scripts in the **bm_example_c** subdirectory. Refer to **build_call_lib.sh** and **build_int_lib.sh** in these directories to see the origin of the pathnames **/lib/bm.lib** and **/lib/bm_int.lib**.

You could also use the bind shell script given in Chapter 10, Section 10.1.2. If you used this script, you would first have to compile the modules **bm_lib.pas** and **bm_int_lib.pas**. You would use the binary output from the compilations for **<call_side.bin>** and **<interrupt_side.bin>**; you would then specify the pathnames **/lib/bm.lib** and **/lib/bm_int.lib** as **<call_lib_pathname>** and **<interrupt_pathname>**. Note that the shell scripts in the online examples place the modules in the **/lib** directory. If the shell script you write to bind the device driver specifies pathnames in the **/lib** directory, ensure that the node's Access Control Lists (ACLs) provide you adequate rights to this directory. For information about shell scripts, refer to the *Aegis Command Reference* manual.

For the DDF's **initialization_routine, cleanup_routine,** and **interrupt_routine** parameters, the shell script provides the name of each routine. Note that these routines are part of the modules you specify for the **call_library** and **interrupt_library** parameters. You specify their names in the shell script to make their entry points available to the GPIO routines.

Certain **crddf** options (**revision, serial_number, user_info, debug,** and **memory_base**) are not used by any internal software and are intended only for the convenience of the user. You can use the **debug** option to turn on and off the driver's debugging logic, as in the following example:

```
if ddf_ptr^.debug then
begin
     flags := flags + [dbg];               { add debug flag }
     if dbg in flags then
           vfmt_$write2 ('ETHER:  Beginning initialization%.', 0, 0);
end;
```

11.2 Version 2 DDF

GPIO software creates a Version 2 DDF if you specify any or all of the following options: **memory_base** (less than 64 KB), **memory_size** (less than 64 KB), and **csr_offset.** The usefulness of Version 2 options is that you can store information that is subject to change in the DDF rather than in the driver, where it is more difficult to update. If, for example, your driver supports a memory-mapped controller, instead of coding the driver to include information about memory size and starting address (information that you might want to change), you can specify this information with the **memory_size** and **memory_base** options, as in the following **build_ddf** shell script (from the subdirectory **/domain_examples/gpio_examples/threecom_example**):

```
von
dlf /dev/ethernet
crddf /dev/ethernet - <<!
-unit 0
-node *
-memory_base 4000
-memory_size 2000
-call_library /lib/ether.lib
-interrupt_library /lib/ether_int.lib
-initialization_routine ether_$init
-cleanup_routine ether_$cleanup
-interrupt_routine 0 ether_$int0
-serial_number
-user_info
-display
-end
!
```

Then, your driver's initialization routine can fetch this information and store it in the control block. This is how the initialization routine in the **threecom_example** driver does it:

```
if ddf.version = pbu_$ddf_version_2 then begin
    mem_base := ddf.memory_iova;
    mem_len := ddf.memory_size;
    end
else begin
    mem_base := 16#6000;
    mem_len := 16#2000;
    end;
```

The **csr_offset** option allows you to supply information to the driver about the address of the controller's CSR page. In the following example, **csr_offset** is used to specify a CSR address that falls within the range 80–FF recommended for 8-bit MULTIBUS controllers (see Chapter 1, Subsection 1.3.1):

```
von
dlf /dev/comm
crddf /dev/comm - <<!
-unit 0
-node *
-csr_page 0
-csr_offset 80
-call_library /lib/comm.lib
-interrupt_library /lib/comm_int.lib
-initialization_routine comm_$init
-cleanup_routine comm_$cleanup
-interrupt_routine 0 comm_$int0
-serial_number
-user_info
-display
-end
!
```

You should note that the information you supply with any Version 2 option is not used by the operating system and can be in any form that is useful to the driver. In fact, you can use these options to store any kind of information you want.

11.3 Version 3 DDF

GPIO software creates a Version 3 DDF if you specify any or all of the following options: **at, vme, dma_channel, memory_base** (greater than 64 KB), or **memory_size** (greater than 64 KB). Subsections 11.3.1 through 11.3.3 present shell scripts for building DDFs for a PC AT compatible device and a VMEbus device. For a full description of all Version 3 options, refer to Appendix A.

11.3.1 DDF for a PC AT Compatible Device

A sample shell script that builds a DDF for a PC AT compatible device follows. Note that the –csr_page iovas are supplied by the **cvt_at** command (see Appendix A).

```
von
dlf /dev/at
crddf /dev/at - <<!
-at
-unit 4
-nodef *
-csr_page 208 21F
-dma_channel 7
-call_library bmlib
-interrupt_library bmintlib
-initialization_routine bm_$init
-cleanup_routine bm_$cleanup
-interrupt_routine 4 bm_$int
-serial_number 01234567
-user_info at_ddf
-display
-end
!
```

The DDF generated by the preceding shell script is as follows:

$ crddf /dev/at –display

```
ddf version:   3
device uid:    00030004.00002CBC  (unit 4, node 2CBC)
controller is an AT device.
dma channel: 7
csr page iova:      200-21F
call library:               bmlib
interrupt library:          bmintlib
initialization entry point: bm_$init
cleanup entry point:        bm_$cleanup
interrupt stack size: 1024
interrupt routines:
   level 0: [unused]
   level 1: [unused]
   level 2: [unused]
   level 3: [unused]
   level 4: bm_$int
   level 5: [unused]
   level 6: [unused]
   level 7: [unused]
   level 8: [unused]
   level 9: [unused]
   level 10: [unused]
   level 11: [unused]
   level 12: [unused]
   level 13: [unused]
   level 14: [unused]
   level 15: [unused]
serial number: "01234567        "
revision:      "        "
user info:     "at_ddf
      "
```

11.3.2 DDF for a VMEbus Device

A sample shell file that builds a DDF for a VMEbus device follows:

```
von
dlf /dev/vme
crddf /dev/vme - <<!
-vme
-unit 14
-nodef *
-csr_page C000
-call_library bmlib
-interrupt_library bmintlib
-initialization_routine bm_$init
-cleanup_routine bm_$cleanup
-interrupt_routine 14 bm_$int
-serial_number 01234567
-user_info vme_ddf
-display
-end
!
```

The DDF generated by the preceding shell script is as follows:

$ build_vme.sh

```
dlf /dev/vme
crddf /dev/vme - <<!
New DDF.
ddf version:   3
device uid:    0003000E.00002CBC  (unit 14, node 2CBC)
controller is a VME device.
csr page iova:      C000
call library:                bmlib
interrupt library:           bmintlib
initialization entry point: bm_$init
cleanup entry point:         bm_$cleanup
interrupt stack size: 1024
interrupt routines:
   ID F8: [unused]
   ID F9: [unused]
   ID FA: [unused]
   ID FB: [unused]
   ID FC: [unused]
   ID FD: [unused]
   ID FE: bm_$int
   ID FF: [unused]
serial number: "01234567        "
revision:      "        "
user info:     "vme_ddf             "
```

11.3.3 DDF for a Device Accessed Through a Streams Manager

The following build script uses the **-major** option to set the major device number of the DDF. The type uid will be set to the type associated with the major device number given. For information about how a major device number is mapped to a trait manager refer to the description of the **mkdevno** command in the *Managing SysV System Software* manual.

```
#!/com/sh
dlf /dev/mm >? /dev/null
#
crddf /dev/mm - <<!
-unit 0
-node *
-csr_page 0
-call_library /lib/mm.lib
-interrupt_library /lib/mm_int.lib
-initialization_routine mm_$init
-cleanup_routine mm_$cleanup
-interrupt_routine 0 mm_$int
-major 4
-minor 0
-check
-display
!
```

The DDF generated by the preceding shell script is as follows:

```
$ build_mm_ddf.sh


New DDF.
No missing fields.

ddf version:   1
device uid:    00030000.0000712E  (unit 0, node 712E)
controller supports 16-bit addresses only (M16).
csr page iova:      0
call library:                /lib/mm.lib
interrupt library:           /lib/mm_int.lib
initialization entry point: mm_$init
cleanup entry point:         mm_$cleanup
interrupt stack size: 1024
interrupt routines:
   level 0: mm_$int
   level 1: [unused]
   level 2: [unused]
   level 3: [unused]
   level 4: [unused]
   level 5: [unused]
   level 6: [unused]
   level 7: [unused]
serial number: "                 "
revision:      "        "
user info:     "
       "
```

The following build script uses the **-type** option to set the type uid of the DDF to the type associated with the installed type "stype". (Refer to the *Using the OPEN Systems Toolkit to Extend Your Domain Streams* manual for more informaion.) The major device number of the DDF will also be set.

```
#!/com/sh
dlf /dev/stype >? /dev/null
#
crddf /dev/stype - <<!
-unit 0
-node *
-csr_page 0
-call_library /lib/stype.lib
-interrupt_library /lib/stype_int.lib
-initialization_routine stype_$init
-cleanup_routine stype_$cleanup
-interrupt_routine 0 stype_$int
-type stype
-check
-display
!
```

The DDF generated by the preceding shell script is as follows:

$ build_stype_ddf.sh

```
New DDF.
No missing fields.

ddf version:   1
device uid:    00030000.0000712E  (unit 0, node 712E)
controller supports 16-bit addresses only (M16).
csr page iova:      0
call library:                /lib/stype.lib
interrupt library:           /lib/stype_int.lib
initialization entry point: stype_$init
cleanup entry point:         stype_$cleanup
interrupt stack size: 1024
interrupt routines:
   level 0: stype_$int
   level 1: [unused]
   level 2: [unused]
   level 3: [unused]
   level 4: [unused]
   level 5: [unused]
   level 6: [unused]
   level 7: [unused]
serial number: "                "
revision:      "        "
user info:     "
```

Chapter 12

Acquiring and Releasing the Device

This chapter describes the routines used for acquiring and releasing the device. It also describes the advantages of using the **pbu_$acquire** routine rather than the **aqdev** command to acquire the device.

12.1 Acquiring the Device

The **pbu_$acquire** routine acquires control of the device by performing the following:

- Mapping the DDF to the address space of the user process from which the call to **pbu_$acquire** was made
- Locking the DDF for the device
- Loading the device driver into the user–process address space
- Wiring the interrupt routine, interrupt data, and interrupt stack
- Mapping the device's CSR page to the user–process address space

In addition, **pbu_$acquire** calls the device initialization routine specified in the DDF. For a description of **aqdev** and **pbu_$acquire**, refer to Appendixes A and B.

There are two ways to make the call:

- Invoking the **pbu_$acquire** routine
- Invoking the **aqdev** command

The end result of either is the same. Note, however, that the **aqdev** command is only available in the Aegis environment.

12.1.1 Using aqdev

If you are working in the Aegis environment and if you plan to execute several application programs that use the device, you can acquire the device with the **aqdev** command, as in the following:

```
 $ aqdev /dev/my_dev

Device 0 acquired.

 $ application_1
 $ application_2
 $ application_3
 $ <CTRL/Z>

*** EOF ***
 Device 0 released.
 $
```

The **aqdev** command invokes **pbu_$acquire**, which loads the driver into the address space of the user process from which the **aqdev** command was issued. The application programs are then invoked. Because the driver routines have been installed in user–process address space, each application program can call the driver routines.

After installing the device driver, **aqdev** creates a new copy of the shell command interpreter. Typing CTRL/Z (inserting the EOF mark) causes the new shell to return control to **aqdev**, which unloads the driver routines from the user process and releases the device so that the application programs may no longer call driver routines.

In previous software releases, some programmers preferred to use **aqdev** to acquire the device because **aqdev** simplified the task of debugging. With Software Release 10 this advantage no longer applies and we recommend that you use **pbu_$acquire** instead, for the following reasons:

- The increased power of the Domain Debugging Environment (DDE) eliminates previous problems in debugging
- The **aqdev** command is available only in the Aegis environment

12.1.2 Acquiring a Device in Your Application

If you need to run only one application, such as a server, then you can call **pbu_$acquire** to load the driver. It is preferable to call **pbu_$acquire** rather than to use **aqdev** to acquire the device. Calling **pbu_$acquire** from your application minimizes your dependence on in–process program execution for future software releases.

The following program uses **pbu_$acquire** to acquire a device, invoke the application, and release the device:

```
#include <apollo/base.h>
#include <apollo/error.h>
#include <apollo/pbu.h>

char *dev = "/dev/my_dev";

main(argc, argv)
int argc;
char *argv[];
{
    pbu_$unit_t unit;
    status_$t   st;

    pbu_$acquire(dev, strlen(dev), unit, &st);
    if (st.all != status_$ok) {
        error_$print(st);
        exit(1);
    }

    /* application code ..... */

    pbu_$release(unit, true, &st);
    if (st.all != status_$ok) {
        error_$print(st);
        exit(1);
    }

    exit(0);
}
```

12.1.3 Acquiring a Device with pbu_$acquire_stream

For information on how to write and use streams, refer to the *Using the OPEN System Toolkit to Extend the Streams Facility* manual.

The following examples show how to acquire a device with a streams manager using the **pbu_$acquire_stream** routine. The first example is written in C, the second example is written in Pascal.

C version:

```
void mgr_open (
            xoid_$t              &xoid,
            ios_$open_options_t &opts,
            spe_handle_ptr_t     *h,
            status_$t            *st);
```

```
{
    pbu_$unit_t unit;

    /* attempt to acquire device */

   pbu_$acquire_stream(xoid, unit, st);
   if (st->all != status_$ok) {

        /* see if device is acquired globally. the status
           pbu_$unit_is_global indicates that not only is the requested
           pbu unit already acquired globally, but it is also the device
           identified by 'xoid'.  'unit' is also returned. */

        if (st->all == pbu_$unit_is_global)
            st.all = status_$ok;            /* it is, report OK */
        else {                              /* problems */
            st.fail = true;
            return;                         /* return with bad status */
        }
    }

    /* device is acquired (either globally or in
       private address space) */

    ....

}   /* mgr_open */
```

Pascal version:

```
module mgr;

define mgr_open;

procedure mgr_open (*
            IN  xoid: xoid_$t;
            IN  opts: ios_$open_options_t;
            OUT h: spe_handle_ptr_t;
            OUT st: status_$t
            *);

var unit:       pbu_$unit_t;

begin

    { attempt to acquire device }

    pbu_$acquire_stream(xoid, unit, st);
    if st.all <> status_$ok then begin
```

```
        { see if device is acquired globally. the status
          pbu_$unit_is_global indicates that not only is the requested
          pbu unit already acquired globally, but it is also the device
          identified by 'xoid'.  'unit' is also returned. }

        if st.all = pbu_$unit_is_global then
            st.all := status_$ok                { it is, report OK }
        else begin                              { problems }
            st.fail := true;
            return;                             { return with bad status }
            end;
        end;

    { device is acquired (either globally or in private address space) }

    ....

end;   { mgr_open }
```

12.2 Releasing the Device

You can release a device by inserting the EOF mark or by calling **pbu_$release**. For the case where **aqdev** was used to acquire the device, the device may be released by issuing an EOF mark to the DM. For the case where **pbu_$acquire** or **pbu_$acquire_stream** was used to acquire the device, the application or streams manager should call **pbu_$release**.

The **pbu_$release** routine unwires all wired procedures and data pages, deallocates any I/O map space, unmaps any mapped controller memory, and releases control of the device. If the DDF contains the entry point of a cleanup routine, **pbu_$release** will call it during device release. The device acquisition program can call **pbu_$release**. However, since **pbu_$release** unloads the driver library, device drivers should not call it. The **pbu_$release** routine is described in Appendix B.

Appendix A

GPIO Commands

This appendix describes the use, format, parameters, and options for the four GPIO commands that the user can invoke: **aqdev**, **crddf**, **cvt_at**, and **rldev**.

aqdev (acquire_device) Acquires control of a peripheral device.

FORMAT

aqdev *pathname* [*–d*[*b*]] [*–c progname arg1 arg2* ...]

ARGUMENT

pathname (required)	The pathname of the DDF associated with the device to be acquired. The pathname normally refers to the a DDF in **/dev** directory on the node to which the device is physically attached.

OPTION

–d[b]	Acquires the device in debug mode. Specifying this option causes **aqdev** to display the addresses of the DDF and the CSR page, and to display information about device driver routines as they are loaded into user-process address space.
–c	Allows **aqdev** to run a command instead of a shell. Specifying this option causes **aqdev** to acquire the device, run *progname* (passing *arg1, arg2* ... to the program), release the device, and finally, return to the shell. This option also allows the user to use **aqdev** in a shell script (see example 3).

DESCRIPTION

NOTE: This command is supported only for compatibility purposes. At SR10 the **aqdev** command is available only in the Aegis environment, and will not be supported in the next major Software Release.

The **aqdev** command is used to acquire a device at the shell command level. When invoked, **aqdev** calls the routine **pbu_$acquire,** which maps to user-process address space the DDF, the device's CSR page, and device driver routines and associated data structures.

Currently, the **aqdev** command (without the **–c** option) creates a new copy of the shell after it installs the device driver. To release the device, exit the shell, which causes the new shell to return to the **aqdev** command. The **aqdev** command then releases the device.

ERROR MESSAGES

`ddf has wrong file type`

The file pointed to by the specified pathname is not a DDF.

`name not found`

The file pointed to by the specified pathname does not exist.

`object is not local`

The DDF belongs to a device that is physically connected to another node.

`PBU not present`

No peripheral bus is present on the system.

`unit in use`

Another process is using the device.

`unit is global`

Device unit number is already acquired as a global device.

EXAMPLES

1.

```
$ aqdev /dev/mt0 -db
DDF mapped at 2D0000 for 1024 bytes.
Interrupt stack_size = 1024
CSR page at 2D8000
Interrupt library: Start address = 000000, n_sects = 3
Name = PROCEDURE$, loc = 2BABAE, len = 0002AC
Name = DATA$, loc = 2BAEFC, len = 000C6E
Name = DEBUG$, loc = 2BAE5A, len = 0000A2
Call library: Start address = 000000, n_sects = 3
Name = PROCEDURE$, loc = 2E0040, len = 00126C
Name = DATA$, loc = 2BBB6A, len = 000190
Name = DEBUG$, loc = 2E12AC, len = 00051C
Device 3 acquired.
$
```

2.

```
$ aqdev /dev/my_dev
Device 0 acquired.
$ (Run your program using the device.)
$ CTRL/Z
*** EOF ***
Device 0 released.
$
```

3.

```
$ aqdev /dev/my_dev -c driver_application
Device 0 acquired.
(driver_application runs using the device.)
Device 0 released.
$
```

crddf (create_ddf) Creates, displays, or modifies a Device Descriptor File (DDF).

FORMAT

crddf *pathname [-option] [-option] ... [-]*

ARGUMENT

pathname (required)	The pathname of the DDF to be created. The pathname normally refers to a DDF in the **/dev** directory on the node to which the device is physically attached.

OPTIONS

–	Specifies that **crddf** is to read further options from **stream_$stdin**.
–at	Specifies that the device resides on the IBM PC AT compatible bus. It is recommended that this option be the first specified when building a new DDF. Valid unit numbers when **–at** is specified must be in the range 0–15 and must not be used by Domain system-supplied devices. Specifying this option results in the generation of a Version 3 DDF.
–call_library *pathname*	Specifies the pathname of the call side of the driver. *This option is required when creating a DDF.*
–check	Checks the DDF to ensure that all required files have been specified. The options associated with these requirements are **call_library, initialization_routine, node,** and **unit**.
–cleanup_routine *entry-name*	Specifies the entry point name of a cleanup routine to be called when the device is released.
–csr_offset *port-number*	Specifies the offset into the CSR page, in hexadecimal format, at which the device's control and status registers are located. Device drivers may use this information during controller initialization. Specifying this option results in the generation of a Version 2 DDF.

–csr_page *iova* Specifies the hexadecimal address of the device's CSR page. If this option is omitted, no CSR page is mapped. The following information applies to the particular bus structure implemented on your node:

- MULTIBUS: Optional. If specified, must be page aligned.
- VMEbus: Optional. If specified, must be page aligned and in the range 0000–7FFF (16–bit addressing, 16–bit data path) and C000–DFFF (24–bit addressing, 16–bit data path).
- PC AT compatible bus: Optional. If specified, must be 8–byte aligned, may indicate a range (–csr_page 200 21F). If the second parameter is missing, a range of eight consecutive bytes is assumed ("–csr_page 200" assumes a range of 200–207). Use the **cvt_at** command (described in this appendix) to derive properly aligned iovas.

–debug Sets a flag (ddf.debug) that can be used to turn on debugging logic in a driver.

–display Displays the current contents of the DDF.

–dma_channel *channel–number*
Specifies to the driver the DMA channel number that a PC AT compatible controller will use. Specifying this option results in the generation of a Version 3 DDF.

–end Closes the updated DDF and exits.

–initialization_routine *entry–name*
Specifies the device driver's initialization routine entry point name. **pbu_$acquire** calls this routine during device acquisition. *This option is required when creating a DDF.*

–interrupt_library *pathname*
Specifies the pathname of the device driver's interrupt side. You only need to specify this parameter if you have user–written interrupt routines.

–interrupt_routine *level* [*entry–name*]
Assigns an interrupt request level to the device and optionally specifies the name of an interrupt routine to handle device interrupts at that level.

The level is required; the name of the interrupt routine is optional. If no *entry–name* is specified, the System Interrupt Handler processes the interrupt and advances the eventcount associated with the device. A single device may interrupt at several levels. If that is the case, this option can be specified more than once.

If the **–interrupt_routine** option is omitted, interrupts are processed at the level equal to the device's unit number.

–major *decimal–number*

Allows the user to set the major device number of the DDF file. With this option, **crddf** looks up the type UID corresponding to the major device number. The type UID of the DDF is set to the result of the lookup. If **crddf** doesn't find a type UID corresponding to the major device number, the user is notified that the major device number is not set. The **–major** and **–type** options are mutually exclusive.

–memory_base *iova*

Specifies the bus address that marks the base of a controller's local memory. Device drivers use this information in arguments to the GPIO routine **pbu[2]_$map_controller** to associate a virtual address with the memory on the controller. Specifying this option with an iova less than 64 KB results in the generation of a Version 2 DDF; if the iova is greater than 64 KB, a Version 3 DDF is generated.

–memory_size *length*

The size, in hexadecimal format, of controller memory. Device drivers use this information in arguments to **pbu[2]_$map_controller** to associate a virtual address with the memory on the controller. Specifying this option with an iova less than 64 KB results in the generation of a Version 2 DDF; if iova is greater than 64 KB, a Version 3 DDF will be generated.

–minor *decimal–number*

Allows the user to set the minor device number of the DDF file in the range 0 to 512.

–multiple

Specifies that the device driver supports more than one device and causes **pbu_$acquire** to use copies of previously loaded call–side and interrupt–side libraries, so as to avoid loading multiple copies of the same driver.

–node [*node–number* | *]

–nodef [*node–number* | *]

Specifies the number, in hexadecimal format, of the node to which the device is physically connected. *This option is required when creating a DDF.* The **–nodef** option suppresses the check that makes certain that the node exists. An asterisk (*) specifies the local node.

–quit

Causes **crddf** to exit without modifying the original DDF.

–revision [*string–8*]

Specifies an optional revision number as an 8–character string.

–serial_number [*string–16*]

Specifies an optional serial number as a 16–character string.

–share Specifies that the DDF describes a memory–mapped controller that can be shared by multiple applications. The **pbu[2]_$map_controller** routine maps the shared controller into global address space, and **pbu_$mem_ptr** returns its address. Unlike a nonshared controller, a shared controller is not automatically unmapped on abnormal termination of a device driver.

> **NOTE:** Apollo recommends that a fault handler be established to ensure that the controller is unmapped should the driver terminate without going through the normal device release mechanism.

–stack_size *decimal–number*
Specifies the number of bytes to be allocated to the interrupt stack (the default is 1024).

–type *type_name*
Allows the user to change the type UID of the DDF file to the type of *type_name*. The **crddf** command looks up the major device number that corresponds to the type of *type_name*. The **crddf** command then uses the result of the look–up as the DDF's major device number. If **crddf** does not find the major device number for *type_name*, a major device number is created for *type_name*, and the DDF's major device number is set to it. The **–type** and **–major** options are mutually exclusive.

–unit *unit_number*
Specifies the device unit number. The unit number must match the lowest interrupt level on which the device interrupts. *This option is required when creating a DDF.* The following information applies to the particular bus structure implemented on your node:

- MULTIBUS: Must lie in the range 0–5 for a 16–bit controller and 0–7 for a 20–bit controller.
- VMEbus: Must lie in the range 8–14.
- IBM PC AT compatible bus: Must lie in the range 0–15.

–update Allows modification of an existing DDF. *This option must be specified before any other option.*

–user_info *string–64*
Specifies up to 64 characters of optional information for use by the device driver. The *string–64* argument is initialized as a field of blanks that you overwrite.

–vme Specifies that the device resides on the VMEbus. It is recommended that this option be the first specified when building a new DDF. Valid unit numbers when **–vme** is specified must be in the range 8–14. Specifying this option results in the generation of a Version 3 DDF.

–20_bit_addressing

Specifies that the DDF describes a 20–bit controller. *You must use this option when creating a DDF for a 20–bit controller on a node that has a 20–bit MULTIBUS.*

DESCRIPTION

Invoke the **crddf** command at the shell prompt or from a shell command file to create, display, or modify a DDF. The DDF created by **crddf** is a character special device file.

You can create different versions of a DDF, depending upon which options you specify with the **crddf** command. Modifying an existing DDF by adding Version 2 or Version 3 options results in the generation of a Version 2 or Version 3 DDF. Refer to Chapter 11, Table 11–1 for a list of the relevant options and to Chapter 11, Section 11.1 through Section 11.3 for a discussion of the different DDFs and examples that show how to create them. *Note that all three versions must include the following options:* **unit, node, call_library**, and **initialization_routine**.

The following options are not used by the operating system and are only for the optional use of the driver: **csr_offset, debug, dma_channel, memory_base, memory_size, revision, serial_number,** and **user_info.**

The entire contents of the DDF are available to the driver's initialization routine by reference through the **ddf_ptr** argument.

EXAMPLES

1. Create a new DDF specifying only the required information.

```
$ crddf /dev/mt0 –
new ddf.
> –unit 3
> –node 2f
> –csr_page 1400
> –call_library /lib/mt.lib
> –initialization_routine mt_$init
> –interrupt_library /lib/mt.int.lib
> –interrupt_routine 3 mt_$int
> –check
no missing fields.
> –end
$
```

2. Display a DDF.

```
$ crddf /dev/mt0 -display
ddf version:    1
device uid:     00030003 0000002f  (unit 3, node 2f)
csr page iova: 1400
call library:                  /lib/mt.lib
interrupt library:             /lib/mt.int.lib
initialization entry point: mt_$init
cleanup entry point:           mt_$cleanup
interrupt stack size: 1024
interrupt routines:
   level 0: [unused]
   level 1: [unused]
   level 2: [unused]
   level 3: mt_$int
   level 4: [unused]
   level 5: [unused]
   level 6: [unused]
   level 7: [unused]
serial number:
revision:
user info:
$
```

3. Change the name of the interrupt routine in an existing DDF.

```
$ crddf /dev/mt0 -update -interrupt_routine 3 mt_$sio
$
```

4. Create a new DDF for a device that will be accessed through streams for the installed type "foodev" (for information on writing and using streams managers refer to the *Using the OPEN System Toolkit to Extend the Streams Facility* manual):

```
$ crddf /dev/foodev -
New DDF.
> -unit 3
> -node *
> -csr_page 1400
> -call_library /lib/foodev.lib
> -initialization_routine foodev_$init
> -interrupt_library /lib/foodev.int.lib
> -interrupt_routine 3 mt_$int
> -type foodev
> -check
No missing fields.
> -end
$
```

cvt_at (convert_at_addresses)	Converts a PC AT compatible I/O address to a processor physical address.

FORMAT

cvt_at *[at_addr [at_addr] ...]* *[at_addr1—at_addr2]*

ARGUMENTS

at_addr

A PC AT compatible I/O address in hexadecimal.

at_addr1—at_addr2

A range of PC AT I/O addresses in hexadecimal.

DESCRIPTION

The **cvt_at** command converts PC AT compatible bus addresses to physical addresses in processor address space. The command reports any conflict between the PC AT address you specify and the address of any system-supplied device. It also supplies the iova for so-called PC AT (16-bit) addresses (see Example 2 that follows).

If one or more addresses are specified, each is translated, and its physical address, page number, offset within a page, and the CSR page iova are displayed. If addresses are specified in pairs and both fall on the same page, a warning is given. Also, if one of the addresses in the pair is in the 0–3FF range for 10-bit controllers and the other address is in the 3FF–FFFF range for 16-bit controllers, a warning is given if they would conflict with each other on the bus.

If a dashed parameter is specified, all addresses between those values are generated and converted. Both addresses must be either 10-bit or 16-bit.

A warning is given if an address conflicts with a known system device control page within processor memory.

EXAMPLES

1. Translate I/O address 5100:

```
$ cvt_at 5100

AT Addr  DOMAIN Phys Addr   DOMAIN PPN   Page offset   CSR Iova
 5100         48140             120          140           100
```

The **cvt_at** command converts the I/O address 5100 to the Domain physical address 48140 and displays a CSR iova of 100. When you create the DDF for the PC AT compatible device, use this iova (not the I/O address) as input to the **crddf** command option **–csr_page** (for example, **crddf /dev/at1 –csr_page 100**).

2. Translate I/O address 1A4:

```
$ cvt_at 1A4

AT Addr  DOMAIN Phys Addr   DOMAIN PPN   Page offset   CSR Iova
  1A4         4D004             134          4             1A4
Warning:  Above address (1A4) may occupy same physical page as
DOMAIN device, if present: winchester (4D000).
```

The **cvt_at** command converts the I/O address 1A4 to the Domain address 4D004 and issues a warning that a conflict between device control pages exists if a Winchester disk is present in the configuration.

3. Translate I/O address 41A4:

```
$ cvt_at 41A4

AT Addr  DOMAIN Phys Addr   DOMAIN PPN   Page offset   Csr Iova
 41A4         4D104             134          104           1A4
Warning:  Above address (41A4) may occupy same physical page as
DOMAIN device, if present: winchester (4D000).
```

rldev (release_device) Releases a peripheral device.

FORMAT

rldev *{unit-number | all} [-force]*

ARGUMENTS

unit-number
The unit number of the device to be released.

all
Causes **rldev** to release all devices acquired by the current process.

OPTION

-force Causes **rldev** to release the device unconditionally, waiting 1 second (at most) for any I/O operations to complete.

DESCRIPTION

NOTE: This command is currently supported only for compatibility purposes. At SR10 it is available only in the Aegis environment, and will not be supported at the next major Software Release.

The **rldev** command releases one or more devices previously acquired by the **aqdev** command (or **pbu_$acquire**). Currently, when you invoke **rldev**, a message appears advising you to insert the EOF mark to release the device.

ERROR MESSAGES

`device not acquired`

The current process has not acquired any device associated with the specified unit number.

`object is not local`

The DDF belongs to a device that is physically connected to another node.

Appendix B

GPIO Routines

This appendix describes the calling format, input and output parameters, and usage of the GPIO routines that application programs and user–written device drivers can call. Also described are the data types that are used by the routines and error messages.

B.1 Data Types

The following are the constants and data types used by GPIO routines. Records are illustrated to show their composition and byte displacements.

CONSTANTS

Name	Value	Description
pbu_$ddf_current_version	1	Current version of DDF.
pbu_$ddf_version_2	2	Version 2 of DDF.
pbu_$ddf_version_3	3	Version 3 of DDF.
pbu_$ddf_lowest_version	1	Lowest supported version of DDF.
pbu_$ddf_highest_version	3	Highest supported version of DDF.
pbu_$ddf_pathname_len	64	Maximum length of pathnames in DDF.
pbu_$ddf_ep_name_len	32	Maximum length of entry point names in DDF.
pbu_$info_version	1	Current version of pbu_$info_t.

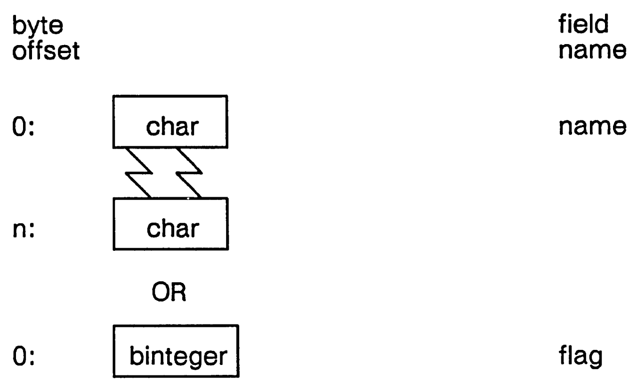
Name	Value	Description
pbu_$no_csr_iova	−1	Indicates no CSR page (Version 2).
pbu_$max_unit	7	Maximum allowable unit number.
pbu_$min_vme_unit	8	Minimum VMEbus unit number.
pbu_$max_vme_unit	15	Maximum VMEbus unit number.
pbu_$max_at_unit	1	Maximum PC AT unit number.
bytes_per_page	1024	Bytes per page.
pbu_$max_virtual_address	16#7FFFFFFF	Maximum user-space virtual address.

DATA TYPES

ec2_$ptr_t	A 4-byte integer. A pointer to an eventcount.
pbu_$buffer_t	An array of up to 1024 characters. A buffer to be mapped.
pbu_$bus_t	A 2-byte integer. Indicates the presence of the specified bus. Returns any combination of the following values: **pbu_multibus_m16** The node supports the 16-bit MULTIBUS. **pbu_multibus_m20** The node supports the 20-bit MULTIBUS. **pbu_atbus** The node supports the PC AT compatible bus. **pbu_vmebus** The node supports the VMEbus.
pbu_$csr_page_ptr_t	A 4-byte integer. A pointer to the CSR page.
pbu_$csr_page_t	An array of up to 1024 characters. A Control and Status Register (CSR) page.

pbu_$ddf_int_list_entry_t

The name of the driver's interrupt routine entry point. The following diagram illustrates this data type.



Field Description

name
Interrupt routine entry point.

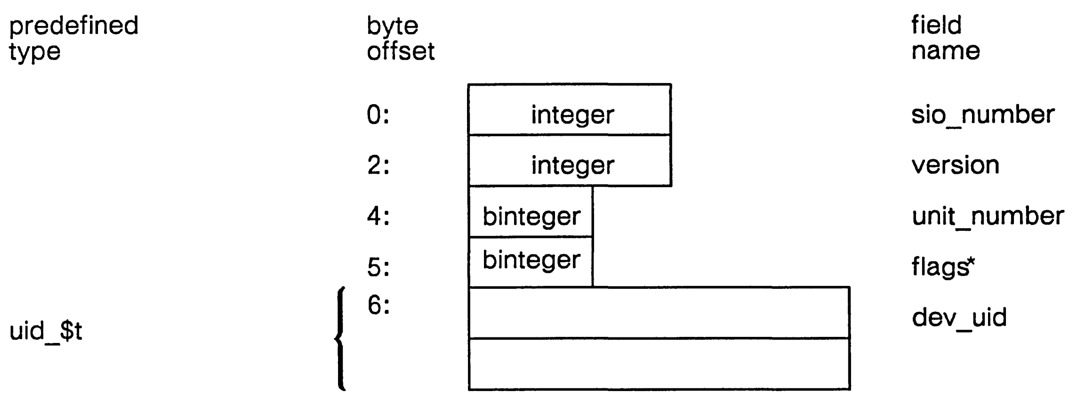
flag
For internal use only.

pbu_$ddf_ptr_t

A 4-byte integer. A pointer to a DDF.

pbu_$ddf_t

A Device Descriptor File (DDF). The following diagram illustrates the pbu_$ddf_t data type.



*See the "Field Description" that appears later for for field names of bits.

predefined type	byte offset		field name
	14:	char	device_sn
	30:	char	call_lib_name
	94:	char	int_lib_name
	158:	char	init_ep
	190:	char	cleanup_ep
pbu_$ddf_int_list_entry_t	222:		int_list
pbu_$iova_t	478:	integer	csr_page_iova
	480:	integer	stack_size
	482:	char	rev
	490:	char	sn
	506:	char	user_info
	570:	integer	csr_base_offset
pbu_$iova_t	572:	integer	memory_iova
	574:	integer	memory_size
pbu2_$iova_t	576:	integer32	csr_page_iova2
pbu2_$iova_t	580:	integer32	memory_iova2
	584:	integer32	memory_size2
	588:	integer	dma_channel
pbu_$iova_t	590:	integer	at_csr_high
pbu_$ddf_int_list_entry_t	592:		int_list2
	n:		

Field Description

sio_number
SIO number in old DDFs.

version
DDF version number.

unit_number
Unit number of this device.

flags
A bit mask that contains Boolean values, indicating device attributes. The following table lists the bit numbers within the mask, the record field names, and a short description of each attribute:

Bit	Field Name	Description
0	large	20-bit controller
1	share	Memory-mapped controller mapped in global address space
2	vme	VMEbus device
3	sys	Reserved
4	debug	User debug flag
5	at	PC AT compatible device
6	multiple	Driver supports multiple devices
7	pad	

dev_uid
Device uid (for locking).

device_sn
Unit serial number.

call_lib_name
Pathname of call-side library.

int_lib_name
Pathname of interrupt-side library.

init_ep
Entry point of driver's initialization routine.

cleanup_ep
Entry point of driver's cleanup routine.

int_list
Interrupt request level and name.

Field Description (Cont.)

csr_page_iova
Address of device CSR page.

stack_size
Size (in bytes) of interrupt stack.

rev
Optional revision number.

sn
User-specified serial number.

user_info
User-specified information.

csr_base_offset
Offset within CSR page of CSR base.

memory_iova
Memory-mapped controller base.

memory_size
Memory-mapped controller memory size.

The following fields are valid in Version 3 DDFs only:

csr_page_iova2
Address of VMEbus and PC AT compatible device CSR page.

memory_iova2
Memory-mapped controller base.

memory_size2
Memory-mapped controller memory size.

dma_channel
PC AT compatible device channel number.

at_csr_high
High PC AT compatible I/O address (if greater than 8-byte area).

int_list2
Interrupt request level and name (VMEbus and PC AT compatible devices).

pbu_$dma_channel_t	A 2-byte integer. The DMA channel number used by PC AT compatible devices. Possible values are integers between 0 and 7.

pbu_$dma_direction_t	A 2-byte integer. Used with **pbu_$dma_start** to specify a read or write DMA operation. One of the following predefined values: **pbu_dma_read** The PC AT compatible controller reads processor memory. **pbu_dma_write** Processor memory writes to the PC AT compatible controller.
pbu_$dma_opts_t	A 2-byte integer. Specifies various DMA modes on the PC AT compatible bus. Any combination of the following predefined values: **pbu_dma_adr_decr** DMA hardware decrements the address to or from which data is transferred. The default is to increment. **pbu_dma_auto_init** DMA hardware reinitializes itself after completing data transfer. **pbu_dma_cascade** Sets DMA channel in cascade mode; use with devices that can request bus mastership doing DMA with their own DMA hardware. **pbu_dma_ext_mem** DMA to PC AT compatible or PC XT* compatible extension memory.
pbu_$get_ec_key_t	A 2-byte integer. Specifies the eventcount to get. Currently, only the following predefined value is supported: **pbu_$get_device_ec** Get device EC.

* PC XT is a registered trademark of International Business Machines Corporation.

pbu_$info_t

I/O bus information. The following diagram illustrates the pbu_$info_t data type:

predefined type	byte offset		field name
	0:	integer	version
pbu_$bus_t	2:		bus_types
pbu_$iomap_t	4:		iomap_types

pbu_$interrupt_flags_t

A 2-byte integer. Flags returned from the device driver's interrupt routine that specify actions the System Interrupt Handler is to perform. One or both of the following predefined values:

pbu_$interrupt_advance
Advance the device's eventcount.

pbu_$interrupt_enable
Enable interrupts from the device.

pbu_$interrupt_return_t

A 2-byte integer. A set of pbu_$interrupt_flags_t.

pbu_$iomap_t

A 2-byte integer. Indicates whether the node's I/O hardware includes an I/O map for the specified bus type. Returns any combination of the following values:

pbu_multibus_iomap
The node is equipped with the MULTIBUS and includes an I/O map.

pbu_atbus_iomap
The node is equipped with the PC AT compatible bus and includes an I/O map.

pbu_$iova_t

A 2-byte integer. A physical address on the I/O bus.

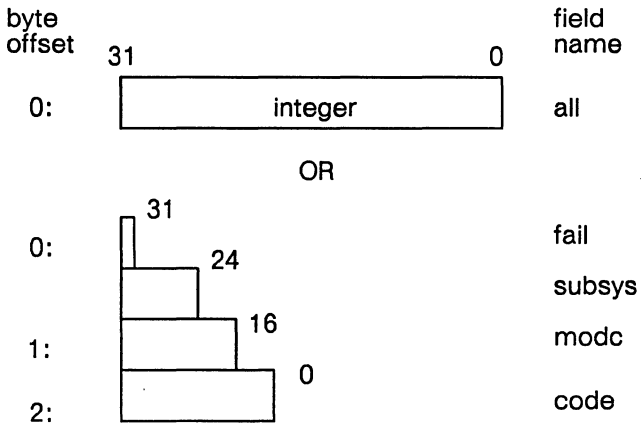
pbu2_$iova_t

A 4-byte integer. A physical address on the I/O bus.

pbu_$opts_t	A 2–byte integer. Available byte–swapping options when using **pbu_$control**. One or more of the following predefined values: **pbu_map_r** Maps pages of processor memory read–only. **pbu_map_rw** Maps pages of processor memory read–write. **pbu_swap_off** Swaps bytes during byte transfers only. **pbu_swap_words** Preserves byte order for character string transfers; swaps bytes for integer transfers. **pbu_swap_bytes** Preserves byte order for integer transfers; swaps bytes for character string transfers.
pbu_$pa_list_t	An array of up to 64 univ_ptrs. A list of physical addresses that locate the buffer in processor memory.
pbu_$unit_t	A 2–byte integer. Device unit number. Possible values are integers between 0 and **pbu_$max_vme_unit**.
pbu_$unit_set_t	A 2–byte integer. A set of pbu_$unit_t.
pbu_$wait_index_t	A 2–byte integer. Indicates the event that caused **pbu_$wait** to return. Possible values are integers between 0 and 2.
pbu_$wire_spec_opt_t	A 2–byte integer. Options when wiring an I/O buffer with **pbu_$wire_special**. Only one predefined value is currently available: **pbu_$wired_buffer** Verifies whether the buffer is already wired.

status_$t

A status code. The following diagram illustrates this data type:



Field Description

all
All 32 bits in the status code.

fail
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

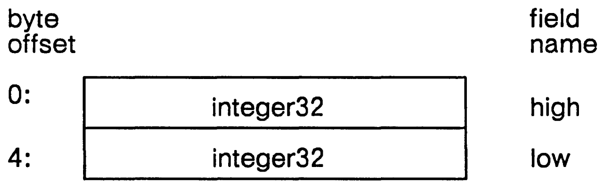
subsys
The subsystem that encountered the error (bits 24–30).

modc
The module that encountered the error (bits 16–23).

code
A signed number that identifies the type of error that occurred (bits 0–15).

uid_$t

Unique identifier for an object. The following diagram illustrates the uid_$t data type:



Field Description

high
The high 4 bytes of the UID.

low
The low 4 bytes of the UID.

univ_ptr

A 4-byte integer. A universal pointer type.

B.2 GPIO Procedures and Functions

Table B–1 lists the GPIO calls described in this section, along with the type of bus each call supports.

Table B–1. GPIO Procedures and Functions

GPIO Call	Supported Bus
pbu_$acquire	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$acquire_stream	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$advance_ec	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$allocate_ec	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$allocate_map	MULTIBUS
pbu_$control	MULTIBUS
pbu_$device_interrupting	MULTIBUS and PC AT Compatible Bus
pbu_$disable_device	MULTIBUS and PC AT Compatible Bus
pbu_$dma_start	PC AT Compatible Bus
pbu_$dma_stop	PC AT Compatible Bus
pbu_$enable_device	MULTIBUS and PC AT Compatible Bus
pbu_$free_map	MULTIBUS
pbu_$get_ec	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$get_info	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$map	MULTIBUS
pbu_$map_controller	MULTIBUS
pbu_$mem_ptr	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$read_csr	MULTIBUS and VMEbus
pbu_$release	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$release_ec	MULTIBUS, VMEbus, and PC AT Compatible Bus

(Continued)

Table B-1. GPIO Procedures and Functions (Cont.)

GPIO Call	Supported Bus
pbu_$unmap	MULTIBUS
pbu_$unmap_controller	MULTIBUS
pbu_$unwire	MULTIBUS
pbu_$wait	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu_$wire	MULTIBUS
pbu_$wire_special	PC AT Compatible Bus and VMEbus
pbu_$write_csr	MULTIBUS and VMEbus
pbu2_$allocate_map	MULTIBUS and PC AT Compatible Bus with I/O Map
pbu2_$dma_start	PC AT Compatible Bus with I/O Map
pbu2_$dma_stop	PC AT Compatible Bus with I/O Map
pbu2_$free_map	MULTIBUS and PC AT Compatible Bus with I/O Map
pbu2_$map	MULTIBUS and PC AT Compatible Bus with I/O Map
pbu2_$map_controller	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu2_$unmap	MULTIBUS and PC AT Compatible Bus with I/O Map
pbu2_$unmap_controller	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu2_$unwire	MULTIBUS, VMEbus, and PC AT Compatible Bus
pbu2_$wire	MULTIBUS, VMEbus, and PC AT Compatible Bus

NAME

pbu_$acquire Acquires control of a peripheral device.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$acquire(
    name_$long_pname_t pathname,
    short              &namelen,
    boolean            &xdebug,
    pbu_$unit_t        *unit,
    status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$acquire(
    in  pathname:      univ name_$long_pname_t;
    in  namelen:       integer;
    in  xdebug:        boolean;
    out unit:          pbu_$unit_t;
    out status:        status_$t);
```

DESCRIPTION

The **pbu_$acquire** routine acquires control of a device as follows:

1. Locates the DDF, using the specified pathname, and maps it into the address space of the user process from which **pbu_$acquire** was called.

2. Locks the device's DDF.

3. Copies information from the DDF into internal I/O tables.

If necessary, **pbu_$acquire** also establishes the device driver entry points and data structures needed to communicate with the device:

1. Locates the device driver routines and maps them into user–process address space.

2. Wires the interrupt stack and associated interrupt code and data.

3. Maps the CSR page for the device into user–process address space.

4. Executes (or calls) the initialization routine specified in the DDF.

Normally, the **aqdev** command calls **pbu_$acquire**, but user–written routines can call it and **pbu_$acquire** can also be called directly by application programs.

The input and output parameters are as follows:

debug_flag	A Boolean value that indicates whether load information is printed. See the **aqdev** command in Appendix A.
namelen	The length in characters of the specified pathname. This is a C unsigned short integer or a 2–byte Pascal integer.
pathname	The pathname of the DDF for the device to be acquired. Specify this parameter as an array of characters.
unit	The unit number in pbu_$unit_t format for use in subsequent calls to PBU routines.
status	Completion status in status_$t format.

NAME

pbu_$acquire_stream	Acquires control of devices from an extensible streams type manager.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$acquire_stream(
    xoid_$t            &xoid,
    boolean            &xdebug,
    pbu_$unit_t        *unit,
    status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$acquire_stream(
    in  xoid_$t:       xoid;
    in  xdebug:        boolean;
    out unit:          pbu_$unit_t;
    out status:        status_$t);
```

DESCRIPTION

The **pbu_$acquire_stream** procedure works in the same way as **pbu_$acquire**. The difference is that **pbu_$acquire_stream** uses only *xoid* as an input parameter.

The input and output parameters are described as follows:

status	Returned status in status_$t format.
unit	The unit number in pbu_$unit_t format for use in subsequent calls to pbu routines.
xoid	The xoid of the DDF in xoid_$t format. A xoid is an extended object identifier, a unique identifier of an object. It is 16 bytes long and has a predefined type of xoid_$t. Every object has its own unique xoid. Refer to *Using the OPEN Systems Toolkit to Extend Your Domain Streams* manual for an explanation of xoid.

NAME

pbu_$advance_ec Advances a device eventcount.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$advance_ec(
     pbu_$unit_t        &unit,
     ec2_$ptr_t         &ec2p,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$advance_ec(
     in  unit:          pbu_$unit_t;
     in  ec2p:          ec2_$ptr_t;
     out status:        status_$t);
```

DESCRIPTION

The **pbu_$advance_ec** routine advances an eventcount from a special pool of eventcounts in wired memory. It enables the interrupt handler of a shared driver to selectively advance a particular eventcount based on the type of interrupt. See also the descriptions of **pbu_$allocate_ec** and **pbu_$release_ec**, as well as the discussion of global drivers in Chapter 9, Section 9.1.

The input and output parameters are described as follows:

ec2p	The eventcount pointer, in ec2_ptr_t format, returned from the GPIO call **pbu_$allocate_ec**. This is a 4-byte integer.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a 2-byte Pascal integer or C unsigned short integer.

NAME

pbu_$allocate_ec Allocates a new device eventcount.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

ec2_$ptr_t pbu_$allocate_ec(
    pbu_$unit_t         &unit,
    status_$t           *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu_$allocate_ec(
    in  unit:           pbu_$unit_t;
    out status:         status_$t); ec2_$ptr_t;
```

DESCRIPTION

The **pbu_$allocate_ec** routine allocates an eventcount from a special pool of eventcounts in wired memory. It is designed for use with shared drivers that occupy global memory. See also the descriptions of **pbu_$advance_ec**, and **pbu_$release_ec** as well as the discussion of global drivers in Chapter 9, Section 9.1.

The input and output parameters are described as follows:

status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a 2-byte Pascal integer or C unsigned short integer.

NAME

pbu_$allocate_map Allocates I/O map space.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

pbu_$iova_t pbu_$allocate_map(
     pbu_$unit_t          &unit,
     pinteger             &length,
     boolean              &force_flag,
     pbu_$iova_t          &iova,
     status_$t            *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

 function pbu_$allocate_map(
     in  unit:          pbu_$unit_t;
     in  length:        pinteger;
     in  force_flag:    boolean;
     in  iova:          pbu_$iova_t;
     out status:        status_$t); pbu_$iova_t;
```

DESCRIPTION

The **pbu_$allocate_map** routine reserves an area of MULTIBUS address space for subsequent DMA transfers. The function allocates the number of I/O map entries that correspond to the required number of pages of MULTIBUS memory plus one (to enable mapping of buffers that are not page aligned).

In general, a driver may allocate only one area of the I/O map for a given device at any time. However, drivers can allocate a second area of the I/O map for a device by calling **pbu[2]_$map_controller**.

The input and output parameters are described as follows:

force_flag	A Boolean value that indicates whether or not a specific MULTIBUS address is to be assigned. For C programs, refer to Appendix C, Subsection C.2.3 for information about using Boolean values in C.
iova	If the *force_flag* parameter is "true", the MULTIBUS address in pbu_$iova_t format to be assigned as the starting address of the portion of MULTIBUS address space to be allocated.
length	The length in bytes of MULTIBUS address space for which an area of the I/O map is to be allocated. This is a C unsigned short integer or a 2-byte Pascal integer.
returned_iova	The MULTIBUS address in pbu_$iova_t format that marks the start of MULTIBUS address space allocated by **pbu_$allocate_map**.
status	Completion status in status_$t format.
unit	The unit number of the device in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$control Specifies mapping controls.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$control(
    pbu_$unit_t         &unit,
    pbu_$opts_t         &opts,
    pbu_$opts_t         *old_opts,
    linteger            &reserved,
    status_$t           *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$control(
     in  unit:          pbu_$unit_t;
     in  opts:          pbu_$opts_t;
     out old_opts:      pbu_$opts_t;
     in  reserved:      univ linteger;
     out status:        status_$t);
```

DESCRIPTION

The **pbu_$control** routine modifies the byte-swapping and protection hardware on 20-bit MULTIBUS implementations. The byte-swapping options are described in Chapter 1, Section 1.4.

The input and output parameters are described as follows:

opts Specifies one or more of the following options in pbu_$opts_t format:

pbu_map_r:
Pages of processor memory are mapped read-only, that is, a MULTIBUS controller cannot modify the data on the page.

pbu_map_rw:
Pages of processor memory are mapped read-write. *This is the default.*

pbu_swap_off:
No byte swapping occurs except during byte transfers. *This is the default.*

pbu_swap_words:
Byte transfers are unchanged; word transfers have their bytes reversed.

pbu_swap_bytes:
Word transfers are unchanged; byte transfers are swapped.

If a null set "[]" is specified, nothing is changed, and the current settings are returned in *old_opts*.

old_opts	The previous setting of the options in pbu_$opts_t format.
reserved	Reserved for future use; pass in 0.
status	Completion status in status_$t format.
unit	The unit number of the device in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$device_interrupting Checks for device interrupts.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

boolean pbu_$device_interrupting(
     pbu_$unit_t        &unit,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu_$device_interrupting(
    in  unit:           pbu_$unit_t;
    out status:         status_$t); boolean;
```

DESCRIPTION

The **pbu_$device_interrupting** routine reads the current state of the device's interrupt request line. This routine can be called from a user-written interrupt routine. Because it reads the current state of the interrupt line, the information this routine returns is not always reliable. The interrupt signal may disappear before the routine is able to read it.

The **pbu_$device_interrupting** routine cannot be used with VMEbus devices.

The input and output parameters are described as follows:

boolean	A value that indicates (if "true") that the device's interrupt line is asserted. For C programs, refer to Appendix C, Subsection C.2.3 for information about using Boolean values in C.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$disable_device Disables interrupts from a peripheral device.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$disable_device(
     pbu_$unit_t       &unit,
     status_$t         *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$disable_device(
    in  unit:          pbu_$unit_t;
    out status:        status_$t);
```

DESCRIPTION

The **pbu_$disable_device** routine prevents a device from requesting interrupts by setting its interrupt mask bit.

The system automatically disables interrupts from a device as follows:

- After it is acquired
- During interrupt processing
- When the device is released

The **pbu_$disable_device** routine cannot be used with VMEbus devices.

The input and output parameters are described as follows:

status	Completion status in status_$t format.
unit	The unit number of the device in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

NAME

pbu_$dma_start Starts a DMA operation.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$dma_start(
    pbu_$unit_t          &unit,
    pbu_$dma_channel_t &chan,
    pbu_$dma_direction_t &direction,
    pbu_$buffer_t        buffer,
    linteger             &length,
    pbu_$dma_opts_t      &opts,
    status_$t            *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$dma_start(
    in  unit:           pbu_$unit_t;
    in  chan:           pbu_$dma_channel_t;
    in  direction:      pbu_$dma_direction_t;
    in  buffer:         univ pbu_$buffer_t;
    in  length:         linteger;
    in  opts:           pbu_$dma_opts_t;
    out status:         status_$t);
```

DESCRIPTION

The **pbu_$dma_start** and **pbu_$dma_stop** routines are paired functions for use with PC AT compatible devices. They should surround each DMA operation, whether successful or not. The **pbu_$dma_start** routine prepares the system DMA hardware for the controller's operation. The driver must call this routine before issuing any I/O commands to the device. After **pbu_$dma_start** is called, the controller can begin its operation. Before calling **pbu_$dma_start** again, the driver must call **pbu_$dma_stop**. Refer also to the description of **pbu_$dma_stop**.

The **pbu_$dma_start** routine can be called from the driver's interrupt side.

For bus–master devices, **pbu_$dma_start** must be called with the option **pbu_dma_cascade** in order to reserve the DMA channel and to provide for proper bus arbitration.

If you are designing your driver to run on the DN4000, you must call **pbu2_$dma_start,** which is described later in this appendix. Refer also to Chapter 3, Section 3.6 for additional information.

The input and output parameters are described as follows:

buffer
: The buffer to be mapped, specified as a universal array of characters, in pbu_$buffer_t format. It must be page aligned.

channel
: This is a C unsigned short integer or a 2–byte Pascal integer in pbu_$dma_channel_t format. Specifies the number (0–7) of the channel to be started.

direction
: The direction of the data transfer in pbu_$dma_direction_t format. Specify one of the following options:

  pbu_dma_read
  Controller to processor memory.

  pbu_dma_write
  Processor memory to controller.

length
: The length of the buffer in bytes. This is a C unsigned short integer or a 2–byte Pascal integer and must be greater than 0 and less than or equal to 1024. If your driver is for a 16–bit device, the length must be expressed as an even number.

opts
: Specifies any combination of the following options in pbu_$dma_opts_t format:

  pbu_dma_auto_init
  Specifies that DMA hardware is to reinitialize itself after completing transfer, using the *buffer* and *length* parameters supplied with the call. Note that **pbu_$dma_start** converts the *length* parameter from bytes to words. For more information, refer to the description of "autoinitialize" for the 8237A in Intel's *Microsystem Components Handbook.*

  pbu_dma_adr_decr
  Specifies that DMA operations decrement the address to or from which data is transferred. The default is that DMA transfers are made to increasing memory addresses.

pbu_dma_cascade
Sets the processor's DMA hardware in cascade mode so that a bus–master device can use its own DMA hardware. It is a way of arbitrating for the PC AT compatible bus. You must specify this option if you want the device to use its own DMA hardware.

pbu_dma_ext_mem
Specifies that the DMA transfer is to PC AT compatible or PC XT compatible extension memory, not processor memory.

status Completion status in status_$t format.

unit The unit number of the device in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

NAME

pbu_$dma_stop Stops a DMA operation.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

long pbu_$dma_stop(
     pbu_$unit_t        &unit,
     pbu_$dma_channel_t &chan,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu_$dma_stop(
    in  unit:          pbu_$unit_t;
    in  chan:          pbu_$dma_channel_t;
    out status:        status_$t); integer32;
```

DESCRIPTION

The **pbu_$dma_start** and **pbu_$dma_stop** routines are paired functions for use with PC AT compatible devices. They should surround each I/O operation, whether successful or not. The **pbu_$dma_start** routine prepares DMA hardware for the controller's operation. After the controller completes its operation, the driver must call **pbu_$dma_stop** to get status from DMA hardware to ensure that the hardware completed its operation as well. Even if the controller reports an error, the driver must call **pbu_$dma_stop**. The driver may ignore the status returned by **pbu_$dma_stop**, but if the controller had a problem, it is likely that the DMA operation did not run to completion. The call to **pbu_$dma_stop** must, in any case, be made so that software can reset its knowledge of who is using the DMA channel.

The **pbu_$dma_stop** routine can be called from the driver's interrupt side.

If you are designing your driver to run on the DN4000, you must call **pbu2_$dma_start,** which is described later in this appendix. Refer also to Chapter 3, Section 3.6 for additional information.

The input and output parameters are described as follows:

channel	This is a C unsigned short integer or a 2-byte Pascal integer in pbu_$dma_channel_t format. Specifies the number (0-7) of the channel to be stopped.
resid_cnt	A 4-byte integer that specifies the residual count in bytes of the amount of data (if any) that was not transferred during the last DMA operation. This return value should only be 0 if there is nothing left to transfer. The purpose of this parameter is to tell the driver if it needs to perform another DMA operation, and if so, how large the buffer length parameter for **pbu[2]_$dma_start** should be.
status	Completion status in status_$t format.
unit	The unit number of the device in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$enable_device Enables interrupts from a peripheral device.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$enable_device(
     pbu_$unit_t       &unit,
     status_$t         *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$enable_device(
    in  unit:          pbu_$unit_t;
    out status:        status_$t);
```

DESCRIPTION

The **pbu_$enable_device** routine enables interrupt requests from a device by clearing its interrupt mask bit in the Peripheral Interrupt Controller (PIC).

Note that a user–written interrupt routine cannot call **pbu_$enable_device**. The routine can optionally enable device interrupts by returning the appropriate function value to the System Interrupt Handler.

The **pbu_$enable_device** routine cannot be used with VMEbus devices.

The input and output parameters are described as follows:

status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

NAME

pbu_$free_map Releases the I/O map area previously allocated to a device.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$free_map(
    pbu_$unit_t         &unit,
    status_$t           *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$free_map(
    in  unit:           pbu_$unit_t;
    out status:         status_$t);
```

DESCRIPTION

The **pbu_$free_map** routine releases the area of the I/O map previously allocated by the GPIO call **pbu_$allocate_map** for MULTIBUS devices.

The input and output parameters are described as follows:

status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

NAME

pbu_$get_ec Retrieves the eventcount associated with a device.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$get_ec(
    pbu_$unit_t        &unit,
    pbu_$get_ec_key_t  &key,
    c2_$ptr_t         &ec2p,
    status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$get_ec(
    in  unit:          pbu_$unit_t;
    in  key:           pbu_$get_ec_key_t;
    in  ec2p:          ec2_$ptr_t;
    out status:        status_$t);
```

DESCRIPTION

The **pbu_$get_ec** routine returns an eventcount identifier that the driver or the application can place into a list of eventcount identifiers that they pass to **ec2_$wait**. Drivers need only call this routine once while the device is acquired and should save the eventcount pointer until the device is released. However, no errors occur if drivers call **pbu_$get_ec** more than once.

Drivers (or any other programs) must not rely solely upon eventcounts to indicate the occurrence of an event; they should provide an additional mechanism to determine whether an event occurred. Refer to Chapter 6, Subsection 6.3.2.

The input and output parameters are described as follows:

ecp	A pointer to the eventcount for the device in ec2_$ptr_t format.
key	The key that specifies which eventcount to get in pbu_$get_ec_key_t format. Currently, the only value allowed is **pbu_$get_device_ec**.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$get_info Gets information concerning I/O bus type and I/O map.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$get_info(
     pinteger          &length,
     pbu_$info_t       *info,
     status_$t         *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$get_info(
     in  length:       pinteger;
     out info:         pbu_$info_t;
     out status:       status_$t);
```

DESCRIPTION

This procedure returns information concerning the presence of the I/O bus type and the I/O map. With this information, you can design your driver to run on different node models. Thus, for example, if you want to design a driver that could run on either the DN3000 or the DN4000, your driver can use the information returned by **pbu_$get_info** to determine whether or not to perform a DMA transfer of more than 1 KB of data.

The input and output parameters are described as follows:

length	The length of info in bytes. This is a C unsigned short integer or a 2-byte Pascal integer.
info	The name of the record, in pbu_$info_t format, in which bus information is returned.
status	Completion status in status_$t format.

NAME

pbu_$map Maps an I/O buffer.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

pbu_$iova_t pbu_$map(
     pbu_$unit_t         &unit,
     pbu_$buffer_t       buffer,
     pinteger            &length,
     pbu_$iova_t         &iova,
     status_$t           *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu_$map(
     in  unit:           pbu_$unit_t;
     in  buffer:         univ pbu_$buffer_t;
     in  length:         pinteger;
     in  iova:           pbu_$iova_t;
     out status:         status_$t); pbu_$iova_t;
```

DESCRIPTION

The **pbu_$map** routine establishes the mapping between the buffer in processor address space and MULTIBUS address space. Drivers must call this routine before using the buffer for I/O operations and only after they have called **pbu_$allocate_map** and **pbu_$wire** (the buffer must be wired before it can be passed to pbu_$map). User-written interrupt routines can call **pbu_$map**.

The address specified as a parameter to **pbu_$map** need not be the address that **pbu_$allocate_map** returned; the address can lie on any page that corresponds to the allocated area of the I/O map. In this way, drivers can map several different buffers into different sections of the allocated I/O map area at the same time.

The input and output parameters are described as follows:

buffer	The buffer to be mapped, specified as an array of characters.
iova	A page-aligned MULTIBUS address within the I/O map area allocated by **pbu_$allocate_map** in pbu_$iova_t format. This is a C unsigned short integer or a 2-byte Pascal integer.
length	The length in bytes of the buffer. This is a C unsigned short integer or a 2-byte Pascal integer.
returned_iova	The MULTIBUS address that marks the start of the buffer in MULTIBUS address space in pbu_$iova_t format. This is a C unsigned short integer or a 2-byte Pascal integer.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$map_controller Maps controller memory to processor address space.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

char *pbu_$map_controller(
      pbu_$unit_t        &unit,
      pbu_$iova_t        &iova,
      pinteger           &length,
      status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu_$map_controller(
     in  unit:          pbu_$unit_t;
     in  iova:          pbu_$iova_t;
     in  length:        pinteger;
     out status:        status_$t); univ_ptr;
```

DESCRIPTION

The **pbu_$map_controller** routine maps controller memory to processor address space. Device drivers can map only one area of controller memory per device at a time.

Memory that is mapped with **pbu_$map_controller** *must* be unmapped with **pbu_$unmap_controller**.

Refer to Chapter 7, Section 7.2 for information on referencing controller memory.

The input and output parameters are described as follows:

address	The virtual address of the first byte of the controller's mapped memory in univ_ptr format. For an equivalent of univ_ptr in C, refer to Appendix C, Subsection C.2.4.

iova	The MULTIBUS address that marks the start of controller memory in pbu_$iova_t format. This is a C unsigned short integer or a 2-byte Pascal integer. The address must lie on a page boundary and must be smaller than 32 KB.
length	The length in bytes of controller memory. This is a C unsigned short integer or a 2-byte Pascal integer. The length to be mapped must be between 0 and 32 KB, and the sum of the length and the MULTIBUS address must be less than 32 KB.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

Possible error messages associated with this routine include:

PBU_$BAD_UNIT

The specified unit number is invalid.

PBU_$NOT_ACQUIRED

The device has not been acquired.

PBU_$ALREADY_MAPPED

Controller memory has already been mapped.

PBU_$BAD_IOVA

The specified MULTIBUS address is larger than 32 KB or causes the sum of length and address to exceed 32 KB.

PBU_$BAD_LEN

The specified length is not between 0 and 32 KB or causes the sum of length and address to exceed 32 KB.

Errors can also include those errors generated by **pbu_$allocate_map**, the most common of which is that the requested memory is already allocated. If this error is generated, check the DMA devices in the configuration to see if they are using the desired MULTIBUS addresses.

NAME

pbu_$mem_ptr Returns the address and length of a shared controller.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

char *pbu_$mem_ptr(
     name_$long_pname_t &pathname,
     short              &namelen,
     linteger           *mem_len,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu_$mem_ptr(
     in  pathname:       univ name_$long_pname_t;
     in  namelen:        integer;
     out mem_len:        linteger;
     out status:         status_$t); univ_ptr;
```

DESCRIPTION

The **pbu_$mem_ptr** routine returns the address of a shared memory-mapped controller mapped in global address space to any application that wants to use it. The following example shows how to use **pbu_$mem_ptr** so that a process that wants to access the shared controller can obtain its address:

```
REPEAT
           mem_pointer := pbu_$mem_ptr(ddf_name,
                          sizeof(ddf_name), mem_len, status);
           if status.all = pbu_$device_not_mapped then
                   time_$wait(time_$relative, delay_time,
                              status2)
           else begin
                   error_$print(status);
                   goto error_exit;
                   end;
UNTIL status.all = 0
```

The input and output parameters are described as follows:

address	The virtual address of the first byte of the controller's mapped memory in univ_ptr format. For an equivalent of univ_ptr in C, refer to Appendix C, Subsection C.2.4.
ddf_length	The length in characters of the specified pathname. This is a C unsigned short integer or a 2-byte Pascal integer.
length	The length in bytes of the area for the mapped controller. This is a C unsigned long integer or a 4-byte Pascal integer.
pathname	The pathname of the DDF for the shared controller. Specify this parameter as an array of characters.
status	Completion status in status_$t format.

NAME

pbu_$read_csr Reads a device's Control and Status Register (CSR).

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$read_csr(
     pbu_$unit_t       &unit,
     char              &csr,
     pinteger          *value,
     boolean           &word_flag,
     status_$t         *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$read_csr(
     in  unit:         pbu_$unit_t;
     in  csr:          univ char;
     out value:        pinteger;
     in  word_flag:    boolean;
     out status:       status_$t);
```

DESCRIPTION

Device drivers can call **pbu_$read_csr** during initialization to determine whether a device is physically present on the bus. If a read to the device's CSR causes a bus time–out error, this routine suppresses normal bus error handling and sets the status code to reflect the error.

If the specified CSR does not reside within the device's CSR page, **pbu_$read_csr** returns an error value. For a memory-mapped controller, **pbu_$read_csr** returns an error if the address does not reside within the area of processor address space to which the memory has been mapped.

The **pbu_$read_csr** routine is typically used in the initialization routine, but other call–side routines can call it.

> **NOTE:** Drivers for PC AT compatible controllers should not use this call to test if the controller is present on the bus. For more information, refer to Chapter 3, Section 3.3.

The input and output parameters are described as follows:

csr	The control and status register to be read in universal character format (C type char or Pascal type UNIV char). Refer to Appendix C, Section C.1 for more information.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.
value	The result of the read, located in the low-order (right-hand) byte if a byte read was performed. This is a C unsigned short integer or a 2-byte Pascal integer.
word_flag	A Boolean value that specifies whether a word or byte read is to be performed ("false" = byte read, "true" = word read). For C programs, refer to Appendix C, Subsection C.2.3 for information about using Boolean values in C.
status	Completion status in status_$t format.

NAME

pbu_$release Releases an acquired device.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$release(
     pbu_$unit_t       &unit,
     boolean           &force,
     status_$t         *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$release(
     in  unit:         pbu_$unit_t;
     in  force:        boolean;
     out status:       status_$t);
```

DESCRIPTION

To release control of a device, **pbu_$release** performs these functions:

- Unloads the device driver
- Unwires all wired procedures and data pages
- Deallocates any I/O map areas that are still allocated
- Unmaps any mapped controller memory
- Calls the user-written cleanup routine whose entry point is specified in the DDF for the device. This routine ensures that there are no I/O operations in progress and clears any pending interrupts generated by the device.

Currently, **pbu_$release** is called only from the **aqdev** command or from the device acquisition program. Since **pbu_$release** unloads the driver, it should not be called by driver routines.

The input and output parameters are described as follows:

force_flag	A Boolean value that indicates whether or not the cleanup routine can abort the device release operation. If this parameter is set to true, the device is released regardless of the status returned by the cleanup routine. If this parameter is set to "false", the cleanup routine can abort the release procedure by returning a non-zero status code. Upon receipt of the status, **pbu_$release** aborts device release and returns to its caller.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$release_ec Releases an eventcount.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$release_ec(
     pbu_$unit_t        &unit,
     ec2_$ptr_t         &ec2p,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$release_ec(
     in  unit:          pbu_$unit_t;
     in  ec2p:          ec2_$ptr_t;
     out status:        status_$t);
```

DESCRIPTION

The **pbu_$release_ec** routine releases an eventcount allocated by **pbu_$allocate_ec** to a special pool of eventcounts in wired memory. It is designed for use with global drivers that occupy global memory. See also the descriptions of **pbu_$advance_ec** and **pbu_$allocate_ec,** as well as the discussion of global drivers in Chapter 9, Section 9.1.

The input and output parameters are described as follows:

ec2p	The eventcount pointer in ec2_ptr_t format, returned from the GPIO call **pbu_$allocate_ec**. This is a 4–byte integer.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

NAME

pbu_$unmap Unmaps an I/O buffer.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$unmap(
     pbu_$unit_t          &unit,
     pbu_$buffer_t        buffer,
     pinteger             &length,
     pbu_$iova_t          &iova,
     status_$t            *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$unmap(
     in  unit:            pbu_$unit_t;
     in  buffer:          univ pbu_$buffer_t;
     in  length:          pinteger;
     in  iova:            pbu_$iova_t;
     out status:          status_$t);
```

DESCRIPTION

The **pbu_$unmap** routine unmaps the buffer from MULTIBUS address space and invalidates the I/O map for the space occupied by the buffer.

Device drivers are not required to unmap previously mapped buffers; another call to **pbu_$map** that specifies the same area of the I/O map effectively unmaps the previously mapped buffer. The **pbu_$unmap** routine is used primarily to protect a buffer from erroneous references by a controller.

The **pbu_$unmap** routine can be called from interrupt–side routines.

The input and output parameters are described as follows:

buffer	The buffer to be unmapped, specified as an array of characters.
iova	The MULTIBUS address that marks the start of the buffer in pbu_$iova_t format. This address must be the address that pbu_$map returned (the actual start of the buffer).
length	The length in bytes of the area to be unmapped. This is a C unsigned short integer or a 2-byte Pascal integer.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$unmap_controller Unmaps a controller's memory from processor address space.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$unmap_controller(
     pbu_$unit_t        &unit,
     char               *&va,
     pinteger           &length,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$unmap_controller(
     in  unit:          pbu_$unit_t;
     in  va:            univ_ptr;
     in  length:        pinteger;
     out status:        status_$t);
```

DESCRIPTION

The **pbu_$unmap_controller** routine unmaps from processor address space the controller memory mapped by **pbu_$map_controller**. The whole mapped length must be unmapped.

The input and output parameters are described as follows:

address	The virtual address of the first byte of the controller's mapped memory in univ_ptr format. For an equivalent of univ_ptr in C, refer to Appendix C, Subsection C.2.4.
length	The length in bytes of the area to be unmapped. This is a C unsigned short integer or a 2–byte Pascal integer.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

Possible errors associated with this routine can include the following:

`PBU_$BAD_UNIT`
: The specified unit number is invalid.

`PBU_$NOT_ACQUIRED`
: The specified device has not been acquired.

`PBU_$NOT MAPPED`
: Controller memory has not been mapped.

NAME

pbu_$unwire Unwires an I/O buffer.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$unwire(
     pbu_$unit_t        &unit,
     pbu_$buffer_t      buffer,
     pinteger           &length,
     boolean            &touch,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$unwire(
     in  unit:          pbu_$unit_t;
     in  buffer:        univ pbu_$buffer_t;
     in  length:        pinteger;
     in  touch:         boolean;
     out status:        status_$t);
```

DESCRIPTION

The **pbu_$unwire** routine makes a buffer previously wired into processor memory with **pbu_$wire** available for MMU paging operations.

> **NOTE:** Buffers that are part of a driver's interrupt side must never be unwired.

The input and output parameters are described as follows:

buffer	The buffer to be unwired, specified as a universal array of characters.
length	The length in bytes of the buffer. This is a C unsigned short integer or a 2-byte Pascal integer.

modify_flag	A Boolean value that indicates whether the buffer pages being unwired should be marked as modified by an input I/O operation. This flag is needed because DMA does not set the page's modify bit in Memory Management Unit (MMU) tables. For more information, refer to Chapter 7, Subsection 7.1.3.2. For C programs, refer to Appendix C, Subsection C.2.3 for information about using Boolean values in C.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

NAME

pbu_$wait Waits for device interrupt.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

pbu_$wait_index_t pbu_$wait(
     pbu_$unit_t        &unit,
     long               &timeout,
     boolean            &quit_enable,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu_$wait(
     in  unit:          pbu_$unit_t;
     in  timeout:       integer32;
     in  quit_enable:   boolean;
     out status:        status_$t); pbu_$wait_index_t;
```

DESCRIPTION

Device drivers call **pbu_$wait** if they need only to wait for device interrupt, time-out, or quit fault. The routine performs these functions:

- Checks the device's eventcount to determine whether the System Interrupt Handler advanced it since the last time **pbu_$wait** was called. If an advance occurred, the routine returns.

- Checks for a positive time-out value. If the time-out value is less than or equal to 0, **pbu_$wait** returns; otherwise, it waits for the specified interval or until the System Interrupt Handler advances the eventcount.

To enable and disable quit faults during the wait, use the *quit_enable* parameter.

The input and output parameters are described as follows:

index	A C unsigned short integer or a 2-byte Pascal integer that corresponds to the event that caused **pbu_$wait** to return, in pbu_$wait_index_t format. Possible values are as follows: 0 = Eventcount advanced by the System Interrupt Handler 1 = Time-out 2 = Quit fault (CTRL/Q or CTRL/D)
quit_enable	A Boolean value that indicates whether or not quit faults are enabled during the wait. When this parameter is set to "true", quit faults terminate the wait state; when it is set to "false", quit faults are disabled. For information on quit faults, refer to Chapter 6, Subsection 6.3.1. For C programs, refer to Appendix C, Subsection C.2.3 for information about using Boolean values.
status	Completion status in status_$t format.
time-out	The length of time in milliseconds that the routine is to wait. This is a 4-byte integer (C or Pascal).
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu_$wire Wires an I/O buffer.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$wire(
     pbu_$unit_t         &unit,
     pbu_$buffer_t       buffer,
     pinteger            &length,
     status_$t           *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$wire(
     in  unit:           pbu_$unit_t;
     in  buffer:         univ pbu_$buffer_t;
     in  length:         pinteger;
     out status:         status_$t);
```

DESCRIPTION

The **pbu_$wire** routine makes the buffer's pages permanently resident in processor memory in preparation for an I/O operation. Drivers must wire I/O buffers before mapping them with **pbu_$map**.

Drivers need not wire interrupt-side buffers with **pbu_$wire** because **pbu_$acquire** automatically wires the data sections of the driver's interrupt routine(s) when the device is acquired. Refer to Chapter 7, Subsection 7.1.1.

The **pbu_$wire** routine returns an error if any page of the specified buffer is already wired.

The input and output parameters are described as follows:

buffer	The buffer to be wired, specified as a universal array of characters.
length	The length in bytes of the buffer. This is a C unsigned short integer or a 2–byte Pascal integer.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

NAME

pbu_$wire_special Wires an I/O buffer and returns its physical addresses.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$wire_special(
     pbu_$unit_t         &unit,
     pbu_$wire_spec_opt_t &opts,
     pbu_$buffer_t       buffer,
     linteger            &length,
     pbu_$pa_list_t      *pa_list,
     short               &max_cnt,
     short               *cnt,
     status_$t           *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$wire_special(
     in  unit:           pbu_$unit_t;
     in  opts:           pbu_$wire_spec_opt_t;
     in  buffer:         univ pbu_$buffer_t;
     in  length:         linteger;
     out pa_list:        univ pbu_$pa_list_t;
     in  max_cnt:        integer;
     out cnt:            integer;
     out status:         status_$t);
```

DESCRIPTION

The **pbu_$wire_special** routine is provided for VMEbus controllers and bus–master PC AT compatible controllers that use physical addresses to access processor memory. Like **pbu[2]_$wire, pbu_$wire_special** makes the buffer's virtual pages permanently resident in processor memory in preparation for an I/O operation. (Drivers must wire I/O buffers before starting an I/O operation.) The physical addresses returned in pa_list are 32–bit, page–aligned physical addresses in processor memory. To obtain the exact physical address of the start of the buffer, the byte offset within the page of the start of the buffer must be added to the first entry in pa_list:

```
buffer_start := pa_list[1] + (ptr(addr(buffer)) mod
                              bytes_per_page);
```

You should use **pbu2_$unwire** to unwire buffers wired with **pbu_$wire_special**.

The input and output parameters are described as follows:

buffer	The buffer to be wired, specified as a universal array of characters.
cnt	The number of entries returned in pa_list. This is a C unsigned short integer or a 2–byte Pascal integer.
length	The length in bytes of the buffer. This is a C unsigned long integer or a 4–byte Pascal integer.
max_cnt	The length (number of entries) in the pa_list array. This is a C unsigned short integer or a 2–byte Pascal integer.
opts	Specify one of the following options in pbu_$wire_spec_t format:

pbu_$wired_buffer	Verifies that buffer is already wired and returns error message if it is not.
[ ]	Wires buffer.

pa_list	An array of physical addresses in pbu_$pa_list_t format.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer. The unit number must refer to a VMEbus or demand–DMA PC AT compatible device.

NAME

pbu_$write_csr Writes to a device's control and status register.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$write_csr(
     pbu_$unit_t         &unit,
     char                &csr,
     pinteger            &value,
     boolean             &word_flag,
     status_$t           *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$write_csr(
     in  unit:           pbu_$unit_t;
     in  csr:            univ char;
     in  value:          pinteger;
     in  word_flag:      boolean;
     out status:         status_$t);
```

DESCRIPTION

Device drivers can call **pbu_$write_csr** during initialization to determine whether a device is physically present on the bus. If a write to the device's CSR causes a bus time-out error, this routine suppresses normal bus error handling and sets the status code to reflect the event.

If the specified CSR does not reside within the device's CSR page, **pbu_$write_csr** returns an error value. For a memory-mapped controller, **pbu_$write_csr** returns an error if the address does not reside within the processor address space to which the memory is mapped.

NOTE: Drivers for PC AT compatible controllers should not use this call to test if the controller is present on the bus. For more information, refer to Chapter 3, Section 3.3.

The input and output parameters are described as follows:

csr	The control and status register to be written in universal character format (C type char or Pascal type UNIV char). Refer to Appendix C, Section C.2.4 for more information.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.
value	The value to write into the CSR. If the routine is to perform a byte–write operation, the value is specified in the low–order (right–hand) byte of the integer. This is a C unsigned short integer or a 2–byte Pascal integer.
word_flag	A Boolean value that specifies whether a word or byte write is to be performed ("false" = byte write, "true" = word write). For C programs, refer to Appendix C, Subsection C.2.3 for information about using Boolean values.

NAME

pbu2_$allocate_map Allocates I/O map space.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

pbu_$iova_t pbu2_$allocate_map(
     pbu_$unit_t        &unit,
     linteger           &length,
     boolean            &force_flag,
     pbu2_$iova_t        &iova,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu2_$allocate_map(
     in  unit:          pbu_$unit_t;
     in  length:        linteger;
     in  force_flag:    boolean;
     in  iova:          pbu2_$iova_t;
     out status:        status_$t); pbu2_$iova_t;
```

DESCRIPTION

You must use **pbu2_$allocate_map** if your device driver supports a 20-bit MULTIBUS controller or an PC AT compatible bus controller designed to run on the DN4000.

The routine reserves an area of the bus address space for subsequent DMA transfers from either a 20-bit MULTIBUS controller or an PC AT compatible controller on the DN4000. The function allocates the number of I/O map entries that correspond to the required number of pages of bus memory plus one, to enable mapping of buffers that are not page aligned.

In general, a driver may allocate only one area of the I/O map for a given device at any time. However, drivers for 20-bit controllers can allocate a second area of the I/O map for a device by calling **pbu2_$map_controller**.

The input and output parameters are described as follows:

force_flag	A Boolean value that indicates whether or not a specific bus address is to be assigned. For C programs, refer to Appendix C, Subsection C.2.3 for information about using Boolean values.
iova	If the *force_flag* parameter is "true", the bus address in pbu2_$iova_t format to be assigned as the starting address of the portion of bus address space to be allocated. This is a 4–byte integer (in C and Pascal).
length	The length in bytes of bus address space for which an area of the I/O map is to be allocated. This is a 4–byte integer, in C and Pascal.
returned_iova	The bus address in pbu2_$iova_t format that marks the start of bus address space allocated by **pbu2_$allocate_map**. This is a 4–byte integer, in C and Pascal.
status	Completion status in status_$t format.
unit	The unit number of the device in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

NAME

pbu2_$dma_start Starts a DMA operation using the I/O map.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu2_$dma_start(
     pbu_$unit_t        &unit,
     pbu_$dma_channel_t &chan,
     pbu_$dma_direction_t &direction,
     pbu_$buffer_t      buffer,
     pbu2_$iova_t       &iova,
     linteger           &length,
     pbu_$dma_opts_t    &opts,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu2_$dma_start(
     in  unit:          pbu_$unit_t;
     in  chan:          pbu_$dma_channel_t;
     in  direction:     pbu_$dma_direction_t;
     in  buffer:        univ pbu_$buffer_t;
     in  iova:          pbu2_$iova_t;
     in  length:        linteger;
     in  opts:          pbu_$dma_opts_t;
     out status:        status_$t);
```

DESCRIPTION

The **pbu2_$dma_start** routine is for use in drivers running on the DN4000.

The **pbu2_$dma_start** and **pbu2_$dma_stop** routines are paired functions for use with PC AT compatible devices. They should surround each DMA operation, whether successful or not. The **pbu2_$dma_start** routine prepares the system DMA hardware for the controller's operation. The driver must call this routine before issuing any I/O commands to the device.

After **pbu2_$dma_start** is called, the controller can begin its operation. Before calling **pbu2_$dma_start** again, the driver must call **pbu2_$dma_stop**. Refer also to the description of **pbu2_$dma_stop**.

The **pbu2_$dma_start** routine can be called from the driver's interrupt side.

For bus-master devices, **pbu2_$dma_start** must be called with the option **pbu_dma_cascade** in order to reserve the DMA channel and to provide for proper bus arbitration.

If you are designing a driver to run only on the DN3000, you must call **pbu_$dma_start,** which is described earlier in this appendix. Refer to Chapter 3, Section 3.6 for additional information.

The input and output parameters are described as follows:

buffer
: The buffer to be mapped, specified as a universal array of characters in pbu_$buffer_t format. It must be page aligned.

channel
: A C unsigned short integer or a 2-byte Pascal integer in pbu_$dma_channel_t format. Specifies the number (0–7) of the channel to be started.

direction
: The direction of the data transfer, in pbu_$dma_direction_t format. Specify one of the following options:

    pbu_dma_read
    Controller to processor memory.

    pbu_dma_write
    Processor memory to controller.

iova
: The starting address of the portion of bus address space that has been allocated by **pbu2_$allocate_map** in pbu2_$iova_t format. This is a 4-byte integer, in C and Pascal.

length
: The length of the buffer in bytes. This is a 4-byte integer, in C and Pascal and must be greater than 0 and less than or equal to 64 KB for 8-bit devices, 128 KB for 16-bit devices, and 512 KB for bus-master devices. If your driver is for a 16-bit device, the length must be expressed as an even number.

opts

Specifies any combination of the following options in pbu_$dma_opts_t format:

pbu_dma_auto_init
Specifies that DMA hardware is to reinitialize itself after completing the transfer, using the buffer and length parameters supplied with the call. Note that **pbu2_$dma_start** converts the length parameter from bytes to words. For more information, refer to the description of "autoinitialize" for the 8237A in Intel's *Microsystem Components Handbook*.

pbu_dma_adr_decr
Specifies that DMA operations decrement the address to or from which data is transferred. The default is that DMA transfers are made to increasing memory addresses.

pbu_dma_cascade
Sets the processor's DMA hardware in cascade mode so that a bus-master device can use its own DMA hardware. It is a way of arbitrating for the PC AT compatible bus. You must specify this option if you want the device to use its own DMA hardware.

pbu_dma_ext_mem
Specifies that the DMA transfer is to PC AT compatible or PC XT compatible extension memory, not processor memory.

status

Completion status in status_$t format.

unit

The unit number of the device in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu2_$dma_stop Stops a DMA operation using the I/O map.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

long pbu2_$dma_stop(
     pbu_$unit_t        &unit,
     pbu_$dma_channel_t &chan,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu2_$dma_stop(
     in unit:           pbu_$unit_t;
     in chan:           pbu_$dma_channel_t;
     out status:        status_$t); integer32;
```

DESCRIPTION

The **pbu2_$dma_start** routine is for use in drivers running on the DN4000.

The **pbu2_$dma_start** and **pbu2_$dma_stop** routines are paired functions for use with PC AT compatible devices. They should surround each I/O operation, whether successful or not. The **pbu2_$dma_start** routine prepares DMA hardware for the controller's operation. After the controller completes its operation, the driver must call **pbu2_$dma_stop** to get status from the DMA hardware to ensure that the hardware completed its operation as well. Even if the controller reports an error, the driver must call **pbu2_$dma_stop**. The driver may ignore the status returned by **pbu2_$dma_stop**, but if the controller had a problem, it is likely that the DMA operation did not run to completion. The call to **pbu2_$dma_stop** must in any case be made so that software can reset its knowledge of who is using the DMA channel.

The **pbu2_$dma_stop** routine can be called from the driver's interrupt side.

If you are designing a driver to run only on the DN3000, you must call **pbu_$dma_stop,** which is described earlier in this appendix. Refer to Chapter 3, Section 3.6 for additional information.

The input and output parameters are described as follows:

channel	A C unsigned short integer or a 2-byte Pascal integer in pbu_$dma_channel_t format. Specifies the number (0–7) of the channel to be stopped.
resid_cnt	A 4-byte integer that specifies the residual count in bytes of the amount of data (if any) that was not transferred during the last DMA operation. This return value should only be 0 if there is nothing left to transfer. The purpose of this parameter is to tell the driver if it needs to perform another DMA operation, and if so, how large the buffer length parameter for **pbu[2]_$dma_start** should be.
status	Completion status in status_$t format.
unit	The unit number of the device in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu2_$free_map Releases a previously allocated I/O map area.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu_$free_map(
     pbu_$unit_t        &unit,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu_$free_map(
     in  unit:          pbu_$unit_t;
     out status:        status_$t);
```

DESCRIPTION

This routine releases the area of the I/O map previously allocated by the call **pbu2_$allocate_map**. You must use **pbu2_$free_map** if your device driver supports a 20-bit MULTIBUS controller or an PC AT compatible bus controller designed to run on the DN4000.

The input and output parameters are described as follows:

status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu2_$map Maps an I/O buffer.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

pbu2_$iova_t pbu2_$map(
     pbu_$unit_t        &unit,
     pbu_$buffer_t      buffer,
     linteger           &length,
     pbu2_$iova_t       &iova,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu2_$map(
     in  unit:          pbu_$unit_t;
     in  buffer:        univ pbu_$buffer_t;
     in  length:        linteger;
     in  iova:          pbu_$iova_t;
     out status:        status_$t); pbu2_$iova_t;
```

DESCRIPTION

You must use **pbu2_$map** if your device driver supports a 20-bit MULTIBUS controller or an PC AT compatible bus controller designed to run on the DN4000.

The **pbu2_$map** routine establishes the mapping between the buffer in processor address space and bus address space. Drivers must call this routine before using the buffer for I/O operations and only after they have called **pbu2_$allocate_map** and **pbu2_$wire**. (I/O buffers must be wired before being passed to **pbu2_$map**.) User-written interrupt routines can call **pbu2_$map**.

The address specified as a parameter to **pbu2_$map** need not be the address that the call **pbu2_$allocate_map** returned, but can reside on any page that corresponds to the allocated area of the I/O map. In this way, drivers can map several different buffers into different sections of the allocated I/O map area at the same time.

The input and output parameters are described as follows.

buffer	The buffer to be mapped, specified as an array of characters.
iova	A page-aligned bus address within the I/O map area allocated by pbu2_$allocate_map in pbu2_$iova_t format. This is a 4-byte integer, in C and Pascal.
length	The length in bytes of the buffer. This is a 4-byte integer, in C and Pascal.
returned_iova	The MULTIBUS address that marks the start of the buffer in bus address space in pbu2_$iova_t format. This is a 4-byte integer, in C and Pascal.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu2_$map_controller Maps 20-bit MULTIBUS, PC AT compatible, or VMEbus controller memory to processor address space.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

 char *pbu2_$map_controller(
      pbu_$unit_t       &unit,
      pbu2_$iova_t      &iova,
      linteger          &length,
      status_$t         *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

function pbu2_$map_controller(
     in  unit:          pbu_$unit_t;
     in  iova:          pbu2_$iova_t;
     in  length:        linteger;
     out status:        status_$t); univ_ptr;
```

DESCRIPTION

The **pbu2_$map_controller** routine maps 20-bit MULTIBUS, PC AT compatible, or VMEbus controller memory to processor address space. Device drivers can map only one area of controller memory per device at a time.

Note that memory mapped with **pbu2_$map_controller** *must* be unmapped with **pbu2_$unmap_controller**.

Refer to Chapter 7, Subsection 7.2.1 for information on referencing controller memory.

The input and output parameters are described as follows:

address	The virtual address of the first byte of the controller's mapped memory in univ_ptr format. For an equivalent of univ_ptr in C, refer to Appendix C, Subsection C.2.4.

iova	The bus address that marks the start of controller memory in pbu2_$iova_t format. This is a 4-byte integer, in C and Pascal. The address must lie on a page boundary.
length	The length in bytes of controller memory. This is a 4-byte integer, in C and Pascal.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

Possible errors associated with this routine include:

`PBU_$BAD_UNIT`

The specified unit number is invalid.

`PBU_$NOT_ACQUIRED`

The device has not been acquired.

`PBU_$ALREADY_MAPPED`

Controller memory has already been mapped.

Errors can also include those generated by **pbu2_$allocate_map**, the most common of which is that the requested memory is already allocated. If this error is generated, check the DMA devices in the configuration to see if they are using the desired bus addresses.

NAME

pbu2_$unmap Unmaps an I/O buffer.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>


void pbu2_$unmap(
     pbu_$unit_t        &unit,
     pbu_$buffer_t      buffer,
     linteger           &length,
     pbu2_$iova_t       &iova,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"


procedure pbu2_$unmap(
     in  unit:          pbu_$unit_t;
     in  buffer:        univ pbu_$buffer_t;
     in  length:        linteger;
     in  iova:          pbu2_$iova_t;
     out status:        status_$t);
```

DESCRIPTION

You must use **pbu2_$map** if your device driver supports a 20-bit MULTIBUS controller or a PC AT compatible bus controller designed to run on the DN4000.

The **pbu2_$unmap** routine unmaps the buffer from bus address space and invalidates the I/O map for the space occupied by the buffer. Device drivers are not, however, required to unmap previously mapped buffers; another call to **pbu2_$map** that specifies the same area of the I/O map effectively unmaps the previously mapped buffer. The **pbu2_$unmap** routine is used primarily to protect a buffer from erroneous references by a controller. The **pbu2_$unmap** routine can be called from the interrupt side.

The input and output parameters are described as follows:

buffer	The buffer to be unmapped. Specifies the buffer as an array of characters.
iova	The bus address that marks the start of the buffer in pbu2_$iova_t format. This address must be the address that **pbu2_$map** returned (the actual start of the buffer).
length	The length in bytes of the area to be unmapped. This is a 4-byte integer, in C and Pascal.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu2_$unmap_controller Unmaps a 20-bit MULTIBUS, PC AT compatible, or VMEbus controller's memory from processor address space.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu2_$unmap_controller(
     pbu_$unit_t        &unit,
     char               *&va,
     linteger           &length,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu2_$unmap_controller(
     in unit:           pbu_$unit_t;
     in va:             univ_ptr;
     in length:         linteger;
     out status:        status_$t);
```

DESCRIPTION

The **pbu2_$unmap_controller** routine unmaps from processor address space the controller memory mapped by **pbu2_$map_controller**. The whole mapped length must be unmapped.

The input and output parameters are described as follows:

address	The virtual address of the first byte of the controller's mapped memory in univ_ptr format. For an equivalent of univ_ptr in C, refer to Appendix C, Subsection C.2.4.
length	The length in bytes of controller memory. This is a C unsigned long integer or a 4-byte Pascal integer.
status	Completion status in status_$t format.

unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

Possible errors associated with this routine can include the following:

`PBU_$BAD_UNIT`

The specified unit number is invalid.

`PBU_$NOT_ACQUIRED`

The specified device has not been acquired.

`PBU_$NOT_MAPPED`

Controller memory has not been mapped.

NAME

pbu2_$unwire Unwires an I/O buffer.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu2_$unwire(
     pbu_$unit_t         &unit,
     pbu_$buffer_t       buffer,
     linteger            &length,
     boolean             &touch,
     status_$t           *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu2_$unwire(
     in  unit:           pbu_$unit_t;
     in  buffer:         univ pbu_$buffer_t;
     in  length:         linteger;
     in  touch:          boolean;
     out status:         status_$t);
```

DESCRIPTION

The **pbu2_$unwire** routine makes a buffer previously wired into processor memory with **pbu2_$wire** and **pbu_$wire_special** available for MMU paging operations.

NOTE: Buffers that are part of a driver's interrupt side must never be unwired.

The input and output parameters are described as follows:

buffer	The buffer to be unwired, specified as a universal array of characters.
length	The length in bytes of the buffer. This is a 4-byte integer, in C and Pascal.
modify_flag	A Boolean value that indicates whether the buffer pages being unwired should be marked as modified by an input I/O operation. This flag is needed because DMA does not set the page's modify bit in Memory Management Unit (MMU) tables. For more information, refer to Chapter 7, Subsection 7.1.3.2. For information about using Boolean values in C, refer to Appendix C, Subsection C.2.3.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2-byte Pascal integer.

NAME

pbu2_$wire Wires an I/O buffer.

SYNOPSIS (C)

```
#include <apollo/base.h>
#include <apollo/pbu.h>

void pbu2_$wire(
     pbu_$unit_t        &unit,
     pbu_$buffer_t      buffer,
     linteger           &length,
     status_$t          *status)
```

SYNOPSIS (Pascal)

```
%include "/sys/ins/base.ins.pas"
%include "/sys/ins/pbu.ins.pas"

procedure pbu2_$wire(
     in  unit:          pbu_$unit_t;
     in  buffer:        univ pbu_$buffer_t;
     in  length:        linteger;
     out status:        status_$t);
```

DESCRIPTION

The **pbu2_$wire** routine makes the buffer's pages permanently resident in processor memory in preparation for an I/O operation. Drivers must wire I/O buffers before mapping them with **pbu2_$map**.

Drivers need not wire interrupt routine buffers with **pbu2_$wire** because **pbu_$acquire** automatically wires the data sections of the driver's interrupt routine(s) when the device is acquired. Refer to Chapter 7, Subsection 7.1.1.

The **pbu2_$wire** routine returns an error if any page of the specified buffer is already wired.

The input and output parameters are described as follows:

buffer	The buffer to be wired, specified as a universal array of characters.
length	The length in bytes of the buffer. This is a 4–byte integer, in C or Pascal.
status	Completion status in status_$t format.
unit	The device unit number in pbu_$unit_t format. This is a C unsigned short integer or a 2–byte Pascal integer.

B.3 Error Messages

This section lists the possible error messages that can be returned by GPIO calls. If a message is returned by only one call, or set of calls, that call is given in parentheses.

PBU_$ALL_IN_USE
: All GPIO units are in use.

PBU_$ALREADY_ACQUIRED
: Unit already acquired.

PBU_$ALREADY_ALLOCATED
: I/O map already allocated (**pbu[2]_$allocate_map**).

PBU_$ALREADY_MAPPED
: Controller already mapped (**pbu[2]_$map_controller**).

PBU_$ALREADY_WIRED
: Page already wired (**pbu[2]_$wire**).

PBU_$BAD_BUFFER
: Bad buffer address.

PBU_$BAD_CSR_ADDRESS
: CSR address not on CSR page (**pbu_$read/write_csr**).

PBU_$BAD_CSR_ADDR_IN_DDF
: Invalid CSR page address.

PBU_$BAD_DDF_TYPE
: Not a DDF (**pbu_$acquire**).

PBU_$BAD_DDF_VERSION
: DDF is wrong version (**pbu_$acquire**).

PBU_$BAD_DIRECTION
: Bad DMA direction specified (**pbu_$dma_start**).

PBU_$BAD_IOVA
: Bad iova (**pbu[2]_$map**).

PBU_$BAD_LEN
: Length parameter too large or too small.

PBU_$BAD_PARM
: Bad parameter.

PBU_$BAD_UNIT

Bad unit number specified in call.

PBU_$BAD_UNIT_IN_DDF

Bad unit number in DDF (**pbu_$acquire**).

PBU_$BUFFER_TOO_BIG

Buffer too big (**pbu[2]_$map**).

PBU_$BUS_TIMEOUT

Read/write CSR caused bus time-out (**pbu_$read/write_csr**).

PBU_$CHANNEL_IN_USE

Requested DMA channel in use (**pbu_$dma_start**).

PBU_$CHANNEL_NOT_IN_USE

Requested DMA channel not in use (**pbu_$dma_stop**).

PBU_$CLEANUP_ROUTINE_MISSING

Cleanup routine not in driver (**pbu_$acquire**).

PBU_$CSR_PAGE_IN_USE

CSR page in use.

PBU_$DDF_TOO_BIG

DDF greater than 1 KB in length.

PBU_$DEVICE_NOT_MAPPED

Controller not mapped (**pbu[2]_$unmap_controller**).

PBU_$DEVICE_NOT_SHARED

pbu_$mem_ptr called for nonshared memory-mapped controller (**pbu_$mem_ptr**).

PBU_$DEVICE_TIMEOUT

MULTIBUS device got bus time-out.

PBU_$DMA_NOT_EOR

DMA channel not at end of range (**pbu_$dma_stop**).

PBU_$EC_NOT_ALLOCATED

Eventcount not allocated to this unit.

PBU_$ILLEGAL_CHANNEL

Illegal DMA channel number (**pbu_$dma_start**).

PBU_$ILLEGAL_TRAP

Trap 6 from level 0.

PBU_$ILLEGAL_TRAP_CODE
: Bad trap 6 code.

PBU_$ILLEGAL_USP
: Invalid USP on trap 6.

PBU_$INIT_ROUTINE_MISSING
: Initialization routine not in driver library (**pbu_$acquire**).

PBU_$INTERRUPT_ROUTINE_MISSING
: Interrupt routine not in driver library (**pbu_$acquire**).

PBU_$INT_LIB_NOT_FOUND
: Interrupt library name (from DDF) not found (**pbu_$acquire**).

PBU_$INT_LIB_TOO_BIG
: Interrupt library larger than 32 KB (**pbu_$acquire**).

PBU_$INT_VECTOR_IN_USE
: VMEbus interrupt vector in use (**pbu_$acquire**).

PBU_$LIB_NOT_FOUND
: Device library not found (**pbu_$acquire**).

PBU_$MAP_IN_USE
: Requested I/O map in use (**pbu[2]_$allocate_map**).

PBU_$NO_MORE_ECS
: No more eventcounts available (**pbu_$allocate_ec**).

PBU_$NO_ROOM
: No room in I/O map (**pbu[2]_$allocate_map**).

PBU_$NOT_ACQUIRED
: Unit not acquired.

PBU_$NOT_ALLOCATED
: I/O map not allocated (**pbu[2]_$free_map**).

PBU_$NOT_MAPPED
: Buffer not mapped (**pbu[2]_$unmap**).

PBU_$NOT_VME
: Operation valid for VMEbus device only (**pbu_$wire_special**).

PBU_$NOT_WIRED
: Page not wired (**pbu[2]_$unwire**).

PBU_$OS_PUBLIC_DEVICE

Unit is publicly owned; can be released by any process.

PBU_$PA_LIST_OVERFLOW

List of physical addresses too small (**pbu_$wire_special**).

PBU_$PAGE_NOT_WIRED

Buffer page not wired (**pbu[2]_$map**).

PBU_$PBU_NOT_PRESENT

MULTIBUS not present in system.

PBU_$PPN_LIST_OFLO

Too many PBU Manager pages wired (crash system).

PBU_$PROTECTION_VIOLATION

Bad argument on trap 6.

PBU_$TOO_MANY_WIRED_PAGES

Too many wired pages.

PBU_$UNEXPECTED_INTERRUPT

Unexpected interrupt from some device.

PBU_$UNIT_IN_USE

Requested unit in use.

PBU_$UNIT_IS_GLOBAL

Unit already in use as a global device.

PBU_$UNSUPPORTED_FUNCTION

Unsupported function requested.

PBU_$WRONG_LIBRARY

Out of date **pbulib**.

STATUS_$OK

Successful completion.

Appendix C

Programming Information

This appendix provides tips, warnings, and rules for Pascal and C programmers who are developing device drivers on our operating system.

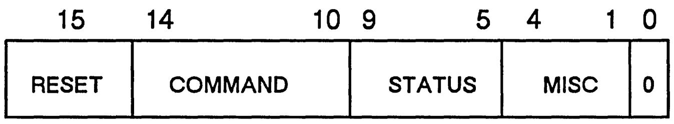
C.1 CSR Page

In general, use data types of char for C, or integer and char for Pascal, when declaring a CSR page because the compiler word–aligns records (or C structs) and arrays even if they appear inside a packed record.

If you want to declare a register as a C or Pascal enumerated type, C struct, or Pascal set, follow these steps:

1. Declare the register as char for C, or type char and integer for Pascal, to ensure proper byte alignment.
2. Copy the register into local storage that contains a union in C, or a variant in Pascal, for the character or integer type and a variant for the structure.
3. Operate on the copy of the register in local storage.
4. Write the modified version back to the actual register.

Suppose that a CSR has the following internal representation:



The definition of the driver's private copy of this register in C would be as follows:

```
typedef union {
        struct {
            unsigned int reset  :1;
            unsigned int cmd    :5;
            unsigned int status :5;
            unsigned int misc   :4;
            unsigned int mbz    :1;
        } fields;
        short all;
} csr_t #attribute[device];
```

The following statement illustrates addressing in C:

```
csr_t *csr_ptr;
```

An example of its use:

```
csr_ptr->fields.reset = 1;       /* reset the device */
```

The following sequence of code illustrates how to define the driver's private copy of this register as a record in Pascal:

```
type csr_t = [device] packed record case integer of
        0: (reset:  boolean;
            cmd:    0..31;
            status: 0..31;
            misc:   0..15;
            mbz:    0..0);
        1: (all:    integer);
        end;   { csr_t }
```

The following statement illustrates addressing in Pascal:

```
var csr_ptr: ^csr_t;
```

An example of its use:

```
csr_ptr^.reset := true;          { reset the device }
```

Declare 8-bit registers within the CSR page as char types. The char type ensures that the registers will be byte aligned. If you want to perform arithmetic or bit-manipulation operations on the register, use the ord function, which will return the integer value of the char data type.

Do not declare 8-bit registers within the CSR page as Pascal sets or Pascal or C enumerated types. If an 8-bit register within the CSR page is declared as a set or an enumerated type (for example, 0...255), the compiler generates code that copies the register to a temporary variable and passes the temporary variable to the routine. This sequence touches the CSR and may cause a bus time-out if the controller is not responding.

C.2 Programming in C

This section contains several additional hints for C programmers. Before you write a device driver in C, refer to the *Domain C Language Reference* manual and the *Domain C Library (CLIB) Reference* manual for complete information about our version of the C language. Use the suggestions in Subsections C.2.1 through C.2.5 to supplement the information in those manuals. An example of a driver coded in C appears in Appendix E.

C.2.1 Insert Files

The GPIO insert file for C programmers is **/usr/include/apollo/pbu.h**. Include this file in your C modules by using the #include compiler control line, as described in the *Domain C Language Reference*. You should also include the standard C include files listed in the C manual.

C.2.2 Type int

In C, type int is four bytes and short int is two bytes. In Pascal, type integer is two bytes. The *Domain C Language Reference* manual contains a table of corresponding data types in the various languages.

C.2.3 Boolean Values

Although C does not support a Boolean type, certain GPIO routines take Boolean arguments in which the routine expects a value of "true" or "false". As arguments for those routines, you must use the definitions of "true" and "false" available in the C include file **/usr/include/apollo/base.h**. Remember to include this file in your device driver program, as described in the *Domain C Language Reference* manual.

In C, any nonzero value is defined as "true"; in Pascal, only a value of FF (hex) is "true." For "true," the GPIO routines expect the Pascal value. "True" is defined in the include file as FF (hex). If you don't use the include file definitions, the GPIO routines may not recognize as "true" the value the C compiler gives as "true."

C.2.4 Universal Pointer Type

Domain Pascal includes a predeclared data type called univ_ptr, which is a universal pointer. To create an equivalent to this type in C, use a pointer to char as follows:

```
char *ptr;
```

C.2.5 Defining Globals

For drivers written in C, the following example from **bm_example_c** shows how to ensure that a global needing to be wired gets defined in your interrupt routine.

The **bm_pvt.h** file includes this line:

```
extern bm_$bmcb_t bmcb;        /* declare bmcb as external, every module
                                  that includes this file knows about
                                  bmcb */
```

The **bm_int_lib.c** file includes this line:

```
bm_$cb_t bmcb;                 /* here we provide bmcb */
```

In Pascal, globals reside in the data section of the module in which they are defined. Thus, globals that are referenced in the interrupt side of a Pascal driver must be declared there with the DEFINE clause.

For drivers written in Pascal, the following example from **bm_example** shows how to ensure that a global needing to be wired gets defined in your interrupt routine.

The **bm.pvt.pas** file includes this line:

```
VAR bmcb: EXTERN bm_$bmcb_t;   { declare bmcb as external, every module
                                 that includes this declaration knows
                                 about bmcb }
```

The **bm_int_lib.pas** file includes this line:

```
DEFINE bmcb;                   { here we provide bmcb }
```

C.3 Considerations for Compiler Optimization

In SR8.0 and later software revisions, the compiler provides optimization (the **-opt** option) by default. For correct optimization in device driver modules, you must identify to the compiler the variables that are actually mapped into device registers. The compilers at SR8.0 and later software revisions provide attributes you may use; this section discusses them. For more specific information about compiler switches, refer to the Software Release Document shipped with the compiler software, or to the online version of the compiler release document. See also the *Language Reference* manual for each compiler product.

In editions of this manual previous to SR8.0, we suggested using dummy labels to thwart compiler optimizations; however, in SR8.0 and later software releases, this technique no longer suffices. Instead, you use the DEVICE attribute to inform the compiler not to perform certain optimizations in some situations.

The DEVICE attribute is necessary because certain sequences of references to device registers may not generate the desired code. Programs commonly use a register for commands on output and status on input. The example that follows shows the code generated by the compiler without optimization (**-nopt** option used).

```
csr := read_status_command;
           MOVEQ.B    #01,D1
           MOVE.B     D1,CSR(DB)
status := csr;
           MOVE.B     CSR(DB),STATUS(DB)
```

Using ordinary optimization (without using the DEVICE attribute in the device register type declaration), the compiler remembers the value in D1 and never makes a second reference to the register.

```
csr := read_status_command;
           MOVEQ.B    #01,D1
           MOVE.B     D1,CSR(DB)
status := csr;
           MOVE.B     D1,STATUS(DB)
```

The code generated is incorrect because D1, not CSR(DB), is written to STATUS(DB), and the value in CSR(DB), depending on the action of the controller, may not be the same as that in D1. The DEVICE attribute informs the compiler that the variable is part of an I/O controller and requires careful handling. Specifically, it ensures that the compiler does not omit assignments or use instructions that involve "hidden" read cycles. In modules that directly reference device registers mapped into the MULTIBUS address space, use the DEVICE attribute in the declaration of the device register data structure. The compiler then will always generate a reference to the register on both reads and writes.

For example, note the following segment of a type declaration. The example is from the module **ether.pvt.pas,** in the directory **/domain_examples/gpio_examples/threecom_example**. (Note that declarations for **ether_mecsr_t, ether_xmit_buf_t,** and **ether_rcv_buf_t** appear earlier in **ether.pvt.pas,** and the declaration for **ether_$adr_t** appears in **ether.ins.pas.**)

```
ether_memory_t = [device] packed record case integer of
0:         (csr:             ether_mecsr_t;     { control &
                                                  status
                                                  registers }
           retran_timr:      integer16;         { Retransmit
                                                  timer }
           pad_to_adr_rom:   array [1..16#3FC] of char;
           adr_rom:          ether_$adr_t       { + 400 }
           pad_to_adr_ram:   array [1..16#1FA] of char;
           adr_ram:          ether_$adr_t;      { + 600 }
           pad_to_tbuf:      array [1..16#1FA] of char;
           tbuf:             ether_xmit_buf_t;  { + 800 }
           rbuf:             array [0..1] of
                             ether_rcv_buf_t);  { + 1000, +1800}
1:         (bytes:array [0..16#1FFF] of char);
end;
```

The example that follows shows the same segment written in C. In this example, the pad arrays are called pad_1, pad_2, etc., instead of the names used in the Pascal example, but they perform the same functions as in the Pascal example. The C example also includes type declarations so that the segment will compile on its own.

```
typedef shortether_mecsr_t;
typedef charether_$adr_t[6];
typedef charether_xmit_buf_t[0x800];
typedef charether_rcv_buf_t[0x800];

typedef union {
      struct {
           ether_mecsr_t     csr;               /* Control
                                                   status
                                                   registers */
           short             retran_timer;      /* Retransmit
                                                   timer */
           char              pad_1[0x3fc];
           ether_$adr_t      adr_rom;           /* + 400 */
           char              pad_2[0x1fa];
           ether_$adr_t      adr_ram;           /* + 600 */
           char              pad_3[0x1fa];
           ether_xmit_buf_t  tbuf;              /* + 800 */
           ether_rcv_buf_t   rbuf[2];           /* + 1000,
                                                   + 1800 */
      } fields;
      charbytes[0x1fff];
} ether_memory_t #attribute[device];
```

Use of the DEVICE attribute guarantees that

- The compiler does not merge adjacent register references into larger references. For example, two MOVE.W instructions do not become a MOVE.L.
- The compiler does not generate gratuitous read–modify–write references for DEVICE registers.
- The compiler does not generate CLR or ST instructions when it writes a 0 or –1 to a location defined as having the DEVICE attribute.

Another attribute, the VOLATILE attribute, informs the compiler that memory contents may change without notice. Any register declared with the DEVICE attribute receives the VOLATILE attribute as well.

Appendix D

Performance Information

This appendix describes hardware and software performance during I/O operations.

D.1 DMA Bandwidth

The rate at which a controller on the bus moves data to or from system memory depends upon how long it has control of the bus, the bus acquisition time, and the number of words transferred per bus acquisition. In turn, bus acquisition time depends upon the current activity of other devices using the bus, such as the CPU, ring/disk board, and so on. Bus acquisition time can range from 100 nanoseconds (minimum) to 2 microseconds (typical) to 1 millisecond (worst case; usually during a ring or disk transfer). Once the controller acquires the bus, it can transfer data over the bus at a rate of 1 microsecond per 16–bit word.

DMA controllers should not cause excessive DMA overruns. A DMA overrun occurs when a controller cannot transfer data to the processor as fast as it is receiving the data and so loses data. If a controller does cause an overrun, it must abort the rest of the transfer so that at least one DMA controller can successfully complete a transfer when an overrun occurs.

As a general rule, a controller should not require a long–term average of more than 20 percent of the bus bandwidth. No single transfer should take longer than 10 microseconds. This limit prevents a controller from unduly interfering with system operation.

D.2 Interrupt Processing Overhead

The amount of CPU time required to process a device interrupt depends upon several considerations:

- Basic system overhead
- The amount of processing the user–written interrupt routine performs
- The directives (interrupt enable or eventcount advance) that the user–written interrupt routine sends to the System Interrupt Handler through the *return_flags* parameter

Table D–1 lists the CPU times in the various stages of interrupt processing. All times are in microseconds. Observed times may vary up to 10 percent, depending on the processor, system activity, hardware caching, and so on.

Table D–1. CPU Times During Interrupt Processing

Interrupt Activity	**CPU Time (μsec)**
Interrupt request by device to first instruction of interrupt routine	125
Interrupt routine	Variable
Enabling the device specifying **pbu_$interrupt _enable** on return	10
Exit to interrupted process with no advance of the device's eventcount	110
Exit to interrupted process with advance, but no one waiting on eventcount	200
Exit to interrupted process with advance, with someone waiting on eventcount	265

Using Table D–1, we can determine that, for example, the total system overhead for an interrupt routine that awakens a waiting process is 125 + 265 = 390 microseconds.

If the *only* action of the interrupt routine is to advance the eventcount, the routine itself can be eliminated. If no user interrupt routine is specified for the device, the system interrupt handler automatically advances the device's eventcount. This requires a total of 260 microseconds if no one is waiting on the eventcount, 325 microseconds if someone is waiting.

D.3 Setting Up DMA Buffers

When designing a device driver for a DMA controller, you have a choice of how to set up the DMA buffers. Assume that the driver has a routine called WRITE, which an application program calls with the address and length of a buffer; WRITE must then perform the appropriate operations to send the data to a device.

The first approach looks like this:

```
Driver initialization routine:
         Allocate iomap for largest possible buffer.
```

```
WRITE routine:
         Wire the buffer.    (pbu2_$wire)
         Map the buffer.     (pbu2_$map)
         Start the I/O and wait for completion.
         Unwire the buffer. (pbu2_$unwire)
         Return to caller.
```

On a DSP80, the total time (ignoring the I/O time) for a buffer n pages in length is as follows:

pbu2_$wire:	0.302 (SVC overhead) + 0.605n
pbu2_$map:	0.295 (SVC overhead) + 0.175n
pbu2_$unwire:	0.312 (SVC overhead) + 0.311n
	==============================
	0.909 (SVC overhead) + 1.091n milliseconds

In the second approach, there is a permanently wired and mapped buffer area, and application data is copied into this buffer for each write operation.

```
Driver initialization routine:
         Allocate iomap for largest possible buffer.
         Create (ms_$crmapl*) and wire the buffer.
         Map the buffer.
```

```
WRITE routine:
         Copy user's data into the buffer.
         Start the I/O and wait for completion.
```

The time for this approach is

page copy:	0.000 (SVC overhead) + 0.913n milliseconds

*Refer to the *Domain/OS System Calls Reference* manual.

The point is that wiring and unwiring buffers are relatively expensive operations, and you should always consider the option of copying data into a permanently allocated and mapped buffer.

Also keep in mind that the stated times do not include the overhead of any page faults required to get the buffer into memory. Such overhead, however, would be the same for both approaches. If data is being collected from several noncontiguous buffers for a single DMA operation, copying saves even more time because **pbu2_$wire**, **pbu2_$map**, and **pbu2_$unwire** will have to be called for each separate buffer. For example, mapping a 5-page buffer with one call to **pbu2_$map** takes 1.561 milliseconds; mapping five 1-page buffers takes 2.765 milliseconds. You will notice that **pbu2_$unmap** is not used (refer to the description of **pbu_$unmap** and **pbu2_$unmap** in Appendix B). If an application requires very large buffers (for example, 512 KB), overall performance may suffer if a buffer is permanently wired. In such cases experimentation is required to determine the best approach.

D.4 Timing Information

Table D-2 lists the times of certain GPIO operations for the DN560, DSP80, and DSP160 as of SR9.5, and DN660, DN3000, DN4000, and DN5xx-T as of SR10. Observed times may vary up to 5 percent, depending on other activity in the system. The times for **pbu_$wire** do not include any page faults; the pages being wired were all resident in physical memory. All times are in milliseconds.

NOTE: Using **pbu_$read_csr** or **pbu_$write_csr** to read or write to a CSR takes around 100 microseconds, depending on the node model. Doing the read/write directly is typically 1-2 instructions or 3-5 microseconds, depending on the node model.

Table D-2. Timing for DN560, DSP80, DSP160, DN660, DN5xx-T, DN3000, and DN4000 Workstations

Model	**Operation**	**Times (msec)**
DN560 (SR9.5)	page copy	0.000 (SVC overhead) + 0.255/page
	pbu2_$wire	0.107 (SVC overhead) + 0.216/page
	pbu2_$unwire	0.112 (SVC overhead) + 0.100/page
	pbu2_$map	0.101 (SVC overhead) + 0.056/page
	pbu2_$unmap	0.146 (SVC overhead) + 0.004/page
DSP80 (SR9.5)	page copy	0.000 (SVC overhead) + 0.913/page
	pbu2_$wire	0.302 (SVC overhead) + 0.605/page
	pbu2_$unwire	0.312 (SVC overhead) + 0.311/page
	pbu2_$map	0.295 (SVC overhead) + 0.175/page
	pbu2_$unmap	0.443 (SVC overhead) + 0.009/page
DSP160 (SR9.5)	page copy	0.000 (SVC overhead) + 0.849/page
	pbu2_$wire	0.159 (SVC overhead) + 0.252/page
	pbu2_$unwire	0.239 (SVC overhead) + 0.166/page
	pbu2_$map	0.116 (SVC overhead) + 0.098/page
	pbu2_$unmap	0.230 (SVC overhead) + 0.004/page
DN660 (SR10)	page copy	0.000 (SVC overhead) + 1.177/page
	pbu2_$wire	0.412 (SVC overhead) + 0.749/page
	pbu2_$unwire	0.313 (SVC overhead) + 0.373/page
	pbu2_$wire_map	0.447 (SVC overhead) + 0.722/page
	pbu2_$map	0.140 (SVC overhead) + 0.091/page
	pbu2_$unmap	0.220 (SVC overhead) + 0.004/page
DN5xx-T (SR10)	page copy	0.000 (SVC overhead) + 0.323/page
	pbu2_$wire	0.095 (SVC overhead) + 0.316/page
	pbu2_$unwire	0.111 (SVC overhead) + 0.295/page
	pbu2_$wire_map	0.124 (SVC overhead) + 0.322/page
	pbu2_$map	0.104 (SVC overhead) + 0.338/page
	pbu2_$unmap	0.431 (SVC overhead) + 0.004/page
DN3000 (SR10)	page copy	0.000 (SVC overhead) + 0.299/page
	pbu2_$wire	0.097 (SVC overhead) + 0.409/page
	pbu2_$unwire	0.106 (SVC overhead) + 0.441/page
	pbu2_$map	Unsupported call
	pbu2_$unmap	Unsupported call
DN4000 (SR10)	page copy	0.000 (SVC overhead) + 0.246/page
	pbu2_$wire	0.149 (SVC overhead) + 0.582/page
	pbu2_$unwire	0.067 (SVC overhead) + 0.321/page
	pbu2_$wire_map	0.174 (SVC overhead) + 0.594/page
	pbu2_$map	0.048 (SVC overhead) + 0.090/page
	pbu2_$unmap	0.152 (SVC overhead) + 0.002/page

Appendix E

Sample Driver in C

This appendix contains the files that make up the online device driver in the subdirectory **/domain_examples/gpio_examples/bm_example_c**. The "makefile" script in Section E.4 organizes the files into call and interrupt libraries at bind time.

Both the functional parts and the operation of this driver are fully described in Chapter 4, Subsections 4.4.2 and 4.4.3, and Figure 4–2. For additional information about the driver and the hypothetical bulk–memory controller it supports, refer to Section E.4. For information about writing device drivers in C, refer to Appendix C, Section C.2.

> **NOTE:** Unlike Pascal, the C programming language is case sensitive; therefore, all system procedure names (such as GPIO routines) must be lowercase, which is consistent with their appearance in the system insert files. However, any global names in C that are accessed by GPIO routines are case–sensitive.

The driver consists of three files (plus the makefile):

- C insert file: **bm.h**
- Call–side library file: **bm_lib.c**
- Interrupt side library file: **bm_int_lib.c**

E.1 bm.h

The bulk memory unit is a MULTIBUS controller at address 0x400 in MULTIBUS address space. It has 8-bit command and status registers at addresses 0x400 and 0x401, a 32-bit bulk memory address register at 402, a 16-bit count register at 406, and a 16-bit MULTIBUS (iova) register at 408. The controller interrupts at level 2. The device supports three operations: read from bulk memory, write to bulk memory, and wait for transfer complete. Up to a 1 MB can be transferred with one call, but since the MULTIBUS can transfer only up to 64 KB in one I/O operation, the interrupt side of the driver (this routine) is given the job of blocking large transfers into portions of size **bm_$block_len**. Note that **bm_$block_len** is not the maximum possible, which is 64 KB. The reason for not allowing 64-KB transfers is that it would require taking over the entire I/O map. Therefore, if another MULTIBUS device is using even a single page of the I/O map, our call to **pbu_$allocate_map** would fail.

This file contains the data structures and constants for the bulk memory device.

```
/*  Error codes from bm manager calls. (We've arbitrarily picked a
 *  subsystem code of OF.)
 */

#define bm_$no_controller        0x0F000001 /* controller not present */
#define bm_$not_init             0x0F000002 /* controller not
                                               initialized */
#define bm_$busy                 0x0F000003 /* controller is busy */
#define bm_$not_ready            0x0F000004 /* unit not ready */
#define bm_$bad_address          0x0F000005 /* buffer beyond protection
                                               boundary */
#define bm_$bad_length           0x0F000006 /* bad buffer length */
#define bm_$bad_bm_address       0x0F000007 /* bad bm address */
#define bm_$transfer_not_started 0x0F000008 /* tried to wait before read
                                               or write */
#define bm_$timeout              0x0F000009 /* timeout during wait */
#define bm_$quit_during_wait     0x0F00000A /* quit during wait */
#define bm_$io_error             0x0F00000B /* i/o error during
                                               transfer */

#define bm_$max_address          2147483647 /* maximum bm address  =
                                               2**31 - 1 */
#define bm_$block_len            32768      /* maximum transfer per i/o
                                               operation = 32K */
#define bm_$max_len              131072     /* maximum amount to
                                               transfer per call =
                                               128K */
                                            /* N.B.: MUST be multiple of
                                               bm_$block_len (see
                                               bm_$int)! */
```

```
typedef pinteger  bm_$buf_len_t;          /* bm buffer dimension */
typedef int       bm_$buf_t[bm_$max_len];
typedef bm_$buf_t *bm_$buf_ptr_t;
typedef long      bm_$bm_address_t;       /* address of block in bulk
                                             memory */

/**************** ALL PRIVATE DEFINITIONS FOLLOW *****************/

/* possible command for csr command register */

#define BM_INIT_CMD     (unsigned char)0x00
#define BM_READ_CMD     (unsigned char)0x01
#define BM_WRITE_CMD    (unsigned char)0x02

#define bm_$status_ok   (unsigned char)0x0c     /* normal completion
                                                   status */
#define bm_$sio_error   (unsigned char)0xff     /* interrupt routine
                                                   got error from
                                                   bm_$sio (start I/O
                                                   routine) */

/*
 * Define the bulk memory controller's csr page. (Note: when defining
 * the contents of a csr page, watch out for the compiler's rules about
 * packing records. In particular, avoid using records inside the csr
 * page record, since embedded records will be word-aligned, even in a
 * packed record. For example, we might have defined the status register
 * to be bm_$status_t (see below), but then the compiler would have
 * aligned it at offset 2 in the page even though bm_$status_t
 * is only 8 bits wide.)
 */

/* use device attribute */

typedef struct {
        unsigned char       command;    /* 00 one byte command register
                                           at offset 0 */
        unsigned char       status;     /* 01 one byte status
                                           register */
        short               iova;       /* 02 io virtual address to use
                                           for transfer */
        short               count;      /* 04 number of bytes to
                                           transfer */
        bm_$bm_address_t    bm_address; /* 06 bulk memory address to
                                           read/write */
} bm_$csr_page_t #attribute[device];

typedef union {
        bm_$csr_page_t      *c;
        pbu_$csr_page_ptr_t  p;
} bm_$csr_page_ptr_t;
```

```
/*
 * Define the bulk memory control block (bmcb). This area is used for
 * communications between the call and interrupt sides of the bm driver.
 * Since it is referenced by the interrupt handler, it must be part of
 * the interrupt library
 */

typedef union {
        struct {
                unsigned int    init: 1    /* set to true when
                                              controller initialized */
                unsigned int    buffer_wired : 1;  /* set when a buffer
                                                      is wired */
                unsigned int    busy : 1;   /* set when an operation is
                                               in progress */
                unsigned int    done : 1;   /* set by interrupt routine
                                               when transfer
                                               completes */
                unsigned int    pad : 4;    /* fill out to byte ? */
        } b;
        char    all;
} bm_$flags_t;

/* status register definition */

typedef union {
        struct {
                unsigned int    attention: 1;          /* 1 => change in
                                                          controller
                                                          status */
                unsigned int    status_modifier : 1;   /* 1 => current
                                                          status
                                                          unavailable */
                unsigned int    control_unit_end : 1;  /* 1 => busy
                                                          condition
                                                          cleared */
                unsigned int    busy : 1;              /* 1 => controller
                                                          currently
                                                          busy */
                unsigned int    channel_end : 1;       /* 1 => end of
                                                          operation */
                unsigned int    device_end : 1;        /* 1 => end of
                                                          operation */
                unsigned int    unit_check : 1;        /* 1 => parity
                                                          error in bm */
                unsigned int    unit_exception : 1;    /* 1 => illegal bm
                                                          address */
        } b;
        unsigned char    all;
} bm_$status_t;
```

```
typedef struct {                                        /* define
                                                           communications
                                                           area */
        pbu_$unit_t             pbu_unit_number;        /* number of this pbu
                                                           device */
        bm_$flags_t             flags;
        char                    pad;                    /* a byte of
                                                           padding */
        pbu_$ddf_ptr_t          ddf_ptr;                /* pointer to mapped
                                                           ddf */
        bm_$csr_page_ptr_t      csr_ptr;                /* pointer to mapped
                                                           csr page */
        pbu_$iova_t             bm_iova;                /* start of our area
                                                           of i/o address
                                                           space */
        bm_$buf_ptr_t           bufaddr;                /* address of start of
                                                           buffer */
        bm_$buf_len_t           buflen;                 /* total length of
                                                           buffer */
        bm_$bm_address_t        bm_address;             /* address of start of
                                                           bm area */
        unsigned char           command;                /* current command
                                                           (read or write) */
        bm_$buf_len_t           rem_len;                /* length remaining to
                                                           read or write */
        bm_$status_t            status;                 /* status from last
                                                           interrupt */
        status_$t               sio_status;             /* status from bm_$sio
                                                           (start I/O routine)
                                                           called from int
                                                           side */
        bm_$buf_ptr_t           io_addr;                /* address of last i/o
                                                           transfer */
        bm_$buf_len_t           io_len;                 /* length of last i/o
                                                           transfer */
        unsigned char           init_cmd;               /* initialization
                                                           command (see
                                                           bm_$init!) */
        unsigned char           read_cmd;               /* read command */
        unsigned char           write_cmd;              /* write command */
} bm_$bmcb_t;

extern bm_$bmcb_t bmcb;                                 /* main control
                                                           block */
```

```
/* Define the library entrypoints. (Note: strictly speaking, the
 * user-visible entry points should be defined in a separate include
 * file so that the internal routines and data structures are not seen
 * by application programs.)
 */

extern void bm_$cleanup(
    pbu_$unit_t             *unit,          /* pbu unit number */
    char                    *force,
    status_$t               *status
);

extern void bm_$init(
    pbu_$unit_t             *unit,           /* pbu unit number */
    pbu_$ddf_ptr_t          *ddf_ptr,
    pbu_$csr_page_ptr_t     *csr_ptr,
    status_$t               *status
);

extern pbu_$interrupt_return_t bm_$int(
    pbu_$unit_t *unit
);

extern void bm_$read(
    bm_$buf_t               buffer,
    bm_$buf_len_t           buflen,
    bm_$bm_address_t        bm_addr,
    status_$t               *s
);

extern void bm_$write(
    bm_$buf_t               buffer,
    bm_$buf_len_t           buflen,
    bm_$bm_address_t        bm_addr,
    status_$t               *s
);

extern void bm_$wait(
    short                   timeout,        /* timeout value */
    bm_$status_t            *bm_status,     /* controller status */
    bm_$buf_len_t           *rem_len,       /* residual count */
    status_$t               *status
);

extern void bm_$sio(
    status_$t       *status
);
```

E.2 bm_lib.c

The **bm_lib.c** file consists of the call-side routines that perform initialization (**bm_$init**), cleanup (**bm_$cleanup**), command processing (**bm_$read**, **bm_$write**, and **bm_command**), and wait for interrupt (**bm_$wait**).

```
/*  This module is the device driver library for a fictitious pbu device
 * -- a bulk memory (BM) unit. The intent of the driver is to show the
 * general structure of a user-space device driver and to demonstrate
 * the use of the pbu manager routines.

 * The bulk memory unit is a pbu device whose controller is at address
 * 400 (hex) in the pbu address space. It has a 8-bit command and status
 * registers at addresses 400 and 401, a 32-bit bulk memory address
 * register at 402, a 16-bit count register at 406, and a 16-bit i/o
 * virtual address (iova) register at 408. The controller interrupts at
 * level 2.

 * The controller is initialized by writing 16#00 to the command
 * register.  Read and write operations are performed by loading the
 * address, count, and iova registers the then writing a 16#01 (read) or
 * 16#02 (write) to the command register. Status is obtained by reading
 * the status register.

 * The bm manager (this module) supports three operations -- read from
 * bulk memory, write to bulk memory, and wait for transfer complete. Up
 * to a 128K can be transferred with one call, but since the pbu cannot
 * transfer 128K in one i/o operation, the interrupt side of the driver
 * (see bm_int_lib.pas) is given the job of blocking large transfers
 * into chunks of size bm_$block_len.  (Note that bm_$block_len is not
 * the maximum possible, which is 64K. The reason for not allowing 64K
 * transfers is that it would require we take over the entire iomap.
 * Therefore if another pbu device is using even a single page of the
 * iomap, our call to pbu_$allocate_map would fail.)

 * A typical invocation of the bm library might appear as follows:
 */

    char          data_buffer[buf_size];
    status_$t     status;
    bm_$status_t  bm_example;
    long          bytes_left;

    bm_$write(data_buffer, 1024*10, 0, &status);  write 10 pages to bm
                                                  addr 0
    if (status.all != 0) {
        error_$print(status);                     display error code
        process_error();
    }
    bm_$wait(1, &bm_status, &bytes_left, &status); wait 1 second for
                                                    completion
    if (status.all != 0) {
        error_$print(status);                      display error code
        if (status.all == bm_$io_error)
            display_status_byte();
        process_error();
    }
    */

#nolist
#include <apollo/base.h>
#include <apollo/error.h>
#include <apollo/pfm.h>
#include "/latest/sr10/gpio/usr/include/apollo/pbu.h"

#include "bm.h"
#list

/* unwire_buffer -- internal routine to unwire a buffer */

static void
unwire_buffer(void)
{
        status_$t               st;

        if (!bmcb.flags.b.buffer_wired)
                return;                         /* nothing to do */

        pbu_$unwire(bmcb.pbu_unit_number,
                    (void *)bmcb.bufaddr,
                    bmcb.buflen,
                    (boolean)bmcb.command == BM_READ_CMD,
                    &st);

        /*
         * If returned status is non-zero, we may have an error on error
         * condition. Since we don't want to overlay the error code from
         * the original error, just print the error message here.
         */

        if (st.all != 0)
                error_$print(st);

        bmcb.flags.b.buffer_wired = false;
}
```

```
/* bm_command -- Common internal command processing for read/write
 * routines.

 * This routine:  (1) finishes common argument validation;
 *                (2) wires down the user's buffer;
 *                (3) calls the internal bm_$sio routine to start the
 *                    transfer.
 */

static void
bm_$command(
    unsigned char            command,       /* read or write */
    bm_$buf_t                *buffer,       /* buffer for transfer */
    bm_$buf_len_t            len,           /* length in bytes of
                                               buffer */
    bm_$bm_address_t         bm_address,    /* bulk memory address
                                               to use */
    status_$t                *status)
{
        /*
         * Make sure the controller has been initialized, it's not busy,
         * and that we have valid parameters for the transfer.
         */

        if (!bmcb.flags.b.init) {
                status->all = bm_$not_init;
                return;
        }

        if (bmcb.flags.b.busy) {   /* make sure it isn't already busy */
                status->all = bm_$busy;
                return;
        }

        if ((len <= 0) || (len > bm_$max_len)) {
                status->all = bm_$bad_length;
                return;
        }

        if ((bmcb.bufaddr < 0) || ((linteger)(bmcb.bufaddr+len) >
             pbu_$max_virtual_address)) {
                status->all = bm_$bad_address;
                return;
        }

        if ((bm_address < 0) || ((linteger) (bm_address+len) >
             bm_$max_address)) {
                status->all = bm_$bad_bm_address;
                return;
        }
        /* Wire down the buffer. */

        bmcb.bufaddr = buffer;      /* save address of buffer */
        bmcb.buflen = len;          /* save length of buffer */

        pbu_$wire(bmcb.pbu_unit_number,
                  (void *)buffer,
                  bmcb.buflen,
                  status);

        if (status->all != 0) {
                status->fail = 1;
                return;
        }

        bmcb.flags.buffer_wired = 1; /* remember we wired the buffer */

        /*
         * Buffer is all ready. Call the internal start I/O (sio)
         * routine to map the buffer and load the controller registers.
         * (bm_$sio, because it is also called from the interrupt side
         * of the driver, is defined in bm_int_lib.c.
         */

        bmcb.command = command;          /* command to perform */
        bmcb.io_addr = bmcb.bufaddr;     /* first address to transfer */
        bmcb.rem_len = len;              /* length "remaining" to
                                            transfer */
        bmcb.bm_address = bm_address;    /* where to start in the bm */
        bm_$sio(status);                 /* start up the operation */

        if (status->all != 0) {
            status->fail = 1;
            unwire_buffer();
            return;
        }

        /* Enable interrupts from the bm controller. */

        pbu_$enable_device(bmcb.pbu_unit_number, status);
}

/* bm_$cleanup -- Cleanup pbu logic.
 * This routine is called by pbu_$release when the user issues the rldev
 * command. */

void
bm_$cleanup(
    pbu_$unit_t     *unit,
    char            *force,
    status_$t       *status)
{
        status_$t               st;
        bm_$status_t            bm_status;
        bm_$buf_len_t           rem_len;

        /* If there's an operation in progress, attempt to clean up
         * nicely. */

        if (bmcb.flags.b.busy) {

                /* If user said -force, then forcibly reset the
                 * controller. */
                if (*force)
                        bmcb.csr_ptr.c->command = bmcb.read_cmd;
                else {
                        bm_$wait(5, &bm_status, &rem_len, status);
                        if (status->all != 0) {     /* probably a
                                                       timeout */
                                status->fail = 1;   /* couldn't clear
                                                       controller */
                                return;
                        }
                }
        }

        /* Give back our iomap space if we have any. */

        if (bmcb.bm_iova != 1) {           /* (1 is impossible iova --
                                               see bm_$init) */
                pbu_$free_map(bmcb.pbu_unit_number, &st);
                if (st.all != 0)
                        error_$print(st);
                bmcb.bm_iova = 1;   /* no longer have any iomap space */
        }

        /* Disable the device to prevent further interrupts. */

        pbu_$disable_device(bmcb.pbu_unit_number, status);
        bmcb.flags.init = 0;               /* no longer initialized */
}

/*  bm_$init -- Initialize BM library.
 *  Since it is being called from Pascal, all parameters are passed by
 * reference */

void
bm_$init(
    pbu_$unit_t             *unit,          /* pbu unit number */
    pbu_$ddf_ptr_t          *ddf_ptr,
    pbu_$csr_page_ptr_t     *csr_ptr,
    status_$t               *status)
{
        printf("unit = %d\n", *unit);
```

```
/* Save the information passed by pbu_$acquire in the bmcb. */

bmcb.pbu_unit_number = *unit;   /* unit number to pass pbu
                                   manager */
bmcb.ddf_ptr = *ddf_ptr;        /* pointer to mapped ddf */
bmcb.csr_ptr.p = *csr_ptr;      /* pointer to mapped controller
                                   page */

/*
 * Initialize the controller. We don't want to try loading the
 * command register ourselves yet because if the controller
 * doesn't exist, we'll get a bus-timeout fault and be
 * unceremoniously dumped back to shell command level.
 */

bmcb.flags.all = 0;   /* nothing going on yet and not
                         initialized */
bmcb.bm_iova = 1;     /* this tells cleanup routine that we
                         haven't gotten iomap space yet */

printf("csr page at %X\n", bmcb.csr_ptr.c);

pbu_$write_csr(bmcb.pbu_unit_number,     /* number of this pbu
                                            device */
        (char)bmcb.csr_ptr.c->command,  /* the command
                                            register */
        BM_INIT_CMD,                     /* initialization
                                            command */
        false,                           /* do a byte, not word
                                            write to command
                                            reg */
        status);                         /* returned status */

if (status->all == pbu_$bus_timeout) { /* controller probably
                                            not there if error */
        status->all = bm_$no_controller;
        return;
}
else if (status->all != 0) {
        status->s.fail = 1;
        return;
}
```

```
        /* Allocate an area of the iomap corresponding to the largest
         * block we are going to read or write. */

        bmcb.bm_iova = pbu_$allocate_map(bmcb.pbu_unit_number,/*number
                                                                of this
                                                                pbu
                                                                device*/
                                         bm_$block_len,   /* maximum
                                                             block size
                                                             we´ll use */
                                         false,           /* don´t need a
                                                             specific
                                                             iova */
                                         0,               /* forced iova
                                                             would go
                                                             here */
                                         status);         /* returned
                                                             status */

        if (status->all != 0) {
                status->s.fail = 1;
                return;
        }

        /* Define controller commands for loading into csr command
         * register.
         * NOTE: THESE COMMANDS SHOULD NOT BE DEFINED AS PASCAL
         * CONSTANTS!
         * The reason for this is that the compiler, when setting a csr
         * page register to a value it knows to be zero, will generate a
         * CLR instruction, which, since a CLR does a read-modify-write,
         * will probably cause a bus timeout error. To be safe, we just
         * make sure that all command values are in
           variables.  */

        bmcb.init_cmd  = BM_INIT_CMD;    /* initialization command */
        bmcb.read_cmd  = BM_READ_CMD;    /* read command */
        bmcb.write_cmd = BM_WRITE_CMD;   /* write command */

        /*
         * We could enable interrupts from the controller here, but
         * we´ll wait until we actually start an operation -- see
         * bm_command above.
         */

        bmcb.flags.b.init = 1;   /* note we´re initialized */
}

/* bm_$read -- Read from bulk memory.
 * This routine reads a block of memory from the bulk memory device into
 * Apollo memory.  This should probably be a macro for maximum
 * performance.
 */
```

```
void
bm_$read(
    bm_$buf_t               buffer,
    bm_$buf_len_t           buflen,
    bm_$bm_address_t        bm_addr,
    status_$t               *s)
{
    bm_command(bmcb.read_cmd, buffer, buflen, bm_addr, s);
}

/* bm_$wait -- Wait for completion of read or write operation.
 * This routine waits for the completion of a bulk memory transfer.
 * Note that for bm_$wait a timeout value of zero means wait forever.
 * This is unlike pbu_$wait, for which a timeout value of zero means
 * return immediately.
 */

void
bm_$wait(
    short           timeout,
    bm_$status_t    *bm_status,     /* controller status */
    bm_$buf_len_t   *rem_len,       /* residual count */
    status_$t       *status)
{
        int                         pbu_timeout;
        pbu_$wait_index_t           index;
        status_$t                   st;

        /* If there's an operation in progress, attempt to clean up
         * nicely.
         */

        if (!bmcb.flags.b.init) {
                status->all = bm_$not_init;
                return;
        }

        if (!bmcb.flags.b.busy) { /* don't wait if no transfer
                                     started */
                status->all = bm_$transfer_not_started;
                return;
        }
        /*
         * Check to see if the operation has already completed ('done'
         * flag set). If it is, we don't have to bother calling
         * pbu_$wait. Note that the done flag may be set AFTER we check
         * it and BEFORE we call pbu_$wait, but this is ok -- pbu_$wait
         * will realize that the event we want to wait for has already
         * happened and return immediately.
         */

        status->all = status_$ok;   /* assume o.k. */

        if (!bmcb.flags.done) {
                pbu_timeout = timeout;   /* value in seconds */
                pbu_timeout = (pbu_timeout == 0) ? (3600 * 1000) :
(pbu_timeout * 1000);

                /*
                 * We want the ability to handle any faults through the
                 * return value of pbu_$wait, when we enable again, we
                 * will get the fault.  If we did not inhibit before the
                 * pbu$wait call, and we received a fault, we would not
                 * be able to cleanup (unmap and unwire buffer) since we
                 * would be blasted back to the shell or the last fault
                 * handler.
                 */

                pfm_$inhibit();                         /* inhibit
                                                           faults */
                index = pbu_$wait(bmcb.pbu_unit_number,
                                  pbu_timeout,          /* number of
                                                           milliseconds
                                                           to wait */
                                  true,                 /* true means
                                                           allow quits
                                                           while
                                                           waiting */
                                  status);

                if (status->all != 0) {   /* he didn't like something */
                        status->s.fail = 1;
                        return;
                }
        }
        else index = 0;                  /* transfer already complete */

        switch (index) {
        case 0:
                /* The operation completed. Get the ending status and
                 * length transferred for the caller.
                 */
                bm_status->all = bmcb.status.all;
                if (bmcb.status.all == bm_$sio_error)
                        status->all = bmcb.sio_status.all;
                else if (bmcb.status.all != bm_$status_ok)
                        status->all = bm_$io_error;
                *rem_len = bmcb.rem_len;            /* residual count */
                break;
        case 1:
                /* the operation did not complete in time. */
                status->all = bm_$timeout;
                break;
        case 2:
                /*
                 * the user typed control-q while we were waiting. Note:
                 * the standard system fault catcher will blast us
                 * directly back to shell command level, so we'd never
                 * get here. But just in case the fault catcher chooses
                 * to ignore the quit, we'll handle it.
                 */
                status->all = bm_$quit_during_wait;
                break;
        default:
                printf("Invalid pbu_$wait index value, %d\n", index);
        }

        /* Unmap and unwire the buffer. */

        pbu_$unmap(bmcb.pbu_unit_number,
                   (void *)bmcb.bufaddr,  /* the buffer */
                   bmcb.io_len,           /* length mapped */
                   bmcb.bm_iova,          /* where it's mapped */
                   &st);                  /* returned status */

        if (st.all != 0)
                error_$print(st);

        unwire_buffer();       /* unwire the buffer regardless of how
                                  operation completed */
        bmcb.flags.busy = 0;   /* controller is no longer busy */
        pfm_$enable();         /* enable again */

        /* If we did receive a fault, we will take it here, but at least
         * we were able to cleanup.
         */
}

/* bm_$write -- Write to bulk memory.
 * This routine writes a block of memory from Apollo memory into the
 * bulk memory device.  This should probably be a macro for maximum
 * performance.
*/
```

```
void
bm_$write(
    bm_$buf_t               buffer,
    bm_$buf_len_t           buflen,
    bm_$bm_address_t        bm_addr,
    status_$t               *s)
{
    bm_command(bmcb.write_cmd, buffer, buflen, bm_addr, s);
}
```

E.3 bm_int_lib.c

The **bm_int_lib.c** file consists of the interrupt routine (**bm_$int**) and the start I/O routine (**bm_$sio**).

```
/* This module is the interrupt handler for a fictitious pbu device -- a
 * bulk memory (BM) unit. The intent of this routine is to show the
 * general structure of a user-space interrupt handler and to
 * demonstrate the use of the pbu manager routines.

 * The bulk memory unit is a pbu device whose controller is at address
 * 400 (hex) in the pbu address space. It has a 8-bit command and status
 * registers at addresses 400 and 401, a 32-bit bulk memory address
 * register at 402, a 16-bit count register at 406, and a 16-bit i/o
 * virtual address (iova) register at 408.  The controller interrupts at
 * level 2.

 * The bm manager supports three operations -- read from bulk memory,
 * write to bulk memory, and wait for transfer complete. Up to a
 * megabyte can be transferred with one call, but since the pbu can
 * transfer only up to 64K in one i/o operation, the interrupt side of
 * the driver (this routine) is given the job of blocking large
 * transfers into chunks of size bm_$block_len.  (Note that
 * bm_$block_len is not the maximum possible, which is 64K.  The reason
 * for not allowing 64K transfers is that it would require we take over
 * the entire iomap.  Therefore if another pbu device is using even a
 * single page of the iomap, our call to pbu_$allocate_map would fail.)
 */

#include <apollo/base.h>
#include "/latest/sr10/gpio/usr/include/apollo/pbu.h"
#include "bm.h"

bm_$bmcb_t bmcb;

pbu_$interrupt_return_t
bm_$int(
    pbu_$unit_t *unit)
{
```

```
/* We're called from the internal pbu interrupt handler when an
 * interrupt is received from the bm. (Note: we could call
 * pbu_$unmap here to unmap the last buffer, but choose not to:
 * if another chunk of the buffer needs to be transferred,
 * mapping the new chunk (see bm_$sio) will effectively unmap
 * the chunk that was just transferred.  If there is no more of
 * the buffer to be transferred, we will wake up the call side
 * of the driver and the bm_$wait routine will unmap the last
 chunk of the buffer.
 *
 * Since we only enable the controller when we've started a
 * transfer, we're pretty sure this is a valid interrupt.  For
 * debugging, or if a controller is left enabled all the time,
 * it might be prudent to make sure this interrupt is expected.
 */

bmcb.flags.b.done = 1;                  /* transfer completed */
bmcb.status.all = bmcb.csr_ptr.c->status;     /* read the status
                                                 and save for
                                                 call side */

/*
 * If an error occurred on last transfer, don't try to continue
 * the operation.  Just wake up the call side to process the bad
 * status.
 */

if (bmcb.status.all != bm_$status_ok)
        return(pbu_$interrupt_advance);       /* advance bm's
                                                 event count */

/*
 * Last transfer completed ok. Decrement the length remaining to
 * be transferred and see if there's more to do.
 */

bmcb.rem_len = bmcb.rem_len - bmcb.io_len;   /* decrement length
                                                remaining to
                                                transfer */

if (bmcb.rem_len == 0 )                     /* we're all done */
        return(pbu_$interrupt_advance);     /* tell call side
                                               we're done */

/*
 * There's more to do. Calculate start of the next chunk of
 * buffer to be transferred and call bm_$sio to start the
 * transfer.
 */

bmcb.io_addr = (bm_$buf_ptr_t) ((char *) bmcb.io_addr +
                bmcb.io_len);
bmcb.bm_address = bmcb.bm_address + bmcb.io_len; /* start in the bulk
                                                    memory */
        bm_$sio(&bmcb.sio_status);        /* call internal start I/O
                                             (sio) routine to start up
                                             controller */

        if (bmcb.sio_status.all != 0) {   /* oops -- bm_$sio had a
                                             problem */

                /*
                 * Note that since we're on in an interrupt routine, we
                 * can't do much about this error, for example, call
                 * error_$print.  So we'll just save the bad status for
                 * inspection by the call side of the driver.
                 */

                bmcb.status.all = bm_$sio_error;
                return(pbu_$interrupt_advance);   /* wake him up */
        }

        /*
         * The transfer was started ok, so tell the pbu interrupt logic
         * to re-enable interrupts from the controller.
         */

        return(pbu_$interrupt_enable);   /* want to get another
                                            interrupt */
}

/* bm_$sio -- Start I/O operation to bulk memory controller.
 * This routine maps (a part of) the buffer and loads the controller
 * registers to start an i/o operation. Since this routine is called
 * from both bm_command (in the call side of the driver) and from the
 * interrupt handler, it must be loaded with the interrupt handler.
 */

void
bm_$sio(
    status_$t *status)
{
        bmcb.csr_ptr.c->bm_address = bmcb.bm_address; /* tell controller
                                                         where to start
                                                         in bulk memory
                                                       */

        /*
         * If the buffer length is less than or equal to bm_$block_len
         * then we can do the whole thing at once.  Otherwise, start
         * with a block of length bm_$block_len.  The interrupt
         * routine will start the next chunk.
         */
        bmcb.io_len = (bmcb.rem_len <= bm_$block_len) ? bmcb.rem_len :
                      bm_$block_len;
        bmcb.csr_ptr.c->count = bmcb.io_len;    /* give byte count to
                                                   controller */

        /*
         * Map the buffer through the area of iomap that we allocated at
         * initialization time and give the controller the pbu address.
         */

        bmcb.csr_ptr.c->iova = pbu_$map(bmcb.pbu_unit_number, /* number
                                                      of this pbu unit */
                              (void *)bmcb.bufaddr, /* virtual address
                                                        of buffer */
                               bmcb.io_len,      /* length of buffer */
                               bmcb.bm_iova,     /* iova we got from
                                                    pbu_$allocate_map */
                               status);          /* returned status */

        if (status->all != 0)
                return;

        /*
         * All set to start operation. Set our internal flags and load
         * command register to fire up controller.
         */

        bmcb.flags.b.busy = 1;        /* controller will be busy after
                                         loading command reg */
        bmcb.flags.b.done = 0;        /* transfer hasn't completed yet */
        bmcb.csr_ptr.c->command = bmcb.command;   /* start read or write
                                                     operation */
}
```

E.4 makefile

The makefile script organizes the files that make up the driver into the call–side and interrupt–side libraries when the driver is bound.

```
all: bm_lib bm_int.lib

bm_lib: bm_lib.bin
         bind -allmark bm_lib.bin -b bm_lib -map -sys

bm_int.lib: bm_int_lib.bin
         bind -allmark bm_int_lib.bin /lib/pbu_int_lib -b bm_int.lib
                                                       -map -sys

bm_lib.bin: bm_lib.c bm.ins.c
         /com/cc bm_lib.c -ndb

bm_int_lib.bin: bm_int_lib.c bm.ins.c
         /com/cc bm_int_lib.c -ndb
```

Appendix F

Sample Driver in Pascal

This appendix lists the files that make up the online device driver in the subdirectory **/domain_examples/gpio_examples/bm_example**. This version differs from the online version in two respects:

- Whereas in the online version the controller commands are assigned values in the initialization routine (**bm_$init**), here they are declared as constants in **bm.pvt.pas**. This is permissible because the CSR page definitions in **bm.pvt.pas** have been marked with the [DEVICE] attribute. For information on the [DEVICE] attribute, refer to Appendix C, Section C.3.

- A private insert file, **bm.pvt.pas**, has been added, and some of the data structures and routines formerly in the public insert file, **bm.ins.pas**, have been moved over to this new file. This change does not affect the running of the driver, but it does show the format of a private insert file.

Both the functional parts and the operation of this driver are fully described in Chapter 4, Subsections 4.4.2 and 4.4.3 and Figure 4-2. For additional information about the driver and the hypothetical bulk-memory controller it supports, refer to the header comments in **bm_lib.pas** (Section F.3). An identical version of this driver coded in C is listed in Appendix E.

Four files make up the **bm_example** driver:

- Private insert file: **bm.pvt.pas**

- Public insert file: **bm.ins.pas**

- Call-side module: **bm_lib.pas**

- Interrupt-side module: **bm_int_lib.pas**

F.1 bm.pvt.pas

The **bm.pvt.pas** file declares the private storage area for the interrupt and call sides of the driver. Specifically, it declares the controller command constants, the CSR page (bm_$csr_page_t), the control block used by the driver (bm_$bmcb_t), and the internal start I/O routine (**bm_$sio**).

```
{  bm.pvt.pas, private definitions for bulk memory device driver }

{ Define controller commands for loading into csr command register. }

CONST bm_init_cmd  := chr(16#00);   { initialization command }
       bm_read_cmd  := chr(16#01);   { read command }
       bm_write_cmd := chr(16#02);   { write command }

{ Define the bulk memory controller's csr page. (Note:  when defining
the contents of a csr page, watch out for the compiler's rules about
packing records. In particular, avoid using records inside the csr page
record, since embedded records are word-aligned, even in a packed re-
cord. For example, we might have defined the status register to be
bm_$status_t (see below), but then the compiler would have aligned it at
offset 2 in the page even though bm_$status_t is only 8 bits wide.) }

TYPE bm_$csr_page_t = [DEVICE] PACKED RECORD
     command : char;           { 00 one byte command register at offset
                                 0}
     status  : char;           { 01 one byte status register }
     iova : integer;           { 02 io virtual address to use for
                                 transfer }
     count : integer;          { 04 number of bytes to transfer }
     bm_address : bm_$bm_address_t;   { 06 bulk memory address to
                                        read/write }
     end;    { of bm_$csr_page_t }

     bm_$csr_page_ptr_t = RECORD CASE INTEGER OF
         0:(c : ^bm_$csr_page_t);
         1:(p : pbu_$csr_page_ptr_t);
         end;    { of bm_$csr_page_ptr_t }


{ Define the bulk memory control block (bmcb). This area is used for
communications  between the call and interrupt sides of the bm driver.
Since it is referenced  by the interrupt handler, it must be part of the
interrupt library -- see bm_int_lib.pas. }

TYPE bm_$flags_t = PACKED RECORD CASE INTEGER OF { define flags field in
                                                   bmcb }
  0:(init : boolean;           { set to true when controller initialized}
     buffer_wired : boolean;   { set when a buffer is wired }
     busy  : boolean;          { set to true when operation in progress }
     done   : boolean;          { set by interrupt routine when transfer
                                  completes }
     pad    : SET OF 0..3);     { fill out to byte }
  1:(all : binteger);
     end; { of bm_$flags_t }

TYPE bm_$status_t = PACKED RECORD CASE INTEGER OF  { define status
                                                     register }
     0:(attention : boolean;             { 1 => change in controller
                                                status }
        status_modifier : boolean;       { 1 => current status
                                                unavailable }
        control_unit_end : boolean;      { 1 => busy condition cleared }
        busy : boolean;                  { 1 => controller currently
                                                busy}
        channel_end : boolean;           { 1 => end of operation }
        device_end : boolean;            { 1 => end of operation }
        unit_check : boolean;            { 1 => parity error in bm }
        unit_exception : boolean);       { 1 => illegal bm address }
     1:(all : char);
        end;    { of bm_$status_t }

CONST bm_$status_ok = chr(16#0C);     { normal completion status }
       bm_$sio_error = chr(16#FF);    { interrupt routine got error
                                       from bm_$sio (start I/O routine) }

TYPE bm_$bmcb_t = RECORD              { define communications area }
     pbu_unit_number : pbu_$unit_t;   { number of this pbu device }
     flags : bm_$flags_t;             { a byte of flags }
     pad : SET OF 0..7;               { a byte of padding }
     ddf_ptr : pbu_$ddf_ptr_t;        { pointer to mapped ddf }
     csr_ptr : bm_$csr_page_ptr_t;    { pointer to mapped csr page }
     bm_iova : pbu_$iova_t;           { start of our area of i/o
                                       address space }
     bufaddr : bm_$both_t;            { address of start of buffer }
     buflen : bm_$buf_len_t;          { total length of buffer }
     bm_address : bm_$bm_address_t;   { address of start of bm area }
     command : char;                  { current command (read or write) }
     rem_len : bm_$buf_len_t;         { length remaining to read or
                                        write}
     status : bm_$status_t;           { status from last interrupt }


sio_status : status_$t;          { status from bm_$sio called from
                                   int side }
     io_addr : bm_$both_t;       { address of last i/o transfer }
     io_len : bm_$buf_len_t;     { length of last i/o transfer }
     end;    { of bm_$bmcb_t }

{ Define global routines not visible to the user. }
```

```
PROCEDURE bm_$cleanup (                      { called from pbu_$release }
               IN  unit: pbu_$unit_t;         { pbu unit number }
               IN  force : boolean;           { force flag }
               OUT status : status_$t         { returned status }
               ); EXTERN;

PROCEDURE bm_$init (                            { called from pbu_$acquire}
               IN  unit : pbu_$unit_t;           { pbu unit number }
               IN  ddf_ptr : pbu_$ddf_ptr_t;     { pointer to mapped ddf }
               IN  csr_ptr : pbu_$csr_page_ptr_t;   { pointer to mapped
                                                     csr page }
               OUT status : status_$t            { returned status }
               ); EXTERN;

PROCEDURE bm_$sio (OUT status : status_$t); EXTERN; { start i/o
                                                      operation }
```

F.2 bm.ins.pas

The **bm.ins.pas** file is the interface between the application and the driver; it defines error codes, buffer parameter information, and driver entry points (**bm_$read**, **bm_$write**, and **bm_$wait**).

```
{ bm.ins.pas, insert file for users of bulk memory device }


{ Error codes from bm manager calls. (We've arbitrarily picked a subsystem code of 0F.) }

CONST bm_$no_controller        = 16#0F000001 ; { controller not present }
      bm_$not_init             = 16#0F000002 ; { controller not initialized }
      bm_$busy                 = 16#0F000003 ; { controller is busy }
      bm_$not_ready            = 16#0F000004 ; { unit not ready }
      bm_$bad_address          = 16#0F000005 ; { buffer beyond protection boundary }
      bm_$bad_length           = 16#0F000006 ; { bad buffer length }
      bm_$bad_bm_address       = 16#0F000007 ; { bad bm address }
      bm_$transfer_not_started = 16#0F000008 ; { tried to wait before read or write }
      bm_$timeout              = 16#0F000009 ; { timeout during wait }
      bm_$quit_during_wait     = 16#0F00000A ; { quit during wait }
      bm_$io_error             = 16#0F00000B ; { i/o error during transfer }

      bm_$max_address          = 2147483647 ;  { maximum bm address =2**31 - 1 }
      bm_$block_len            = 32768 ;       { maximum transfer per i/o operation = 32K }
      bm_$max_len              = 131072 ;      { maximum amount to transfer per call = 128K
                                                 N.B.: MUST be multiple of bm_$block_len
                                                 (see bm_$int)! }
```

```
TYPE bm_$buf_len_t = 1..bm_$max_len;  { bm buffer dimension }

TYPE bm_$buf_t = ARRAY [bm_$buf_len_t] OF INTEGER;
       bm_$buf_ptr_t = ^bm_$buf_t;

TYPE bm_$bm_address_t = integer32;   { address of block in bulk memory }


TYPE bm_$both_t = RECORD CASE INTEGER OF    { for handling buffer pointers }
      0:(p : bm_$buf_ptr_t);
      1:(i : integer32);
       end;   { of bm_$both_t }


{ Define the application-visible library entry points.  }

PROCEDURE bm_$read (                              { read record }
           OUT buffer : UNIV bm_$buf_t;         { data buffer }
           IN  buflen : UNIV bm_$buf_len_t;    { buffer length }
           IN  bm_address : UNIV bm_$bm_address_t;  { address in bulk memory }
           OUT status : status_$t              { returned status }
       ); EXTERN;


PROCEDURE bm_$wait (                      { wait for transfer completion }
            IN  timeout : integer;       { optional timeout value (secs) }
            OUT bm_status : bm_$status_t;      { status from controller }
            OUT rem_len : UNIV bm_$buf_len_t; { residual count }
            OUT status : status_$t              { return code }
            ); EXTERN;


PROCEDURE bm_$write (                         { write record }
            IN  buffer : UNIV bm_$buf_t;     { data buffer }
            IN  buflen :UNIV bm_$buf_len_t; { buffer length }
           IN  bm_address : UNIV bm_$bm_address_t;  { address in bulk memory }
            OUT status : status_$t           { returned status }
        ); EXTERN;
```

F.3 bm_lib.pas

The **bm_lib.pas** file consists of the call-side routines that perform initialization (**bm_$init**), cleanup (**bm_$cleanup**), command processing (**bm_$read, bm_$write,** and **bm_command**), and wait for interrupt (**bm_$wait**).

```
{ bm.pas, device driver library for bulk memory device }

{  This module is the device driver library for a hypothetical pbu (pe-
ripheral bus unit) -- a bulk memory (BM) unit.  The intent of the driver
is to show the general structure of a user-space device driver and to
demonstrate the use of the pbu manager routines.

The bulk memory unit is a pbu device whose controller is at address 400
(hex) in the pbu address space. It has an 8-bit command and status reg-
isters at  addresses 400 and 401, a 32-bit bulk memory address register
at 402, a 16-bit count register at 406, and a 16-bit i/o virtual address
(iova) register at 408. The controller interrupts at level 2.

The controller is initialized by writing 16#00 to the command register.
Read and write operations are performed by loading the address, count,
and iova registers the then writing a 16#01 (read) or 16#02 (write) to
the command register. Status is obtained by reading the status register.

The bm manager (this module) supports three operations -- read from bulk
memory, write to bulk memory, and wait for transfer complete. Up to a
128K can be transferred with one call, but since the pbu cannot transfer
128K in one i/o operation, the interrupt side of the driver (see
bm_int_lib.pas) is given the job of blocking large transfers into chunks
of size bm_$block_len.  (Note that bm_$block_len is not the maximum pos-
sible, which is 64K. The reason for not allowing 64K transfers is that
it would require we take over the entire iomap. Therefore, if another
pbu device is using even a single page of the iomap, our call to
pbu_$allocate_map would fail.)

A typical invocation of the bm library might appear as follows:

    VAR data_buffer : ARRAY[0..buf_size] OF CHAR;
         status : status_$t;
         bm_status : bm_$status_t;
         bytes_left : integer32;

    bm_$write(data_buffer,1024*10,0,status);    write 10 pages to bm
                                                addr 0
     IF status.all <> 0 THEN BEGIN
         error_$print(status);                  display error code
         GOTO process_error;
         END;
   bm_$wait(1,bm_status,bytes_left,status);   wait 1 second for
                                              completion
    IF status.all <> 0 THEN BEGIN
        error_$print(status);                display error code
        IF status.all := bm_$io_error THEN display_status_byte;
        GOTO process_error;
        END;                       }


MODULE bm;

DEFINE bm_$cleanup,
        bm_$init,
        bm_$read,
        bm_$wait,
        bm_$write;

%nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/vfmt.ins.pas';
 %include '/sys/ins/error.ins.pas';
 %include '/sys/ins/pbu.ins.pas';
 %include '/sys/ins/pbu_acquire.ins.pas';
 %list;
 %include 'bm.ins.pas';
 %include 'bm.pvt.pas';
 %eject;

VAR bmcb : EXTERN bm_$bmcb_t;         { bulk memory control block
                                        (defined in bm_int_lib.pas) }

PROCEDURE unwire_buffer; INTERNAL;    { internal routine to unwire
                                        a buffer }
 VAR st : status_$t;
 BEGIN
     IF NOT bmcb.flags.buffer_wired THEN RETURN;    { nothing to do }

    pbu_$unwire(bmcb.pbu_unit_number,        { number of this pbu unit }
                bmcb.bufaddr.p^,              { buffer to unwire }
                bmcb.buflen,                  { length of buffer }
                bmcb.command = bm_read_cmd,   { touch pages if read
                                                command }
                st);                          { returned status }

{ If returned status is nonzero, we may have an error on error condi-
tion.  Since we don't want to overlay the error code from the original
error, just print the error message here. }

    IF st.all <> 0 THEN error_$print(st);

    bmcb.flags.buffer_wired := false;

END;   { of unwire_buffer }
 %eject;
```

```
{  BM_COMMAND -- Common internal command processing for read/write rou-
tines. }

{ This routine:

          (1) finishes common argument validation;
          (2) wires down the user's buffer;
          (3) calls the internal bm_$sio routine to start the transfer. }

PROCEDURE bm_command (
                IN  command : char;                { command byte (read
                                                     or write) }
                IN  buffer : UNIV bm_$buf_t;       { buffer for
                                                     transfer }
                IN  len : bm_$buf_len_t;           { length in bytes of
                                                     buffer }
                IN  bm_address : bm_$bm_address_t; { bulk memory
                                                     address to use }
                OUT status : status_$t); INTERNAL; { returned status }


VAR
     i, j : integer;
     temp : bm_$buf_len_t;
     st : status_$t;

BEGIN

{ Make sure the controller has been initialized, it's not busy, and that
we have valid parameters for the transfer. }

    IF NOT bmcb.flags.init THEN BEGIN
         status.all := bm_$not_init;
         RETURN;
         END;

    IF bmcb.flags.busy THEN BEGIN    { make sure controller isn't already
                                       busy }
         status.all := bm_$busy;
         RETURN;
         END;

    IF (len <= 0) OR (len > bm_$max_len) THEN BEGIN
         status.all := bm_$bad_length;
         RETURN;
         END;

    bmcb.bufaddr.p := addr(buffer);    { save address of buffer }


    IF (bmcb.bufaddr.i < 0) OR (bmcb.bufaddr.i+len >
     pbu_$max_virtual_address) THEN BEGIN
         status.all := bm_$bad_address;
         RETURN;
         END;

     IF (bm_address < 0) OR (bm_address + len > bm_$max_address) THEN
        BEGIN
        status.all := bm_$bad_bm_address;
        RETURN;
        END;

{ Wire down the buffer. }

     bmcb.buflen := len;              { save length of buffer }

     pbu_$wire(bmcb.pbu_unit_number,  { number of this pbu unit }
               buffer,                { buffer to wire }
               bmcb.buflen,           { length to wire (in bytes) }
               status);               { returned status }

     IF status.all <> 0 THEN BEGIN;   { give up if something wrong }
         status.fail := true;
         RETURN;
         END;

     bmcb.flags.buffer_wired := true; { remember we wired the buffer }

{ Buffer is all ready. Call the start I/O routine (sio) routine to map
  the buffer and load the controller registers. (Because bm_$sio is
  called from the interrupt side of the driver, it is defined in
  bm_int_lib.pas. }

        bmcb.command := command;         { command to perform }
         bmcb.io_addr := bmcb.bufaddr;   { first address to transfer }
         bmcb.rem_len := len;            { length "remaining" to
                                           transfer}
         bmcb.bm_address := bm_address;  { where to start in the bm }
         bm_$sio(status);                { start up the i/o operation }
         IF status.all <> 0 THEN BEGIN;
             status.fail := true;
             unwire_buffer;
             RETURN;
             END;


{ Enable interrupts from the bm controller. }

        pbu_$enable_device(bmcb.pbu_unit_number,     { number of this pbu
                                                       device }
                            status);                 { returned status }

        RETURN;

END;  { of BM_COMMAND }
 %eject;
```

```
{  BM_$CLEANUP -- Cleanup pbu logic.  }

PROCEDURE bm_$cleanup (*              { called by pbu_$release }
               IN  unit : pbu_$unit_t;
               IN  force : boolean;
               OUT status : status_$t
              *);

VAR st : status_$t;
     bm_status : bm_$status_t;
     rem_len : bm_$buf_len_t;

BEGIN

{ If there's an operation in progress, attempt to clean up nicely. }

    IF bmcb.flags.busy THEN

{ If user said -force, then forcibly reset the controller. }

        IF force THEN bmcb.csr_ptr.c^.command := bm_init_cmd

{ If user didn't say -FORCE, wait 5 seconds for operation to complete. }

        ELSE BEGIN
             bm_$wait(5,bm_status,rem_len,status);
             IF status.all <> 0 THEN BEGIN   { probably a timeout }
                 status.fail := true; { couldn't clear controller}
                 RETURN;
                 END;   { of status <> 0 }
             END;   { of ELSE }

{ Give back our iomap space if we have any. }

    IF bmcb.bm_iova <> 1 THEN BEGIN { (1 is impossible iova--see
                                      bm_$init) }
         pbu_$free_map(bmcb.pbu_unit_number,{number of this pbu device }
                       st);                   { returned status }
         IF st.all <> 0 THEN error_$print(st);
         bmcb.bm_iova := 1;               { no longer have any iomap
                                            space }
         END;

{ Disable the device to prevent further interrupts. }

    pbu_$disable_device(bmcb.pbu_unit_number,  { number of this pbu
                                                 device }
                        status);               { returned status }

    bmcb.flags.init := false;                  { no longer initialized }

END;   { BM_$CLEANUP }
 %eject;
```

```
{   BM_$INIT -- Initialize BM library.   }

PROCEDURE bm_$init (*                    { called from pbu_$acquire }
          IN  unit : pbu_$unit_t;        { pbu unit number }
          IN  ddf_ptr : pbu_$ddf_ptr_t;
          IN  csr_ptr : pbu_$csr_page_ptr_t;
          OUT status : status_$t
         *) ;

{ This routine is called from pbu_$acquire to device-dependent initiali-
zation.  (Note: pbu_$acquire has already checked that the device isn't
already acquired, so we don't need to worry about it here.) }

VAR i : integer;

BEGIN

{ Save the information passed by pbu_$acquire in the bmcb. }

    bmcb.pbu_unit_number := unit;    { unit number to pass pbu manager }
     bmcb.ddf_ptr := ddf_ptr;        { pointer to mapped ddf }
     bmcb.csr_ptr.p := csr_ptr;      { pointer to mapped controller
                                       page}


{ Initialize the controller. We don't want to try loading the command
register ourselves yet because if the controller doesn't exist, we'll
get a bus-timeout fault and be unceremoniously dumped back to shell com-
mand level. }

    bmcb.flags.all := 0;    { nothing going on yet and not initialized }
     bmcb.bm_iova := 1;     { this tells clean-up routine that we
                              haven't gotten iomap space yet }

    vfmt_$write2('csr page at %lh%.',bmcb.csr_ptr.c,0); {*** temp ***}



    pbu_$write_csr(bmcb.pbu_unit_number,      { number of this pbu
                                                device }
                   bmcb.csr_ptr.c^.command,   { the command register }
                   ord(bm_init_cmd),          { initialization command }
                   false,                     { do a byte, not word
                                                write to command reg }
                   status);                   { returned status }

    IF status.all <> 0 THEN BEGIN   { controller probably not there if
                                      error }
       IF status.all = pbu_$bus_timeout THEN
          status.all  := bm_$no_controller ELSE status.fail := true;
       RETURN;
       END;
```

```
{ Allocate an area of the iomap corresponding to the largest block we
are going to read or write. }

    bmcb.bm_iova := pbu_$allocate_map(
                          bmcb.pbu_unit_number,    { number of this pbu
                                                     device }
                          bm_$block_len,           { maximum block size
                                                     we'll use }
                          false,                   { don't need a
                                                     specific iova }
                          0,                       { forced iova would go
                                                     here }
                          status);                 { returned status }

    IF status.all <> 0 THEN BEGIN
         status.fail := true;
         RETURN;
         END;

{ We could enable interrupts from the controller here, but we'll wait
until we actually start an operation -- see bm_command above. }

      bmcb.flags.init := true;    { note we're initialized }


END;   { of BM_$INIT }
 %eject;

{  BM_$READ -- Read from bulk memory. }

PROCEDURE bm_$read (*
          IN  unit : bm_$unit_t;
          IN  buffer : bm_$buf_t;
          IN  buflen : bm_$buf_len_t;
          IN  bm_address : bm_$bm_address_t;
          OUT status : status_$t;  *) ;

{ This routine reads a block of memory from the bulk memory device into
Apollo memory. }

BEGIN

    bm_command (bm_read_cmd, { let bm_command do all the work }
                       buffer,buflen,
                       bm_address,
                       status);

END;   { BM_$READ }
 %eject;
```

```
{  BM_$WAIT -- Wait for completion of read or write operation.   }

PROCEDURE bm_$wait (*                          { wait for DMA completion }
               IN  timeout : integer;          { optional timeout value
                                                 (secs) }
               OUT bm_status : bm_$status_t { status from controller }
               OUT rem_len : bm_$buf_len_t; { residual count }
               OUT status : status_$t       { return code }
         *);

{ This routine waits for the completion of a bulk memory transfer. Note
that for BM_$WAIT a timeout value of zero means wait forever. This is
unlike PBU_$WAIT, for which a timeout value of zero means return immedi-
ately. }

VAR
     pbu_timeout : integer32;
     st : status_$t;
     index : pbu_$wait_index_t;

BEGIN

    IF NOT bmcb.flags.init THEN BEGIN
         status.all := bm_$not_init;
         RETURN;
         END;

    IF NOT bmcb.flags.busy THEN BEGIN    { shouldn't wait if no transfer
                                          started }
         status.all := bm_$transfer_not_started;
         RETURN;
         END;

{ Check to see if the operation has already completed ('done' flag set).
If it is, we don't have to bother calling pbu_$wait. Note that the done
flag may be set AFTER we check it and BEFORE we call pbu_$wait, but this
is ok --pbu_$wait will realize that the event we want to wait for has
already happened and return immediately. }

    status.all := status_$ok;      { assume ok for now }

    IF NOT bmcb.flags.done THEN BEGIN

         pbu_timeout := timeout;    { value in seconds }
         IF pbu_timeout = 0 THEN pbu_timeout := 3600 * 1000 { default to
                                                               1 hour}
                           ELSE pbu_timeout := pbu_timeout * 1000;

           index := pbu_$wait(
                    bmcb.pbu_unit_number, { number of this pbu device }
                    pbu_timeout,      { number of milliseconds to wait }
                    true,       { true means allow quits while waiting }
                    status);                         { returned status }
```

```
            IF status.all <> 0 THEN BEGIN { pbu_$wait didn't like
                                            something }
                  status.fail := true;
                  RETURN;
                  END;

         END    { of not done }

     ELSE index := 0;    { transfer already complete }

     CASE index OF

{ If index = 0, the operation completed. Get the ending status and
length transferred for the caller. }

        0:  BEGIN
                 bm_status.all := bmcb.status.all;
                 IF bmcb.status.all = bm_$sio_error THEN
                    status := bmcb.sio_status
                 ELSE IF bmcb.status.all <> bm_$status_ok THEN
                    status.all := bm_$io_error;
                 rem_len := bmcb.rem_len;    { residual count }
                 END;

{ If index = 1, then the operation did not complete in time. }

        1:   status.all := bm_$timeout;

{ If index = 2, the user typed CTRL/Q while we were waiting. Note:  the
standard system fault catcher will blast us directly back to shell com-
mand level, so we'd never get here. But just in case the fault catcher
chooses to ignore the quit, we'll handle it. }

        2:   status.all := bm_$quit_during_wait;

         END;    { of CASE }

{ Unmap and unwire the buffer. }

     pbu_$unmap(bmcb.pbu_unit_number,    { number of this pbu unit }
                bmcb.bufaddr.p^,          { the buffer }
                bmcb.io_len,              { length mapped }
                bmcb.bm_iova,             { where it's mapped }
                st);                      { returned status }
      IF st.all <> 0 THEN error_$print(st);

     unwire_buffer;    { unwire the buffer regardless of how operation
                         completed }
      bmcb.flags.busy := false;    { controller is no longer busy }

END;    { of BM_$WAIT }
 %eject;
```

```
{  BM_$WRITE -- Write a record  }

PROCEDURE bm_$write (*
          IN  unit : bm_$unit_t;
          IN  buffer : bm_$buf_t;
          IN  buflen : bm_$buf_len_t;
          IN  bm_address : bm_$bm_address_t;
          OUT status : status_$t;  *) ;

{ This routine writes a block of processor memory out to the bulk memory
device. }

BEGIN

    bm_command(bm_write_cmd,     { let bm_command do all the work }
                buffer,
                buflen,
                bm_address,status);

END;   { BM_$WRITE }
 %eject;
```

F.4 bm_int_lib.pas

The **bm_int_lib.pas** file consists of the interrupt routine (**bm_$int**) and the start I/O routine (**bm_$sio**). Since the control block, like **bm_$sio**, is referenced by the interrupt routine, it must be DEFINEd here.

```
{  bm_int_lib.pas, interrupt handler for bulk memory device }

MODULE bm_int_lib;

DEFINE bmcb,  { define anything here that the interrupt routine has to
                 reference }
        bm_$sio;

%nolist;
 %include '/sys/ins/base.ins.pas';
 %include '/sys/ins/pbu.ins.pas';
 %include '/sys/ins/pbu_acquire.ins.pas';
 %include 'bm.ins.pas';
 %list;
 %include 'bm.pvt.pas';
 %eject;


VAR bmcb : EXTERN bm_$bmcb_t;    { bulk memory control block }

%eject;
 FUNCTION bm_$int : pbu_$interrupt_return_t;
```

```
{ We're called from the System Interrupt Handler when an interrupt is
received from the device.  (Note:  we could call pbu_$unmap here to un-
map the last buffer, but choose not to: if another portion of the
buffer needs to be transferred, mapping the new portion (see bm_$sio)
will effectively unmap the portion that was just transferred. If there
is no more of the buffer to be  transferred, we will wake up the call
side of the driver and the bm_$wait  routine will unmap the last chunk
of the buffer.) }

VAR st : status_$t;

BEGIN

    WITH bmcb.csr_ptr.c^ : csr DO BEGIN    { shorthand name for csr page}

{ Since we only enable the controller when we've started a transfer,
we're pretty sure this is a valid interrupt. For debugging, or if a con-
troller is left enabled all the time, it might be prudent to make sure
this interrupt is expected. Something like:

        if not bmcb.flags.busy then BEGIN
             set_bitchy_flag_for_call_side_or_cause_bus_timeout_error;
             bm_$int := [];   no advance, no enable
             return;
             END;            }

    bmcb.flags.done := true;          { transfer completed }
     bmcb.status.all := csr.status;   { read the status and save for
                                        call side }

{ If an error occurred on last transfer, don't try to continue the op-
eration.  Just wake up the call side to process the bad status. }

    IF bmcb.status.all <> bm_$status_ok THEN BEGIN
         bm_$int := [pbu_$interrupt_advance];    { advance bm's event
                                                    count }
         RETURN;
         END;

{ Last transfer completed ok. Decrement the length remaining to be
transferred and see if there's more to do. }

    bmcb.rem_len := bmcb.rem_len - bmcb.io_len;     { decrement length
                                                      remaining to trans-
fer }
     IF bmcb.rem_len = 0 THEN BEGIN                 { we're all done }
         bm_$int := [pbu_$interrupt_advance];       { tell call side
                                                      we're done }
         RETURN;
         END;
```

```
{ There's more to do. Calculate start of the next portion of buffer to
be transferred and call bm_$sio to start the transfer. }

    bmcb.io_addr.i := bmcb.io_addr.i + bmcb.io_len;      { start of next
                                                           chunk }
    bmcb.bm_address := bmcb.bm_address + bmcb.io_len;    { start in bulk
                                                            memory }
    bm_$sio(bmcb.sio_status);      { call internal start I/O (sio)
                                     routine to start up controller }
    IF bmcb.sio_status.all <> 0 THEN BEGIN    { oops -- bm_$sio had a
                                                problem }


{ Note that since we're in an interrupt routine, we can't do much about
this error, for example, call error_$print. So we'll just save the bad
status for inspection by the call side of the driver. }

        bmcb.status.all := bm_$sio_error;     { fake i/o status to tell
                                                him to look at
                                                sio_status }
         bm_$int := [pbu_$interrupt_advance];  { wake him up }
         END    { of st <> 0 }

{ The transfer was started ok, so tell pbu interrupt logic to re-enable
interrupts from the controller. }

    ELSE bm_$int := [pbu_$interrupt_enable];  { want to get another
                                                interrupt }
     RETURN;
     END;    { of WITH csr }

END;    { of BM_$INT }
 %eject;

{  BM_$SIO -- Start I/O operation to bulk memory controller.  }

PROCEDURE bm_$sio (* OUT status : status_$t *);

{ This routine maps (a part of) the buffer and loads the controller reg-
isters to start an i/o operation. Since this routine is called from both
bm_command (in the call side of the driver) and from the interrupt han-
dler, it must be loaded with the interrupt handler. }

BEGIN

    WITH bmcb.csr_ptr.c^ : csr DO BEGIN

        csr.bm_address := bmcb.bm_address;   { tell controller where to
                                               start in bulk memory }
```

```
{ If the buffer length is less than or equal to bm_$block_len then we
can do the whole thing at once.  Otherwise, start with a block of length
bm_$block_len. The interrupt routine will start the next chunk. }

        IF bmcb.rem_len <= bm_$block_len THEN bmcb.io_len :=
                                                          bmcb.rem_len
                                  ELSE bmcb.io_len :=
                                                        bm_$block_len;

        csr.count := bmcb.io_len;   { give byte count to controller }

{ Map the buffer through the area of iomap that we allocated at in-
itialization time and give the controller the pbu address. }

        csr.iova := pbu_$map(bmcb.pbu_unit_number,  { number of this
                                                       pbu unit }
                      bmcb.bufaddr,      { virtual address of
                                           buffer }
                      bmcb.io_len,       { length of buffer}
                      bmcb.bm_iova,      { iova we got from
                                           pbu_$allocate_map}
                      status);           { returned status }

        IF status.all <> 0 THEN RETURN;     { if error, just return }

{ All set to start operation. Set our internal flags and load command
register to fire up controller. }

        bmcb.flags.busy := true;        { controller will be busy after
                                          loading command reg }
         bmcb.flags.done := false;      { transfer hasn't completed yet}
         csr.command := bmcb.command;   { start read or write operation}

        END;   { of WITH csr }

END;   { of BM_$SIO }
 %eject;
```

Glossary

acquire a device

To reserve a particular device for exclusive use. Application programs can acquire a device only when that device is not acquired by any other programs.

address translation unit

A hardware function that handles virtual–memory address translation operations in Domain system nodes. See also **memory management unit**.

asynchronous fault

A fault that is unrelated to program or hardware action. Asynchronous faults include the quit fault, which is generated when you type CTRL/Q to exit from a program, and the process stop fault, generated when you log out. See also **fault**.

bus

A network of signal routes through which device controllers and the processor address one another and pass data; one of the buses that we currently support (that is, MULTIBUS, VMEbus, and PC AT compatible bus).

bus master

The hardware component that currently controls the bus. When a controller acquires the bus, it becomes bus master.

bus slave

The hardware component that decodes addresses and acts on commands from the bus master.

byte swapping

Rearranging the left and right bytes of a word to compensate for the difference between the way our processor orders bytes and the way a controller does.

call side

The set of routines and procedures within a device driver that programs actively call to perform operations. A device driver's call side is bound separately from its interrupt side. See also **interrupt side.**

cleanup routine

The device driver routine called during device release to ensure that no I/O is in progress and that the device will not generate further interrupts. The cleanup routine is a call–side routine.

control and status register (CSR)

A control and status register for a device or controller. Control and status registers are located in bus I/O space.

CSR

See **control and status register.**

CSR page

A page of bus I/O space that contains the control and status registers for a particular device or controller. A device or controller's CSR page is loaded into user–process address space when the device is acquired.

data structure

Any table, list, queue, or array whose format and access conventions are well defined for reference by one or more programs.

DDF

See **device descriptor file.**

device

One drive and its controlling logic (for example, a storage module device). In this document, the terms device and controller are synonymous.

device descriptor file (DDF)

A data structure that describes the device to the system. Each device has one associated DDF.

device driver

The set of user–written routines and procedures that handle I/O operations to and from a peripheral device. The device driver is composed of a call side and an interrupt side, bound in separate modules.

device interrupt

A signal sent to the processor by a peripheral device through an interrupt request line.

direct memory access (DMA)

A type of I/O transfer where a device transfers data directly to processor memory.

DMA

See **direct memory access.**

DMA controller

A controller that performs direct memory access I/O transfers.

DMA overrun

A condition in which a device cannot transfer data to the processor as fast as it is receiving it, and so loses data.

eventcount

A 32–bit integer that processes establish to count the occurrence of an event or events. The eventcount is the primary method of interprocess synchronization.

fault

A fatal error from which a program cannot recover.

fault handler

The routine that performs cleanup services after a fault occurs and before the program exits. Both application programs and device drivers can contain fault handling routines.

general purpose input/output (GPIO) software

The set of routines and commands that application programs and device drivers use to perform I/O operations on a peripheral device.

hard–wired memory

Device data structures or CSRs that are located at preset fixed addresses.

initialization routine

The device driver routine that readies a device for I/O operations. The initialization routine is a call–side routine.

interrupt

See **device interrupt.**

interrupt mask register

A register that determines whether or not the processor will receive an interrupt from a given device. Each bit within the register corresponds to an interrupt line. When clear, the process can receive interrupt requests on the line; when set, the processor does not receive the request. See also **interrupt request line.**

interrupt request line

Lines that devices use to generate interrupt requests to the processor.

interrupt routine

The device driver routine that performs device–specific interrupt processing. The interrupt routine is part of the driver's interrupt side.

interrupt side

The part of a device driver that is called by the System Interrupt Handler in response to an interrupt condition. The interrupt side is composed of one or more user–written interrupt routines and data.

interrupt stack

Wired memory that contains scratch storage, saved registers, and subroutine addresses used by a device driver. The default interrupt stack size is 1024 bytes (one page).

interrupt vector

The address generated that identifies an interrupting device to the processor.

I/O map

A data structure used to map MULTIBUS or PC AT compatible bus memory to processor memory. Each entry within the I/O map maps one page of MULTIBUS or PC AT compatible bus memory to processor memory.

I/O space

The region of the bus address space that contains device CSRs.

iova

A virtual address that is mapped into the physical address space of any of the buses that we support.

mapping an I/O buffer

The process by which a device driver establishes an association between pages of MULTIBUS or PC AT compatible bus memory and the pages of a buffer within process address space.

memory management unit (MMU)

The hardware component that handles virtual memory translation operations within Domain system nodes. Also called the **Address Translation Unit**.

memory–mapped controller

A controller that contains on–board memory in which it stores data from external devices.

memory–mapped I/O

Data transfers to and from the local memory of memory–mapped controllers. Device drivers must map the local memory to virtual address space before they can read and write to it.

memory space

The region of the bus address space that contains memory locations.

MMU

See **memory management unit.**

nonbus–vectored interrupt

A type of interrupt where the device raises its interrupt request line, but does not send an interrupt vector over the bus. See also **interrupt vector**.

offset

A fixed displacement from the beginning of a data structure.

page

1024 bytes; the unit of measure in our systems.

paging

Moving pages of virtual memory to and from physical memory. The MMU controls paging operations.

PBU

Peripheral bus unit, synonymous with MULTIBUS, PC AT compatible bus, or VMEbus device.

PBU Manager

The collection of routines that are internal to the operating system and manage GPIO resources.

peripheral interrupt controller (PIC)

The hardware component that arbitrates interrupt requests sent by devices along their interrupt request lines.

PIC

See **peripheral interrupt controller**.

processor memory

The main memory of a Domain node.

programmed I/O

Data transfers of single words or bytes through CSRs.

scatter–gather

Contiguous disk transfer to and/or from discontiguous pages of memory.

serial priority resolution

A method of bus arbitration where position in the card cage determines a controller's bus request arbitration priority level.

synchronous fault

A fault that occurs as a result of program or hardware errors, such as floating–point overflow or disk errors. See also **asynchronous fault, fault**.

system interrupt handler

The part of the operating system that processes device interrupts.

user–process address space

The area of virtual address space in which a process executes. When a device is acquired, its device driver, CSR page, and other I/O data structures are loaded into user–process address space.

virtual address

The 32–bit integer that identifies a “location” in virtual address space. The MMU translates virtual addresses to physical addresses.

virtual address space

The set of all possible virtual addresses that a program executing within a process can use to identify the location of an instruction or data.

wired memory

One or more pages of virtual address space that are made permanently resident in processor memory and therefore cannot be paged out by the MMU.

wiring a buffer

Making the pages of a buffer ineligible for virtual memory paging operations. Device drivers must wire the pages of an I/O buffer before initiating a DMA transfer.

Index

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Writing Device Drivers with GPIO Calls*
Order No.: *000959-A00*
Date of Publication: *July 1988*

What type of user are you?

_____ System programmer; language ____________	
_____ Applications programmer; language ____________	
_____ System maintenance person	_____ Manager/Professional
_____ System Administrator	_____ Technical Professional
_____ Student Programmer	_____ Novice
_____ Other	

How often do you use the Domain system?______________________

What parts of the manual are especially useful for the job you are doing?______________________

What additional information would you like the manual to include?______________________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)______________________

____
Your Name Date

____
Organization

____
Street Address

____
City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

OLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

OLD

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Writing Device Drivers with GPIO Calls*
Order No.: *000959-A00*
Date of Publication: *July 1988*

What type of user are you?

_____ System programmer; language _______________	
_____ Applications programmer; language _______________	
_____ System maintenance person	_____ Manager/Professional
_____ System Administrator	_____ Technical Professional
_____ Student Programmer	_____ Novice
_____ Other	

How often do you use the Domain system?_______________

What parts of the manual are especially useful for the job you are doing?_______________

What additional information would you like the manual to include?_______________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)_______________

Your Name Date

Organization

Street Address

City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

OLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

›LD

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Writing Device Drivers with GPIO Calls*
Order No.: *000959-A00*
Date of Publication: *July 1988*

What type of user are you?

_____ System programmer; language ______________________
_____ Applications programmer; language ____________________
_____ System maintenance person _____ Manager/Professional
_____ System Administrator _____ Technical Professional
_____ Student Programmer _____ Novice
_____ Other

How often do you use the Domain system?____________________________________

What parts of the manual are especially useful for the job you are doing?____________________

What additional information would you like the manual to include?____________________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)____________________

____
Your Name Date

____
Organization

____
Street Address

____
City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

OLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

)LD