Writing Device Drivers with GPI/O Calls

Order No. 000959
Revision 10

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: November 1981
Latest Printing: June 1987

This document was produced using the Interleaf Workstation Publishing Software (WPS). Interleaf and WPS are trademarks of Interleaf, Inc.

Preface

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

Audience

This manual is intended for programmers who must write drivers for devices that we do 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.

We've organized this manual as follows:

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 that will help you to write drivers for VME devices.
Chapter 3	Describes the AT–compatible bus and its interface with our system that will help you to write drivers for AT devices.
Chapter 4	Gives an overview of the major components of I/O software—i.e., the application, GPI/O software, and the device 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 shared 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.
Appendix A	Describes the GPI/O commands that the user invokes to run the driver.

Appendix B	Describes the calling format and parameters of the GPI/O routines.
Appendix C	Provides some tips on setting up the CSR page and using datatypes in C.
Appendix D	Provides performance and timing information that relates to driver execution on our operating system.
Appendix E	Gives a program listing of a device driver coded in Pascal.
Appendix F	Gives a program listing of a device driver coded in C.

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

Summary of Technical Changes

This manual [formerly *General Purpose Input/Output (GPIO) Reference*] has been revised for Revision 10. New technical information to support Software Release 9.6 includes infomation on writing GPI/O drivers for VME devices.

Related Manuals

The *Domain Pascal Language Reference*, Order No. 000792, describes our implementation of the Pascal language. Appendix C lists our extensions to Standard Pascal.

The *Domain C Language Reference*, Order No. 002093, and *DOMAIN C Library (CLIB) Reference*, Order No. 005805, describe our implementation of the C language.

The *Domain System Call Reference*, Order No. 007196, describes the calling syntax for the system services that your driver can call.

Programming with General System Calls, Order No. 005506 describes the general purpose Domain system calls that you can use to perform system services for your driver.

The *Domain System Command Reference*, Order No. 002547, describes the command environment as well as the function and format of the commands that users can invoke.

The *Domain Binder and Librarian Reference,* Order No. 004977, describes how to use the Domain binder to combine several object modules (e.g., a call library and an interrupt library) into one executable object module.

The *Domain Language Level Debugger Reference*, Order No. 001525, describes how to use DEBUG.

Installing Input/Output (I/O) Devices for DOMAIN Nodes describes the hardware requirements for attaching peripheral devices to the DOMAIN system bus.

DN570-T, DN580-T, DN590-T, and DSP500-T Technical Reference Manual, Order No. 009491, describes our implementation of the VMEbus.

The *Domain Series 3000 Technical Reference*, Order No. 008778, and the *Domain Series 3000 Hardware Architecture Handbook*, Order No. 007861, describe our implementation of the AT-compatible bus.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels, you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. Refer to the CRUCR (CREATE_USER_CHANGE_REQUEST) shell command description. You can view the same description online by typing:

```
$ HELP CRUCR <RETURN>
```

For your documentation comments, we've included a Reader's Response form at the back of each manual.

Documentation Conventions

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

UPPERCASE	Bold, uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally.
lowercase	Bold, lowercase words or characters in formats and command descriptions represent values that you must supply.
example	Bold words in command examples represent literal user keyboard input.
`output`	Typewriter font words in command examples represent literal system output.
[ ]	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/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down <CTRL> while typing the character.
...	Horizontal ellipsis points indicate that the preceding item can be repeated one or more times.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

For your documentation comments, we've included a Reader's Response form at the back of each manual.

Contents

Part III Reference Information

Illustrations

Tables

Chapter 1

I/O Bus Structures: The MULTIBUS

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 the I/O structure of a computer system. 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.

Chapters 1, 2, and 3 describe what you need to know about each of the three I/O bus structures that we support—the Intel MULTIBUS*, the VMEbus, and the IBM AT–compatible bus—in order to use the General Purpose Input/Output (GPI/O) software package to write device drivers for the particular I/O bus implemented on your node.

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 *IEEE Standard Microcomputer System Bus* (IEEE–796 specification).

*MULTIBUS is a trademark of the Intel Corporation.

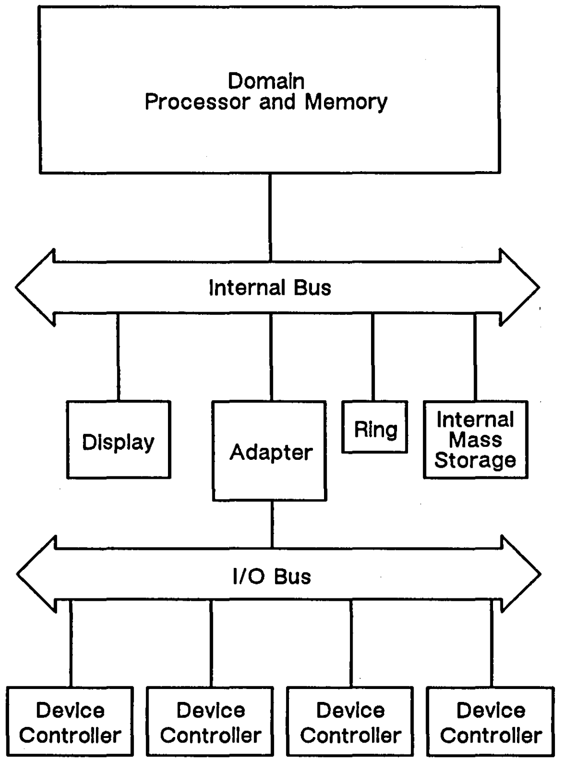


Figure 1-1. Relationship between Domain Node and Peripheral Controllers

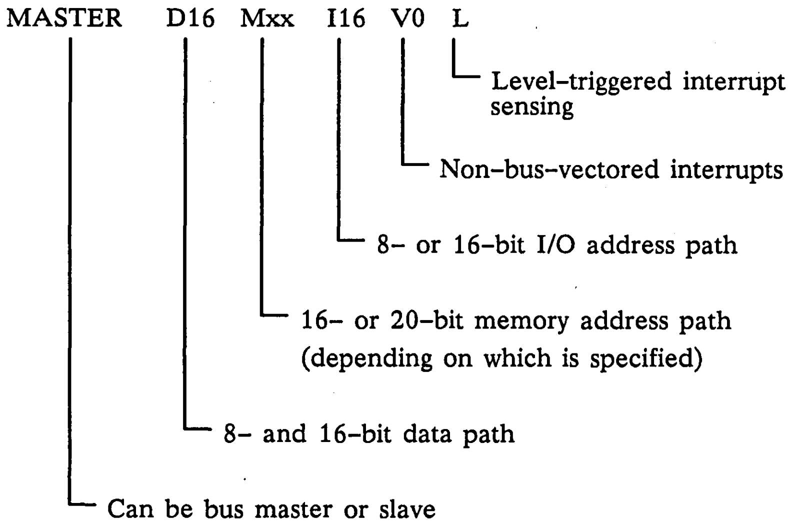
1.1 MULTIBUS Compliance Levels

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 see 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 it. 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
DN4xx/DN6xx, DSP160 Family	16-bit MULTIBUS, serial arbitration priority	MASTER D16 M16 I16 V0 L
DSP80 Family, DSP90	20-bit MULTIBUS, parallel arbitration priority	MASTER D16 M20 I16 V0 L
DN550, DN560, DN570-T, DN580-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 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 (i.e., DSP160, DN420, DN460, DN600, and DN660) allow both the central processor and any attached controller to act as bus masters. When the processor is bus master, it can address 32K bytes of MULTIBUS I/O space and 32K bytes 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 (i.e., DSP80, DSP80A, DSP90, DN550, and DN560) also allow either processor or controllers to act as bus masters. When the processor is bus master, it can address 64K bytes of MULTIBUS I/O space and 1M byte 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 1M byte of address space is available.

> **NOTE:** Although the full 64K bytes 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 16K bytes of MULTIBUS I/O space; refer to section 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 3M bytes of memory, the upper half (512K bytes) 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 non-bus-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 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.

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 GPI/O device acquisition routine, PBU_$ACQUIRE, (or the AQDEV command) 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.

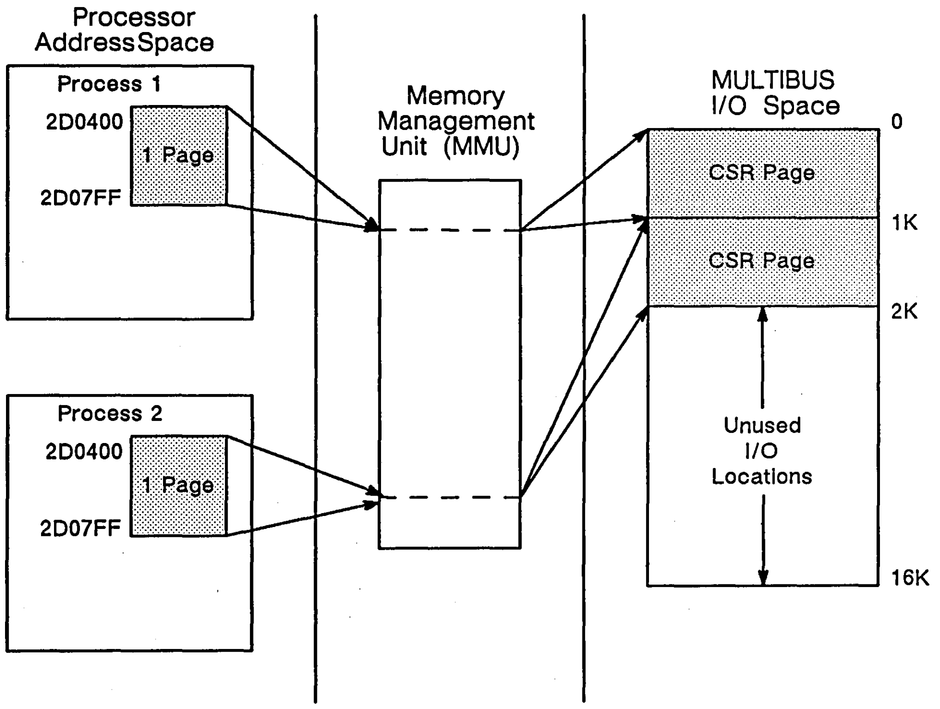


Figure 1-2. Mapping CSR Pages to MULTIBUS I/O Space

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 GPI/O routines. These routines map areas of processor address space and particular sections of MULTIBUS memory space. Device drivers next call the GPI/O 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.

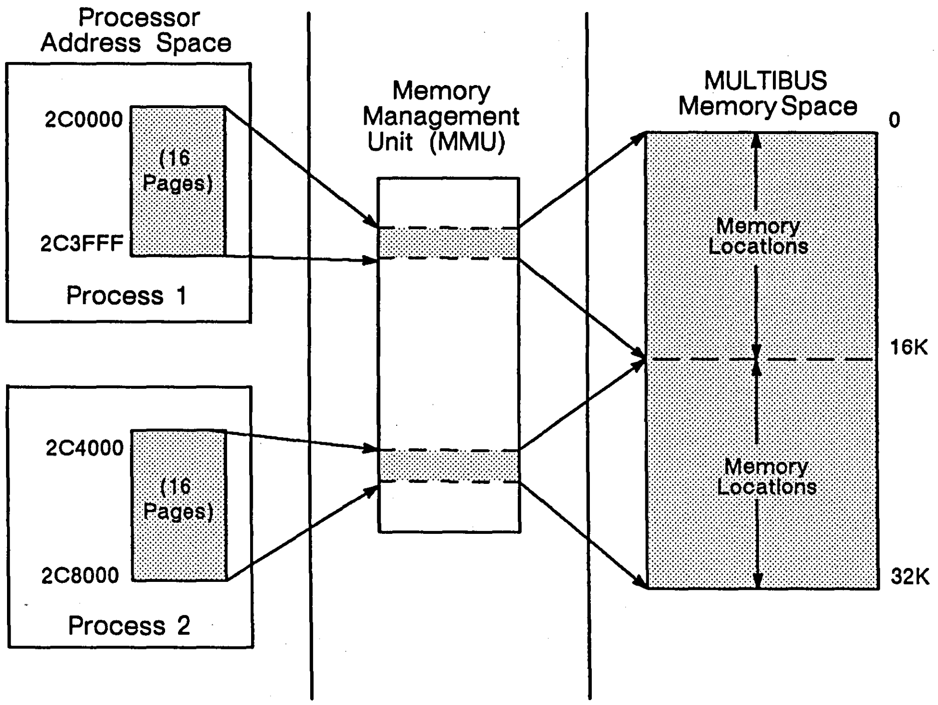


Figure 1-3. Mapping Processor Address Space to MULTIBUS Memory Space

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 that (1) the controller is the bus master, (2) address translation proceeds from the MULTIBUS to the processor, and (3) 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 GPI/O routines to allocate I/O map entries for the DMA. Chapter 7, section 7.1 describes these GPI/O 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.

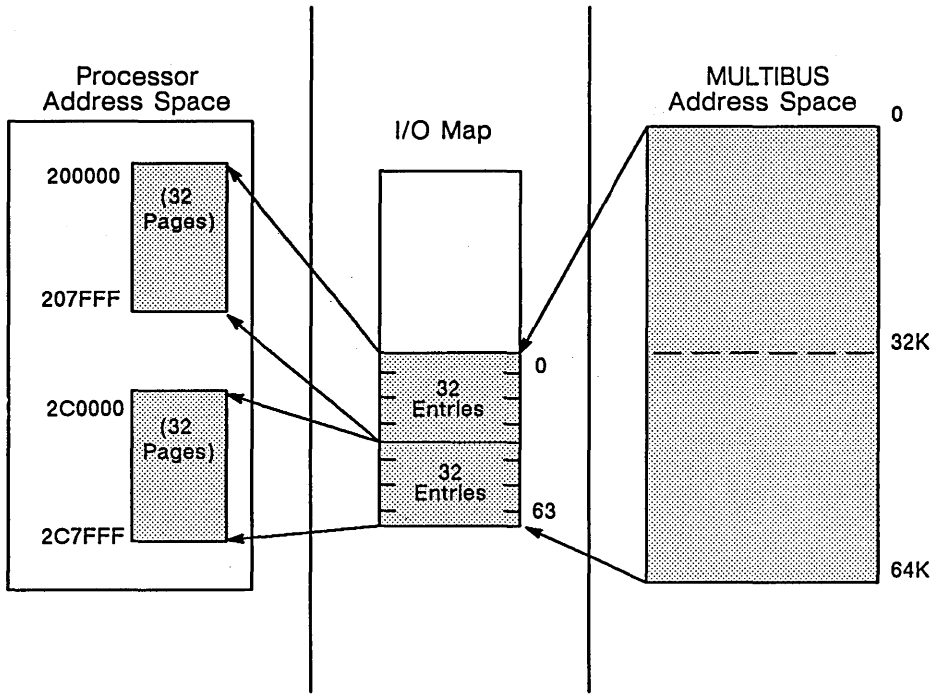


Figure 1-4. Mapping MULTIBUS Address Space to Processor Address Space

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

Software	Addresses Used
Domain/ComController	Memory pages 4000 to 7F00 and I/O page 0800 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 (i.e., 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 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 always in use.

*ETHERNET is a registered trademark of the Xerox Corporation.
**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.

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 we supply. 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 lie 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 to 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 to 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 lie on even-byte addresses.

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

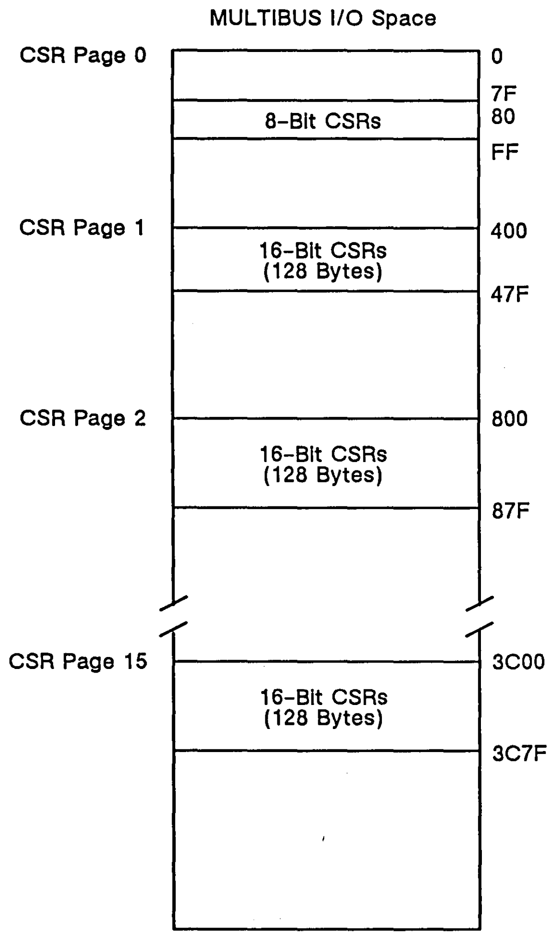


Figure 1–5. 8-Bit Controller CSR Assignment

Configuring Controller Memory

Drivers call GPI/O 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 lie completely in the first 32K bytes (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 (refer to Chapter 7, section 7.2.1).

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.

Using Interrupt Request Lines

Of the eight interrupt request lines available on the MULTIBUS, the highest priority 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

Since 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.

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, since 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 lie on even-byte addresses. If an 8-bit controller is present in your configuration, assign its CSRs according to the rules outlined in section 1.3.1.

Configuring Controller Memory

If a node with a 20-bit MULTIBUS is fully configured with 3M bytes of memory, the upper half (512K bytes) 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.

Configuring Controller Address Lines

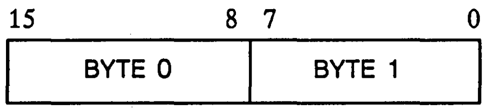
Nodes with a 20-bit MULTIBUS implementation can map up to 1024 pages of MULTIBUS address space through the I/O map to processor memory. As in 16-bit MULTIBUS systems, controller references to MULTIBUS addresses above 1M byte are wrapped. Consequently, when you have the choice of configuring a controller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

Using Interrupt Request Lines

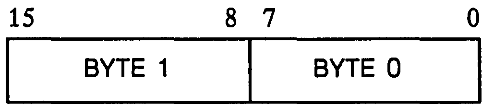
Nodes with a 20-bit MULTIBUS allocate interrupt request lines in the same way as nodes with a 16-bit MULTIBUS, except that lines 6 and 7 are also available (although they are reserved for Domain system-supplied devices). Again, the processor is solely responsible for acknowledging peripheral device interrupt 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

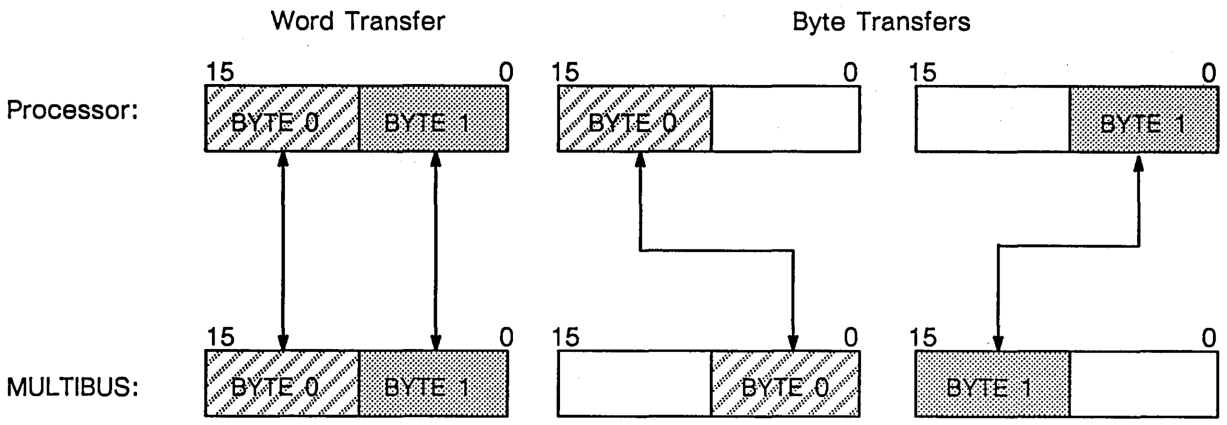
The necessity for byte swapping (exchanging the left and right bytes of a word) arises from the fact that the Domain processor, which is based on the Motorola 68000 family, orders bytes within a word the opposite of the way Intel processors order them on MULTIBUS controllers. This is how our processor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



and this is how the MULTIBUS does it:

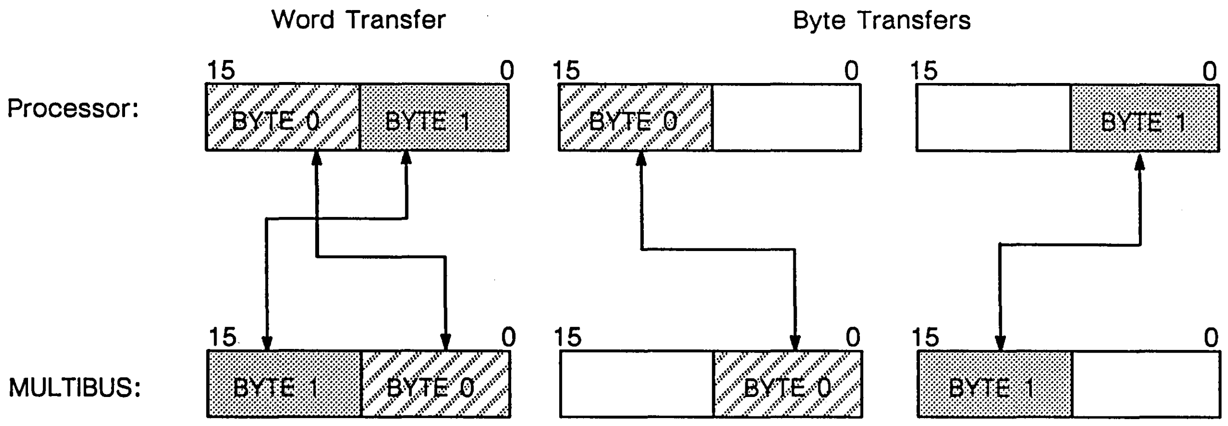


We deal with this incompatibility by swapping bytes in hardware during byte transfers. Effectively, then, character strings copied as bytes and integers copied as words are preserved, but character strings copied as words (and words copied as bytes) are byte swapped. The following illustrates this strategy:

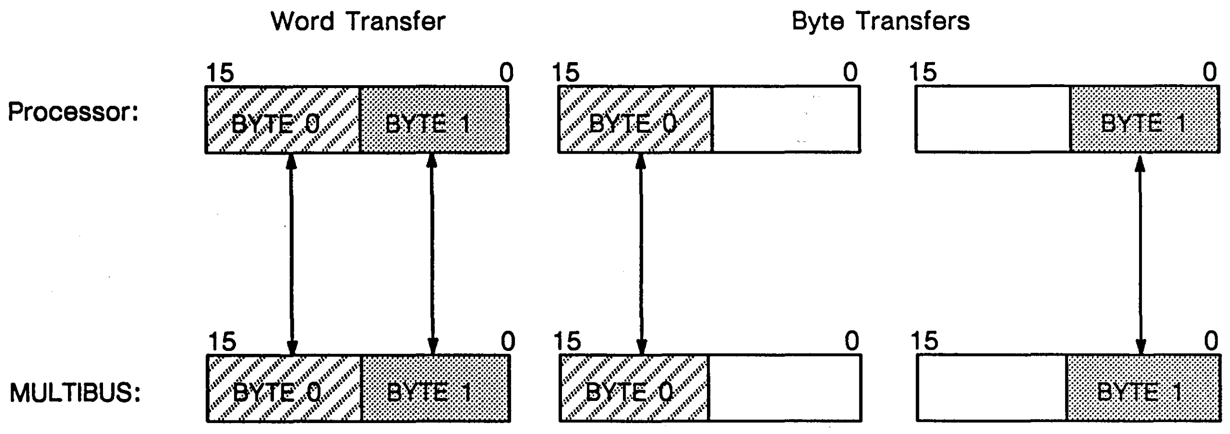


Pointers to words must be even. Pointers to processor left bytes (byte 0) must be even; pointers to processor right bytes (byte 1) must be odd.

The GPI/O 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 spelled out above (the pbu_swap_off option gives you the arrangement described above). By specifying the pbu_swap_words option with this call, you ensures 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 GPI/O software to write device drivers for VME 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 *DN570-T, DN580-T, DN590-T, and DSP500-T Technical Reference Manual* and the *Motorola VMEbus Specification Manual,* Rev. C.1 or IEEE P1014/D1.2.

2.1 Address Space Allocation

Since there is no mapping mechanism between the VMEbus and a customer VME device, there must be agreement as to what VME 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 DN570-T and DN580-T (currently the only released VME-based machines) is listed in Table 2-1.

Table 2-1. Address Space Allocated for DN570/580-T VME Devices

Physical Addresses	Resource	Address/Data Lines
0000-7FFF	VME CSRs	16-Bit Addressing 16-Bit Data Path
C000-DFFF	VME CSRs	24-Bit Addressing 16-Bit Data Path
80000-FFFFF	User VME	24-Bit Addressing 16-Bit Data Path
200000-2FFFFF	User VME	24-Bit Addressing 16-Bit Data Path
310000-3FFFFF	User VME	24-Bit Addressing 16-Bit Data Path
600000-7FFFFF	User VME	24-Bit Addressing 32-Bit Data Path
800000-FFFFFF	User VME*	24-Bit Addressing 32-Bit Data Path

*Available only on DN570-T workstations.

2.2 Bus Grant Level

VME devices should use bus grant level 2.

2.3 Address Modifiers

The current DN570-T and DN580-T VME interface defines the following address modifiers for all references to VME 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.

We recommend 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 VME devices are currently assigned to VME interrupt level 5. The VME 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 VME controller presents a status/ID byte during a VME interrupt acknowledge cycle. The operating system uses this byte to distinguish between multiple VME devices and by GPI/O as the unit number identifying the device. Status/ID bytes F8 through FE (coressponding 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 VME device defines the bottom nibble of the status/ID as the device unit number.

2.6 Software Considerations

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

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

- PBU[2]_$ALLOCATE_MAP
- PBU[2]_$FREE_MAP
- PBU[2]_$MAP
- PBU[2]_$UNMAP

In addition, the following GPI/O calls are not applicable to VME devices and cannot be used in drivers for VME devices:

- PBU_$DEVICE_INTERRUPTING
- PBU_$DISABLE_DEVICE
- PBU_$ENABLE_DEVICE
- PBU_$CONTROL
- PBU_$DMA_START
- PBU_$DMA_STOP

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

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_$WIRE_SPECIAL (instead of PBU_$WIRE) to wire buffers for DMA operations. This call returns a list of physical (i.e., VME) 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 VME Device

To create a DDF for a VME device, you must specify the -VME option with the CRDDF command. This option indicates to GPI/O 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, section 11.3.2 for an example of the CRDDF command with the -VME option.

Chapter 3

I/O Bus Structures: The IBM AT–Compatible Bus

This chapter presents information you need to know about the IBM AT–compatible bus in order to use GPI/O software to write device drivers for AT–compatible devices—specifically, I/O address and memory space allocation, unit numbering, bus timeout, DMA and interrupt lines, byte swapping, and software considerations. For additional information about the AT–compatible bus, refer to the *Domain Series 3000 Technical Reference*.

3.1 AT–Compatible Address Space

The physical address space on the AT–compatible bus that is available to the user consists of I/O address space, which is reserved for device CSRs, and memory address space, which is reserved for memory-mapped controllers. The following subsections describe these two address spaces in detail. For additional information on the AT–compatible bus address space, refer to the *Domain Series 3000 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
1A8–210	Customer Devices
218–21F	Tape Controller
220–23F	Domain Ring Controller
240–2F7	Customer Devices
2F8–2FF	Serial–Parallel Expansion (SPE) option—Serial Line 2
300–310	Ethernet Controller
320–33F	Domain Ring Controller
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 AT–compatible I/O address space are mapped differently from addresses in MULTIBUS and VME address space. Ten-bit consecutive addresses in the I/O address space are mapped into processor address space in groups of eight bytes: each group is assigned the first eight bytes of a different, but consecutive, page (1024 bytes). Thus, the first 1024 addresses in AT–compatible address space (0–3FF) map to 128 physical pages (40000–5FFFF) in processor address space.

An AT–compatible controller using CSR addresses 200 through 217 might have the following type declaration:

```
TYPE csr_page_t = [device] packed record
      first_eight : array[0..7] of char;
      next_eight  : array[0..7] of char;
      last_eight  : array[0..7] of char;
      end;
```

But in our system, the type declaration should be:

```
TYPE csr_page_t = [device] packed record
      first_eight : array[0..7] of char;
      pad1        : array[8..bytes_per_page-1] of char;
      next_eight  : array[0..7] of char;
      pad2        : array[8..bytes_per_page-1] of char;
      last_eight  : array[0..7] of char;
      pad3        : array[8..bytes_per_page-1] of char;
      end;
```

Figure 3–1 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 in driver address space).

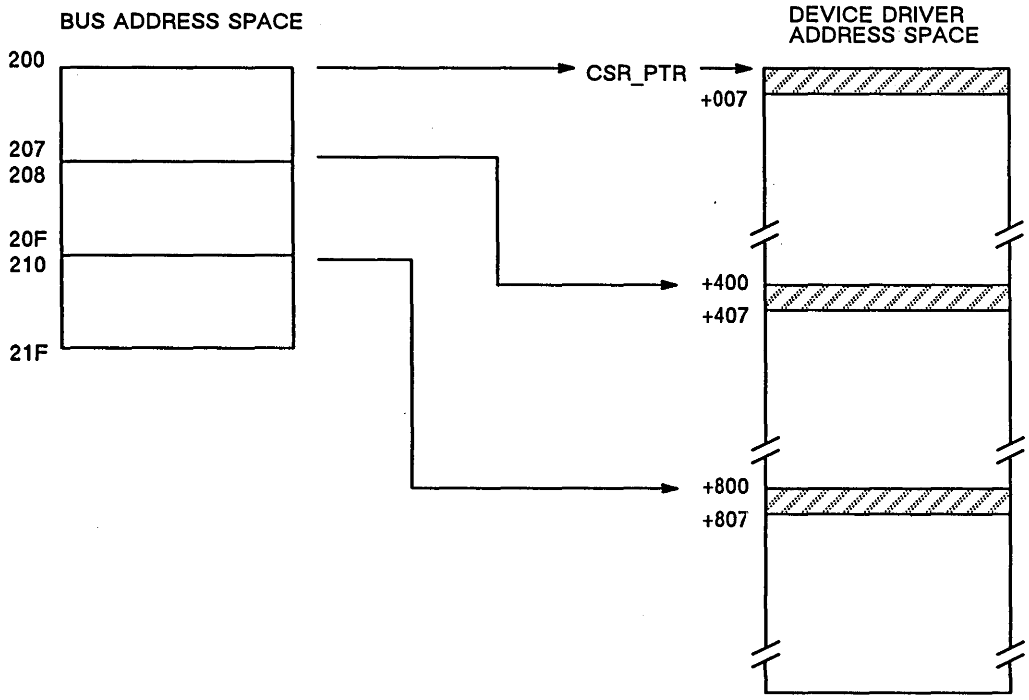


Figure 3-1. CSR Mapping Scheme for AT-Compatible Devices

Sixteen-bit addresses (i.e., so-called AT addresses, which are not supported on the PC bus) extend the address range beyond the 1K byte (0-3FF) range of 10-bit addresses up to 64K bytes (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 3-2 shows how the 16 bits of an AT-compatible I/O address are translated to a processor physical address.

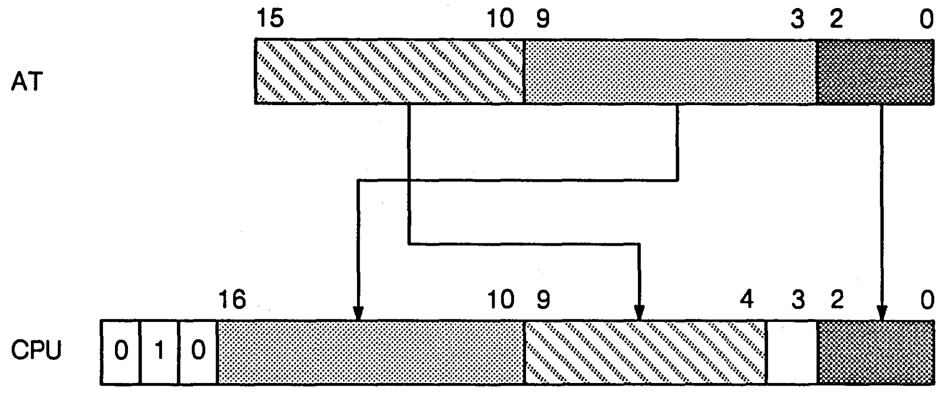


Figure 3-2. Mapping a 16-Bit AT Address to Processor Address Space

We provide a utility, CVT_AT (CONVERT_AT_ADDRESSES), that translates any AT-compatible I/O address (10- or 16-bit) into the processor physical address to which it is mapped. The command's syntax and usage are fully described in Appendix A.

3.1.2 Memory Space

The AT-compatible memory space is used for memory-mapped controllers. Addresses are mapped one-to-one to processor physical address space. Controllers are mapped and unmapped using GPI/O routines PBU2_$MAP_CONTROLLER and PBU2_$UNMAP_CONTROLLER.

Table 3-2 lists the address ranges within the memory space 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 Series 3000 Hardware Architecture Handbook.*

Table 3-2. 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)
060000-09FFFF	Available for Customer Devices
0A0000-0BFFFF	Color or Alternate Monochrome Graphics
0C0000-0DFFFF	Alternate Monochrome Graphics
0E0000-0FFFFF	Alternate Color Graphics
100000-8FFFFF	Main Memory
900000-BFFFFF	Available for Customer Devices
C00000-CFFFFF	PC Co-processor
D00000-DFFFFF	PC Co-processor Alternate
E00000-F9FFFF	Available for Customer Devices
FA0000-FDFFFF	Monochrome Graphics
FE0000-FFFFFF	Available for Customer Devices

3.2 Unit Numbering

The unit number of an AT-compatible device is identical with the Interrupt Request (IRQ) line. There are 16 possible unit numbers, 0 being the highest. But 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.

Table 3-3. Allocation of Unit Numbers

Unit No. and IRQ	Interrupt Priority	Device
0*	1 ✓	Timer
1*	2 ✓	Keyboard
2*		Reserved
3	3 ✓	Domain Ring Controller
4	12	SPE—Serial Line 1 or User Device
5	13	Tape Controller
6	14	Disk Controller or User Device
7	15	User Device
8*	4 ✓	Calendar
9	5 ✓	Ethernet 2, SPE—Serial Line 2, or User Device
10	6	Ethernet 1 or User Device
11	7	PC Co-processor or User Device
12	8	User Device
13*	9	Reserved
14	10	Disk Controller
15	11	PC Co-processor 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 AT–compatible bus does *not* generate bus timeouts. Therefore, you cannot use the GPI/O 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

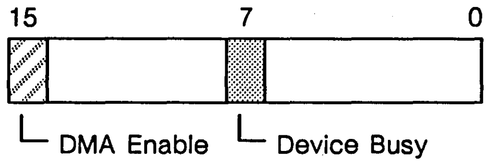
DMA and IRQ lines typically float on AT–compatible controllers. Refer to the device documentation for specific information on enabling these lines. Generally, however, you should do the following:

- Call PBU_$DMA_START *after* enabling the DMA lines and PBU_$DMA_STOP *before* disabling them (refer to Appendix B for information concerning these GPI/O 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_$DMA_START followed by the data transfer command to the device in your driver.
- Call PBU_$ENABLE_DEVICE *after* you have set up the controller to have some interrupts enabled. Likewise, you should call PBU_$DISABLE_DEVICE *before* you clear all interrupt enables from the controller. Refer to Appendix B for more information concerning these GPI/O calls.

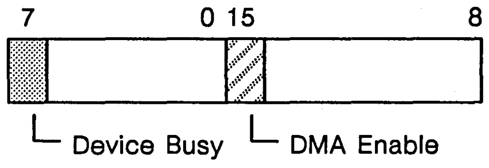
3.5 Byte Swapping

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 an 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 AT–compatible controller:



would look like this on the processor:



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

Figure 3–3 illustrates byte swapping between the processor and the AT–compatible bus.

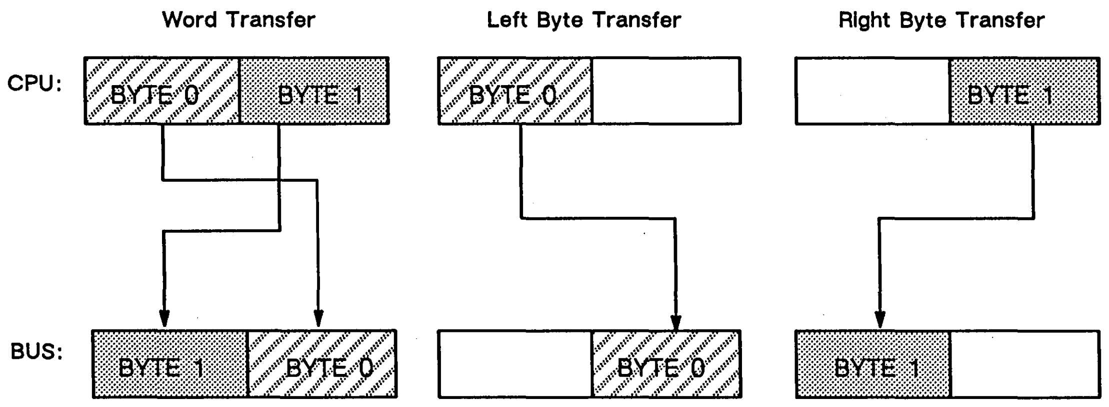


Figure 3-3. Byte Swapping between Processor and AT-Compatible Bus

3.6 Software Considerations

GPI/O software supports four kinds of I/O operations on the AT-compatible bus:

- Memory-Mapped I/O.
- Programmed I/O.
- DMA: The processor has the DMA hardware.
- Demand-DMA: The controller has its own DMA hardware and can request external bus mastership.

Section 3.6.1 describes the GPI/O routines that drivers can call to perform these operations.

There is no DMA address translation hardware (i.e., I/O map) for the AT-compatible bus; the following GPI/O calls are, therefore, not applicable to drivers that support AT-compatible devices:

- PBU[2]_$ALLOCATE_MAP
- PBU[2]_$FREE_MAP
- PBU[2]_$MAP
- PBU[2]_$UNMAP

Otherwise, you use GPI/O software when writing drivers for AT-compatible devices just as you would for MULTIBUS devices.

3.6.1 GPI/O Calls for AT-Compatible Devices

Three GPI/O calls are specially designed for use with AT-compatible devices:

- PBU_$WIRE_SPECIAL
- PBU_$DMA_START
- PBU_$DMA_STOP

The following paragraphs briefly describe when and how to use these calls. For a full description of the calls, refer to Appendix B.

Wiring for DMA: PBU_$WIRE_SPECIAL

Since there is no mapping hardware between the customer's device and the AT–compatible bus, drivers for AT–compatible devices should call PBU_$WIRE_SPECIAL (instead of PBU_$WIRE) to wire buffers for DMA operations *when the controller has demand–DMA capability* (refer to the next paragraph for DMA operations with controllers that do not have on–board DMA hardware). This call returns a list of physical addresses at which the buffer is located. The customer's driver or controller hardware uses the addresses to perform the necessary scatter–gather operations.

Use PBU2_$UNWIRE to unwire buffers that have been wired with PBU_$WIRE_SPECIAL.

Starting and Stopping a DMA Operation

Performing a DMA operation with an AT–compatible device that does not have demand–DMA capability (i.e., cannot request to become an external bus master) requires two GPI/O routines: PBU_$DMA_START and PBU_$DMA_STOP. These are paired functions that must surround each DMA operation, whether successful or not. PBU_$DMA_START prepares DMA hardware for the controller's operation. After the driver calls PBU_$DMA_START, the controller can begin its operation. When the controller indicates that the operation has completed, the driver next calls PBU_$DMA_STOP to get status from DMA hardware to ensure that the hardware has completed its share of the operation as well. The driver must call PBU_$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.

NOTE: Data transferred in one DMA operation must not exceed 1K byte and must not cross page boundaries.

Use PBU2_$WIRE and PBU2_$UNWIRE to wire buffers that are to be transferred via PBU_$DMA_START and PBU_$DMA_STOP, just as you would with MULTIBUS devices.

3.6.2 Creating a DDF for an AT–Compatible Device

To create a Device Descriptor File (DDF) for an AT–compatible device, you must specify the –AT option with the CRDDF command. This option indicates to GPI/O software that the device in question resides on the 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 are in the range 0–15, excepting those assigned to Domain system–supplied devices (refer to section 3.2 and Table 3–3).

The –DMA_CHANNEL option can be used with AT–compatible devices to specify the DMA channel number that a controller will use. This option is provided only as a means of passing the channel number to the driver; GPI/O software makes no use of this field.

Refer to Appendix A for a full description of the CRDDF command and the –AT option and to Chapter 11, section 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)
- General Purpose I/O (GPI/O) routines and commands (supplied by us)
- Device driver routines (user written)

The following sections 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 GPI/O routines and commands.

The last section of this chapter provides a driver component checklist for your use when writing a driver.

4.1 The 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 GPI/O Commands and Routines

The *GPI/O commands and routines* create the environment in which a device driver runs. They control the acquisition and release of the device, create and delete the mapping between a device's memory or registers and processor address space, and set up the mechanisms to facilitate data transfers to and from a device. Table 4–1 lists the files associated with GPI/O software product. The individual commands are fully described in Appendix A, the routines in Appendix B.

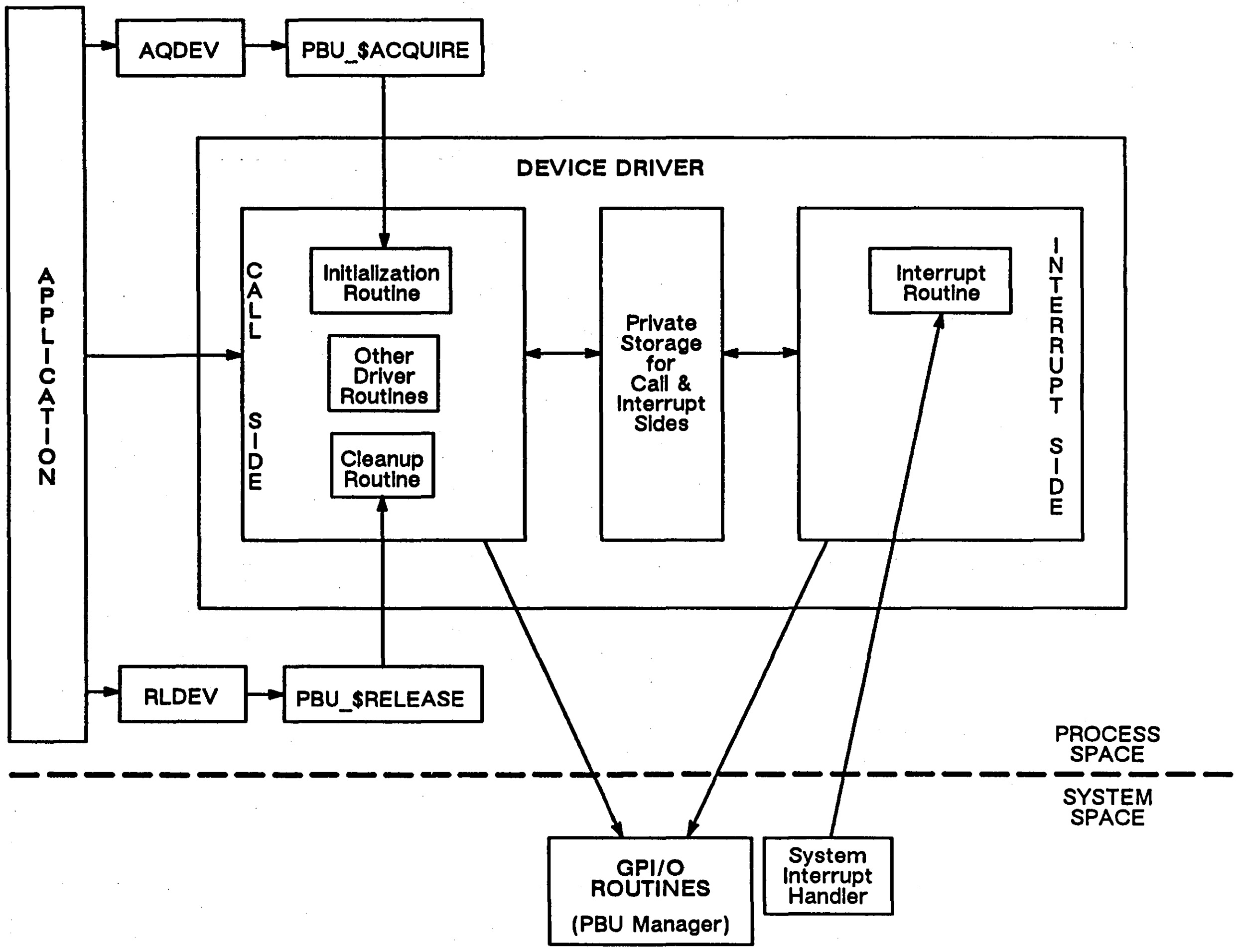


Figure 4-1. Interaction of I/O Software

Table 4-1. GPI/O Software

File	Contents
/lib/pbu_int_lib	Library to be bound with user-written interrupt routine(s)
/lib/pbulib	GPI/O routines and interface to internal GPI/O manager, automatically installed at system startup
/com/aqdev	AQDEV (ACQUIRE_DEVICE) command
/com/rldev	RLDEV (RELEASE_DEVICE) command
/com/crddf	CRDDF (CREATE_DDF) command
/com/cvt_at	CVT_AT (CONVERT_AT_ADDRESSES) command
/sys/ins/pbu.ins.pas	Insert file for Pascal programs using GPI/O routines
/sys/ins/pbu.ins.c	Insert file for C programs using GPI/O routines
/sys/help/syscalls/pbu.hlp	Help file for GPI/O routines
/sys/help/pbu.hlp	Command index to GPI/O 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
/domain_examples/gpio_examples	Directory containing sample drivers

4.3 The 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.

4.3.1 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.

4.3.2 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, section 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, section 8.2.1.
- A clean-up 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 further interrupts. You write the clean-up routine according to the calling sequence specified 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 pre-processing 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 timeout 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, of course, on the requirements of the device and the application. To help you with the design of your driver, Part II of this book describes the driver components in detail and explains how to construct them by using GPI/O routines. Part III provides reference information, such as the format and syntax of GPI/O 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, in subdirectories BM_EXAMPLE and BM_EXAMPLE.C (see also the program listings in Appendixes E [Pascal] and F [C])
- A device driver for an Interlan controller, in subdirectory INTERLAN_EXAMPLE
- A device driver for a 3COM controller, in subdirectory THREECOM_EXAMPLE
- A shared driver for the SPE board, in subdirectory SHARED_EXAMPLE
- A device driver for an AT-compatible device, in subdirectory AT_EXAMPLE

To make the device driver accessible to user programs, you must bind the routines as described in Chapter 10, section 10.1. 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 GPI/O 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.

4.3.3 The 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 step you through 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 GPI/O 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 GPI/O routines. A slightly reorganized version of the BM_EXAMPLE driver appears in Appendixes E (Pascal) and F (C).

Note that names of driver routines begin with *BM* (Bulk Memory), whereas names of GPI/O routines all begin with *PBU* (Peripheral Bus Unit). Also, names of driver routines that do not include a dollar ($) sign (e.g., BM_COMMAND) are internal subroutines that are not referenced outside the module in which they are defined.

Initialization

After the device has been acquired, the *PBU Manager* (a collection of routines that are internal to the operating system and manage GPI/O 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.

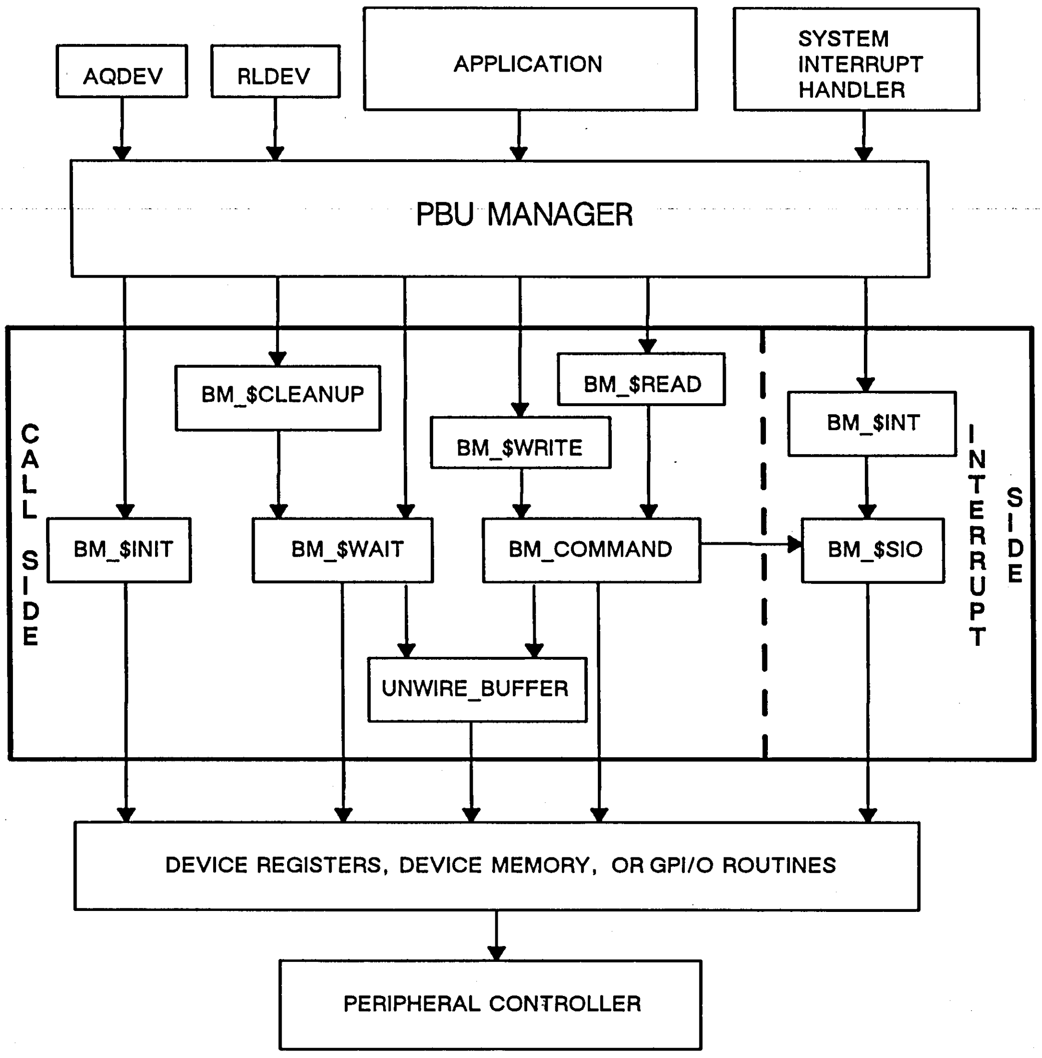


Figure 4-2. Driver Routines in BM_EXAMPLE

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 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, 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 GPI/O 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.

Interrupt Handling

When the I/O operation is complete, the device issues an interrupt which 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.

Cleanup

The PBU Manager calls the driver's clean-up 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). 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 GPI/O 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.

4.3.4 A 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)
  - ❍ Driver Control Block (subsection 5.2.1)
- ❑ *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
  - ❍ Clean-Up 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 (1) handles interrupts and (2) performs asynchronous transfers
  - ❍ SIO Routine (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 (DDF)*—(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 (e.g., GPI/O 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 directive in Pascal and #include keyword in C) 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, refer to *Programming with General System Calls.*

NOTE: Unlike Pascal, the C programming language is case sensitive; therefore, all system procedure names (such as GPI/O routines) must be lowercase, consistently with their appearance in the system insert files. Likewise, any global names in C that are accessed by GPI/O routines must be lowercase.

5.1 System Insert Files

Two system insert files must be included in any GPI/O device driver:

- /SYS/INS/BASE.INS.lan*: Base definitions for the particular language in which the driver is written
- /SYS/INS/PBU.INS.lan: GPI/O routines

Other insert files that you might want to include are

- /SYS/INS/VFMT.INS.lan: Variable formatting (VFMT) calls
- /SYS/INS/ERROR.INS.lan: Error reporting calls

*Substitute either .PAS (Pascal) or .C (C) for .lan.

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 and private. Public insert files declare data structures and driver routines that the application can use, whereas 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. But 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 the next subsections describe public and private insert files separately.

Examples of public and private insert files appear in the BM_EXAMPLE in Appendix E, sections E.1 and E.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 below.

CSR Page

The *CSR page* is a record that defines the controller's internal registers that 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.

Following is an example of a CSR page as declared in a private insert file:

```
mm_csr_page_t = [device] packed record case integer of
       0 : (command        : char;
            status         : char;
            pad_1          : char;
            r_data         : char;
            pad_2          : char;
            int_status     : char;
            pad_3          : char;
            pad_4          : char;
            d_data         : char;
            int_enable     : char;
            pad_5          : char;
            pad_6          : char;
            pad_7          : char;
            pad_8          : char;
            pad_9          : char;
            d_data         : char);
       1 :  (all           : array [0..1023] of char);
       end;
```

The [device] attribute 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 itself is of the variant type so that, in this case, the CSR page can be accessed either as a whole or register by register; it could, however, just as well have been constructed of fixed parts only, depending upon the requirements of the driver. Each field is of the char data type because each register consists of eight bits—the space allocated to the char data type. (Use of the char data type, or arrays of chars, to specify fields ensures that the compiler does not perform improper compressions.) If any of the registers should consist of a set of single–bit flags, each flag should be declared as a Boolean. The field *all* is declared as an array of 1024 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:

```
                pads : array [5..9] of char;
```

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 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:

```
    mm_csr_page_ptr_t = record case integer of
          0 : (c : ^mm_csr_page_t);
          1 : (p : pbu_$csr_page_ptr_t);
          end;
```

The pointer in this example is declared as a variant record so that it can be used in two different contexts, either in the driver or in a GPI/O routine.

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

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 communication 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. 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 drivers written in C, refer to Appendix C, subsection C.2.6.

The driver control block in BM_EXAMPLE is declared in BM.PVT.PAS as follows:

```
    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_$sio called from
                                           interrupt side }
          io_addr : bm_$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 }
```

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. However, you may wish to list all external routines (except those already referenced in the public insert file; refer to subsection 5.2.2) for documentation purposes.

If you are writing your driver in C, you needn't be as concerned about where to allocate global routines; refer to Appendix C, subsection C.2.6.

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 public insert file, BM.INS.PAS, 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
- Clean–Up

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 descrption of the PFM calls in the *DOMAIN System Call Reference*.

NOTE: Unlike Pascal, the C programming language is case sensitive; therefore, all system procedure names (such as GPI/O routines) must be lowercase, consistently with their appearance in the system insert files. Likewise, any global names in C that are accessed by GPI/O routines must be lowercase.

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. The following subsections 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.3 (Pascal) and Appendix F, section F.3 (C).

6.1.1 Initialization Routine Format

The initialization routine is called by GPI/O software and must, therefore, conform to the following calling sequence:

FORMAT

initialization_routine_name (unit, ddf_ptr, csr_ptr, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format.
ddf_ptr	A virtual address of the DDF in PBU_$DDF_PTR_T format. This data type is described in Appendix B, section B.1.
csr_ptr	The virtual address of the device's CSR page in PBU_$CSR_PAGE_PTR_T format.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

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

If you are writing your driver in C, you should bear in mind that GPI/O software is written in Pascal and therefore passes parameters by reference, whereas routines written in C expect parameters to be passed by value. To compensate for this discrepancy, the initialization routine (and any other routine called by GPI/O software) must declare each parameter with the indirection operator (*) so that your routine gets the value of the parameter and not its address. Refer to the example of the initialization routine written in C in Appendix F, section F.3. For additional information on programming in C, refer to Appendix C, section C.2.

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 communication between their call and interrupt sides. (The interrupt side of the driver allocates this storage area, using the DEFINE clause; if you are writing your driver in C, refer to Appendix C, subsection C.2.6.) 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, 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; refer to Chapter 7, subsection 7.1.5). 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 AT-compatible bus does not generate bus time-out errors, which means that you cannot use PBU_$READ/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, 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 status is nonzero, program control returns to PBU_$ACQUIRE.

```
pbu_$write_csr(bmcb.pbu_unit_number,       { number of this 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;
```

In the next example (taken from /DOMAIN_EXAMPLES/GPIO_EXAMPLES/AT_EXAMPLE), the driver tests for the presence of the floppy controller on the AT-compatible bus by issuing a device-specific command. The SPECIFY command is used to initalize the controller, but it also be used to test for device presence, since the only reason that the command might fail would be if the device were not responding. Thus, in the following segment, the driver assumes that, if the return status (i.e., STS) from the call to PROC_CMD_STS is set at any other value than OK, the device is not present:

```
f.command := specify;
proc_cmd_sts(spec_cmds, 3, 0, sts);  { tell the floppy what it looks like }
IF sts <> ok THEN                    { guess it's not there }
BEGIN
     error_$print(status);
     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.

Allocating Hard-Wired Control Blocks on the MULTIBUS

The initialization routine allocates hard-wired addresses by calling the routine PBU_$ALLOCATE_MAP, specifying the memory's starting address within MULTIBUS memory and giving a length, which must be in 1024-byte increments. As stated in Chapter 1, subsection 1.2.2, each I/O map entry maps one page of MULTIBUS memory address space. PBU_$ALLOCATE_MAP 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_$ALLOCATE_MAP and specifies MULTIBUS address FC00 (because it is a page-aligned address) and a length of 1024. Because the routine specifies a particular address, the force_flag parameter must be set to "true"; see Appendix B for a syntactic description of the GPI/O call PBU_$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.

Defining Page-Aligned Control Blocks

Device drivers for controllers using hard-wired initialization control blocks or AT-compatible and VME controllers that need to align a 1K-byte 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):

```
PROGRAM TOUCH_PAGE;
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pbu.ins.pas';
%list;

TYPE   controller_t = RECORD              { Define the controller's page }
       page : ARRAY[0..bytes_per_page - 1]OF char;
       end;

TYPE   temp = RECORD CASE INTEGER OF      { Dummy type for }
       0: (p : ^controller_t);            { manipulating pointer }
       1: (i : integer32);
       end;

VAR    buffer : ARRAY[0..bytes_per_page*2-1]OF char;
       pointer : temp;

BEGIN
       pointer.p := addr(buffer);          { point to start of buffer }
       pointer.i := (pointer.i + bytes_per_page - 1)
                   & (-bytes_per_page);    { round up to page }
       WITH  pointer.p^ : controller DO BEGIN
             controller.page[5] := chr(0);
             END;
END.
```

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 10.1.1.

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: it 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: (1) command-specific routines that the application can call and (2) an internal routine 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. BM_$READ and BM_$WRITE are the command-specific routines, and BM_COMMAND is the internal routine (see Appendix E, section E.3 [Pascal] and Appendix F, section F.4 [C]). Depending on whether the application wants the controller to do a read or write operation, it calls one of the two command-specific routines, passing as parameters the data buffer to be transferred, its address, and the bulk memory address. These two routines pass the same parameters, along with the specific controller commands, to BM_COMMAND. First, BM_COMMAND 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. Next, BM_COMMAND wires down the buffer by calling PBU_$WIRE (wiring ensures that no buffers are removed from memory, or "paged out," during the I/O operation) and then calls BM_$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 bulk memory }
bm_$sio(status);                   { start up the i/o operation }
```

Finally, 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.3 (Pascal) and Appendix F, section F.6 (C).

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 timeout
- Quit fault from the terminal user

PBU_$WAIT 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 fully described in Chapter 8, section 8.3.
- 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.

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

The caller can also permit quit faults (CTRL/Qs) 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_$WAIT routine in BM_EXAMPLE specifies "index" as the output parameter of PBU_$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 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   { he didn't like something }
                 status.fail := true;
                 RETURN;
                 END;

         END    { of not done }

     ELSE index := 0;    { transfer already complete }

     CASE index OF

         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;

         1:   status.all := bm_$timeout;

         2:   status.all := bm_$quit_during_wait;

         END;   { of CASE }
```

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 as arguments to PBU_$GET_EC the unit number of the device and a key that indicates which eventcount to get (currently, the key must be PBU_$GET_DEVICE_EC). PBU_$GET_EC returns a value that identifies the device's eventcount. Drivers need to call PBU_$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 *DOMAIN System Call Reference*.)

```
ec_ptr_list[i] := dev_ec_ptr;               { pointer returned by PBU_$GET_EC }
ec_val_list[i] := ec2_$read(dev_ec_ptr^)+1; { value of ec to wait for }
IF NOT op_already_done THEN BEGIN
     ec_index := ec2_$wait(ec_ptr_list,ec_val_list,status);
     IF ec_index = i THEN op_already_done := false;
     END;
```

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_$GET_EC is an EC_$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 Clean-Up Functions

User-written device drivers can optionally supply a clean-up routine to perform device-specific clean-up functions before a device is released. The routine PBU_$RELEASE obtains the entry point of the clean-up routine from the DDF and calls the routine during device release. The clean-up routine in BM_EXAMPLE is called BM_$CLEANUP; refer to Appendix E, section E.3 (Pascal) and Appendix F, section F.9 (C).

Functions performed by the clean-up 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 AT-compatible device drivers, ensuring that the last call to PBU_$DMA_START had a corresponding call to PBU_$DMA_STOP.
- Releasing any acquired I/O resources.

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

The clean-up routine is called by GPI/O software and must, therefore, conform to the following calling sequence:

FORMAT

cleanup_routine_name (unit, force_flag, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format.
force_flag	A Boolean value that indicates whether or not the clean-up routine can abort the device release operation. If this parameter is set to true, the device is released regardless of the status returned by the clean-up routine. If this flag is set to false, the clean-up routine can abort the release procedure by returning a nonzero status code. Upon receipt of the status, PBU_$RELEASE aborts device release and returns to its caller. This flag is the same as the force_flag parameter for PBU_$RELEASE.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

If you are writing your driver in C, you should bear in mind that GPI/O software is in written Pascal and therefore passes parameters by reference, whereas routines written in C expect parameters to be passed by value. To compensate for this discrepancy, the clean-up routine (and any other routine called by GPI/O software) must declare each parameter with the indirection operator (*) so that your routine gets the value of the parameter and not its address. Refer to the example of the clean-up routine written in C in Appendix F, section F.9. For additional information on programming in C, refer to Appendix C, section C.2.

Chapter 7

Transferring Data

Data can be transferred between the application and the device by means of DMA, memory mapping, or programmed I/O—which method you use depends on the kind of controller your driver supports. GPI/O routines are designed to facilitate the transfer, no matter what method you use.

We provide two kinds of calls, PBU_$ and PBU2_$, for several GPI/O operations. PBU_$ calls are reserved for nodes with a 16-bit MULTIBUS; PBU2_$ calls are for use with the 20-bit MULTIBUS, AT-compatible bus, and VMEbus. This manual refers to a PBU_$ routine and its PBU2_$ counterpart as PBU[2]_$routine_name.

> NOTE: Unlike Pascal, the C programming language is case sensitive; therefore, all system procedure names (such as GPI/O routines) must be lowercase, consistently with their appearance in the system insert files. Likewise, any global names in C that are accessed by GPI/O routines must be lowercase.

7.1 DMA Transfers

A *DMA transfer* to or from processor memory occurs when a DMA controller makes memory references to bus address space. We support DMA transfers on the MULTIBUS, AT-compatible bus, and VMEbus. The following subsections describe how to use GPI/O routines to prepare for a DMA transfer on any of these buses. You should also refer to Chapters 1 (MULTIBUS), 2 (VMEbus), and 3 (AT-compatible) for additional bus-specific information.

7.1.1 DMA Transfers on the MULTIBUS

As mentioned in Chapter 1, the I/O map translates memory references to MULTIBUS address space into processor memory references. Before the controller can initiate memory references, the device driver must establish an association between the pages of MULTIBUS address space and the pages of processor memory, known as *mapping an I/O buffer*.

The driver maps an I/O buffer by

- Allocating MULTIBUS 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 MULTIBUS address space.

NOTES: If a device driver has specified a 20-bit MULTIBUS address and is running on a node with a 16-bit MULTIBUS, the GPI/O routines will return an error indication because the 16-bit MULTIBUS supports only 16-bit MULTIBUS addresses.

PBU2_$ routines used with MULTIBUS devices take addresses and lengths specified as 4-byte integers rather than 2-byte integers. Drivers running on nodes equipped with a 16-bit MULTIBUS can also use PBU2_$ routines; but on nodes with a 20-bit MULTIBUS, a 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.

Allocating MULTIBUS Address Space

All controllers use the same MULTIBUS address space to access processor memory. The region is 64K bytes in size for nodes with a 16-bit MULTIBUS and 1024K bytes for nodes with a 20-bit MULTIBUS. Since I/O buffers concurrently in use by controllers must not overlap in MULTIBUS address space, the device driver must ensure against overlap by allocating a section of MULTIBUS address space for the controller. You use the GPI/O 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 MULTIBUS 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 MULTIBUS address may already be in use (refer to Chapter 6, subsection 6.1.4). In this case, the driver has several choices:

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

The following call to PBU_$ALLOCATE_MAP (from the initialization routine of BM_EXAMPLE) allocates an area of the I/O map corresponding to the largest block (32K bytes) that the driver ever reads or writes. The constant BM_$BLOCK_LEN is declared in BM.INS.PAS as having a value of 32768; BMCB.BM_IOVA contains the start of the allocated area of MULTIBUS 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 }
```

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 for MULTIBUS devices 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 calls PBU_$WIRE just before sending the read or write command to the routine, as follows:

```
bmcb.buflen := len;               { save length of buffer to wire }

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 }
```

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, subtract 256 from the number of pages of memory that the node has. For example, for a node with one megabyte of main memory, 1024 pages minus 256 (pages) equals 768, so drivers must wire fewer than 768 pages. The absolute maximum number of pages that can be wired is 4096.

The driver can also wire an I/O buffer by defining a permanently allocated storage area in the interrupt routine and copying data to 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. In the following example, the storage area is part of the driver control block, which is DEFINEd in the interrupt side (drivers written in C do not require the DEFINE clause; refer to Appendix C, subsection C.2.6).

The private insert file specifies the dimensions of the buffer:

```
CONST buflen =     4096;              { buffer length in bytes }
      buflast =    buflen - 1;        { index of last byte in buffer }
      hdrlen =     4;                 { # bytes in buffer header }

TYPE  buf_t =      ARRAY [0..temp_buflast] OF char;     { buffer to wire }
```

The buffer itself (BUF) is declared as an area of the control block:

```
TYPE  control_block_t = RECORD;              { control block for driver }
             unit:         pbu_$unit_t;
             ddf_ptr:      pbu_$ddf_ptr_t;
             csr_ptr:      csr_page_ptr_t;
             .
             .
             .
             buf:   buf_t                    { buffer to wire }

VAR   dev_$cb:     EXTERN control_block_t ;
```

The interrupt library DEFINEs the buffer, as well as any other routines and data structures that must be be wired there:

```
DEFINE       dev_$sio,
             dev_$cb;
```

When the device is acquired and the interrupt side becomes wired, DEV_$CB.BUF is also wired—and, therefore, cannot be paged out—so that data from the application's buffer can be copied to or from it for DMA operations.

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

Setting Up the I/O Map

After the driver has allocated pages of MULTIBUS address space for the buffer and wired the buffer into processor memory, it must establish the mapping between the buffer and the pages of MULTIBUS address space by calling the GPI/O routine PBU[2]_$MAP. This routine takes three arguments:

- The I/O buffer
- The I/O buffer's length
- A MULTIBUS address within any page of the area allocated by PBU[2]_$ALLOCATE_MAP

PBU[2]_$MAP establishes the displacement within MULTIBUS 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 (from BM_EXAMPLE), PBU_$MAP is called in the start I/O routine (BM_$SIO), just before touching the controller's command register. The return value (CSR.IOVA) is the buffer's address, which is written to the controller's address register:

```
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 }
```

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]_$MAP to establish a correspondence between this preallocated buffer and a section of MULTIBUS address space. The routine saves the MULTIBUS address returned by PBU[2]_$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. The user-written clean-up routine frees the allocated I/O map space. Appendix D, section D.3 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 an earlier subsection, "Wiring I/O Buffers."

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, as 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_$WIRE returns an error. In this case, the driver can wire the buffer by

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

Scatter-Gather Operations

A scatter-gather I/O operation consists of reading (scattering) or writing (gathering) a single block of data in MULTIBUS 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.

PBU[2]_$MAP can be used to implement limited forms of scatter-gather by observing the following rules:

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 MULTIBUS address that is one page higher than the MULTIBUS 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 discussion of an 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 MULTIBUS memory—in this example, 3000.

Next, the driver calls PBU[2]_$MAP, specifying iova 3000, the length 5C, and buffer address 2A9FA4—the start of the area in processor address space where the header is to be transferred. The buffer address is obtained by subtracting the length of 5C from a page-aligned address in processor address space (2AA000), giving the starting address 2A9FA4. This procedure satisfies rule 1 by ensuring that the first section ends on a page-aligned boundary. PBU[2]_$MAP returns the header's starting address (33A4) in MULTIBUS 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]_$MAP, specifying iova 3400, the length of the data 1A0, and the address 2E4400. PBU_$MAP returns a MULTIBUS address 3400 for the data, in accordance with rule 2, which requires the driver to map each subsequent block to a MULTIBUS address that is one page higher than the MULTIBUS address of the previous block.

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

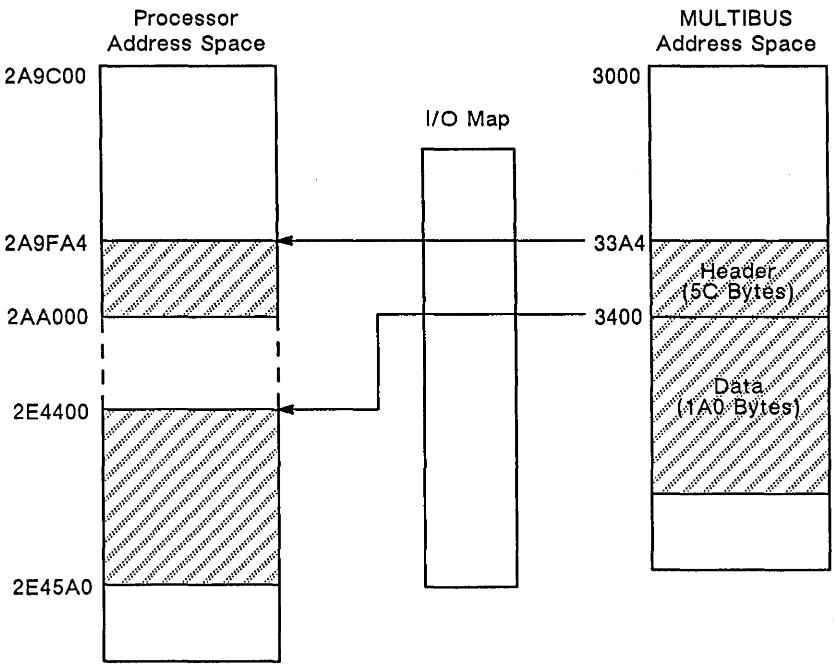


Figure 7-1. Mapping Discontiguous Buffers

7.1.2 DMA Transfers on the VMEbus

The following restrictions apply to DMA operations on the VMEbus:

- Because there is no address translation mechanism for the VMEbus, a driver for a VME device must not make any calls to PBU[2]_$ALLOCATE_MAP or PBU[2]_$MAP.
- The driver wires 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.
- Our system operates on page (1024 bytes) boundaries. This means that, because there is no mapping mechanism between the VME device and physical memory, the device must support scatter-gather (and the driver must be able to implement scatter-gather) if it is to perform a DMA operation of more than 1024 bytes or if the transfer is to cross a page boundary.

7.1.3 DMA Transfers on the AT-Compatible Bus

Because there is no address translation mechanism for the AT-compatible bus, a driver for an AT device must not make any calls to PBU[2]_$ALLOCATE_MAP or PBU[2]_$MAP. And since the DMA hardware does not provide for scatter-gather operations, the amount of data that can be transferred in one continuous DMA operation must not exceed 1K byte, must lie within a page boundary, 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 10.1.1. If the DMA hardware being used is the system DMA on the mother board rather than the DMA on the device, the driver must wire its I/O buffer by calling the routine PBU2_$WIRE. Devices having their own DMA hardware (so-called demand-DMA devices) must call PBU_$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 for its DMA hardware.

Drivers for AT–compatible devices that do not have their own DMA hardware but use the DMA hardware on the mother board must call PBU_$DMA_START and PBU_$DMA_STOP to start and stop a DMA operation. 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 CHANNEL_IN_USE error message. If you get this message, however, you can call PBU_$DMA_STOP to release the channel.

Drivers for devices using their own DMA hardware must call PBU_$DMA_START once, specifying the PBU_DMA_CASCADE option. This option reserves the DMA channel and provides bus arbitration. PBU_$DMA_STOP must be called when the device is released.

> **NOTE:** DMA lines typically float on the AT–compatible bus; refer to Chapter 3, section 3.4 for important information on enabling and disabling DMA lines.

The following program segments are from a driver for an AT–compatible 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 1K–byte limit per DMA operation. If the amount of data exceeds 1K, the interrupt routine restarts the DMA operation for the next 1K block of data and continues to do so until all of the data has been 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
                                                   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 the driver's copy to the
                                               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 1K chunks. }

       cb.dma_buf_ptr := va;
       if cnt > 1024 then begin;
            cb.dma_remainder := cnt - 1024;
```

```
          cnt := 1024;
          end
     else cb.dma_remainder := 0;

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

     pbu_$dma_start (cb.pbu_unit, cb.dma_chan, cb.dma_dir,
                     cb.dma_buf_ptr^, cnt, [], status);   { start DMA }
     if status.code <> status_$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 ... }
dma_fail:
         discard(pbu_$dma_stop (cb.pbu_unit,
                 cb.dma_chan, st));    { ... so abort DMA.  The discard function
                                         allows us to throw away the value that
                                         this function returns—we don't need
                                         it—without the compiler complaining. }
          cb.dev_control := cb.dev_control -
                  [dma_ienable, dma_enable];  { turn off device's DMA enables }
          csr.dev_control := cb.dev_control;  { write the driver's copy out to
                                                the csr page }
          end; { if not cb.dma_complete }

    end;   { with cb_ptr^, cb.csr_ptr^ }
    return;
end { dma_data };
```

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

```
FUNCTION dev_$int: pbu_$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)); { The discard function allows us to
                                          throw away the value that this function
                                          returns - we don't need it - without
                                          the compiler complaining. }

          { 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 1K block }

                    cb.dma_buf_ptr := univ_ptr (integer32(cb.dma_buf_ptr) +
                                                   1024);

                    { check to see if we have more than 1k left to transfer }

                    if cb.dma_remainder > 1024 then begin
                         cnt := 1024;
                         cb.dma_remainder := cb.dma_remainder - 1024;
                         end
                    else begin
                         cnt := cb.dma_remainder;
                         cb.dma_remainder := 0;
                         end;

                    { start up the DMA channel for the next 1K block }

                    cb.dev_control  := cb.dev_control - [dma_enable,
                              dma_ienable];             { disable DMA interrupt
                                                          and DMA }
                    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 }
```

For another example of a DMA data transfer on the AT-compatible bus, see /DOMAIN_EXAMPLES/GPIO_EXAMPLES/AT_EXAMPLE. This driver lets the System Interrupt Handler do all the interrupt processing. Also, to speed up the entire I/O operation, it uses a technique known as *double buffering*: it allocates and wires two buffers, and while the driver is waiting for the device to complete its DMA into one buffer, it is copying the contents of the other buffer into the application's file.

7.1.4 Releasing I/O Resources After Data Transfer

The driver uses GPI/O routines to release I/O resources following the completion of a DMA transfer. The following paragraphs describe what routines to call and how to use them. (Those paragraphs that apply only to the MULTIBUS are so indicated).

Unmapping the I/O Buffer on the MULTIBUS

Device drivers do not need to unmap an I/O buffer after a data transfer. If the driver re-calls PBU[2]_$MAP and specifies the same area of the I/O map, the operation effectively unmaps the previously mapped buffer. A driver usually unmaps an I/O buffer with PBU[2]_$UNMAP to protect it from erroneous references by a controller. The MULTIBUS address specified as an argument to the call PBU[2]_$UNMAP must be the address returned by PBU[2]_$MAP.

Unwiring the I/O Buffer

Device drivers that have wired their buffers 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

Deallocating the I/O Map on the MULTIBUS

Because each MULTIBUS device can have only one piece 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. Refer to subsection 7.1.1 for more information about I/O resource allocation.

7.1.5 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 clean-up 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 clean-up routine that deallocates I/O resources and disables the device. The driver should monitor the allocation of I/O resources, including

- The area of the I/O map that has been allocated (applicable only to the MULTIBUS)
- 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 clean-up 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 AQDEV command (through PBU_$ACQUIRE) 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 3COM controller is an example of 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. Device drivers running on a node equipped with a 16-bit MULTIBUS call GPI/O 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 the two GPI/O routines PBU2_$MAP_CONTROLLER and PBU2_$UNMAP_CONTROLLER. Drivers for VME and AT-compatible devices call the two GPI/O routines PBU2_$MAP_CONTROLLER and PBU2_$UNMAP_CONTROLLER.

NOTE: If a node with a 20-bit MULTIBUS is fully configured with 3M bytes of memory, only 512K bytes of the MULTIBUS address space is available for memory-mapped operations.

7.2.1 Referencing Controller Memory

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

For the MULTIBUS:

- Controller memory must be page aligned and must occupy only the first 32K of MULTIBUS memory space on nodes with a 16-bit MULTIBUS and 1M byte 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 has not been 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; refer to Table 2-1.
- The area of memory space occupied by controller memory is permanently unavailable to DMA operations by any controller.

For the AT-compatible bus:

- Controller memory must be page aligned.
- Controller memory must occupy user-available locations in processor physical address space; refer to Table 3-2.

7.2.2 Mapping Controller Memory

The device driver calls PBU[2]_$MAP_CONTROLLER to map controller memory to processor address space. PBU[2]_$MAP_CONTROLLER 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 AT-compatible bus does not generate bus timeouts, 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;
```

The following precautions apply only to the MULTIBUS:

- PBU[2]_$MAP_CONTROLLER makes the area of MULTIBUS memory space allocated to the controller unavailable for any subsequent DMA operations. Note that the MULTIBUS 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]_$MAP_CONTROLLER is called, the I/O map allocation routines may allocate, for the memory-mapped controller or for another controller, an I/O map area that overlaps the area allocated to the memory-mapped controller. As a precaution, you should configure the controller memory to occupy the high end of MULTIBUS memory space (0–7FFF on nodes with a 16-bit MULTIBUS, 0–FFFFF on nodes with a 20-bit MULTIBUS), since the I/O map allocation routines allocate I/O map areas from low addresses to high addresses.
- If the driver of a memory-mapped controller needs to perform a DMA transfer, it can call PBU[2]_$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 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 much the simplest of the three methods—there are no buffers to allocate and wire (and deallocate and unwire), no I/O map (in the case of the MULTIBUS) to set up, no calls to PBU_$DMA_START/STOP (in the case of the AT-compatible bus). But on the MULTIBUS and VMEbus, programmed I/O is also generally the slowest, since (1) the rate of transfer is limited to one word or byte at a time, (2) the transfer itself is under the control of software rather than hardware, and (3) the device must inform the processor after each transfer. In the case of the AT-compatible bus, however, programmed I/O is appreciably faster than DMA because the processor is so much faster than the DMA hardware on the mother board and because DMA transfers on the AT-compatible bus are limited to 1K byte. Thus, given the choice, you may wish to opt for programmed I/O, especially in drivers for slow (e.g., serial lines) or fast (e.g., hard disk) buffered devices, and reserve DMA for devices of intermediate speed (e.g., 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.2 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; refer to Chapter 10, section 10.1.

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 GPI/O routines) must be lowercase, consistently with their appearance in the system insert files. Likewise, any global names in C that are accessed by GPI/O routines must be lowercase.

8.1 Dos and Don'ts of the Interrupt Side

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. (This restriction does not apply to drivers written in C; refer to Appendix C, subsection C.2.6.)

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; refer to subsection 8.2.4. The only GPI/O routines that an interrupt-side routine can call are PBU_$MAP, PBU2_$MAP, PBU_$UNMAP, PBU2_$UNMAP, PBU_$DEVICE_INTERRUPTING (which determines whether an interrupt has occurred), PBU_$DMA_START, and PBU_$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" (refer to Chapter 10, subsection 10.1.2).

NOTE: PBU_$ACQUIRE and AQDEV will refuse to load an interrupt library having 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 32K bytes, including procedure, data, and debug information.

8.2 The 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

The following subsections discuss these functions as well as other aspects of interrupt routines.

8.2.1 Interrupt Routine Format

The interrupt routine is called by GPI/O software and must, therefore, conform to the following format:

```
FUNCTION interrupt_routine (unit : pbu_$unit_t) : pbu_$interrupt_t;
```

The input parameter, unit, is optional (for more information, refer to Chapter 9, section 9.6). The output parameter, return_flags, is a set of flags in PBU_$INTERRUPT_FLAGS_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 (0–7). 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, 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. In AT_EXAMPLE (which does not have an interrupt routine), interrupts are first enabled in the device initialization routine (START_FLOP) and re-enabled after each call to PBU_$WAIT in driver routine PROC_CMD_STS.

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 AT-compatible bus; refer to 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 has 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. Thus, the device's eventcount provides the method by which the interrupt handler can signal to the driver's call side that an interrupt has completed. *Programming with General System Calls* 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 an example of a driver that does not have an interrupt side, see /DOMAIN_EXAMPLES/GPIO_EXAMPLES/AT_EXAMPLE. For information about interrupt processing overhead, refer to Appendix D, section D.2.

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.

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
- In the case of the MULTIBUS, 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.4 (Pascal) and Appendix F, section F.8 (C).

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.

At the command level in the owning process, information about the fault can be obtained by using the FST (Fault_Status) command, as in the following example:

```
$ #  This sequence gives the fault info for the owning process:
$ #
$ fst -a
Fault Diagnostic Information
Fault Status   = 00120017:
fault in user-space interrupt handler for pbu device (OS/fault handler)
User Fault PC  = 0001AB5A
D0-D7:  00120017 00000030 00000004 00000074 FFFF0003 00000004 00000003 00000005
A0-A7:  0020851C 00E10CB8 00E10DD2 00E0F25E 00E0F25E 00049064 00276BE0 00276BC0
Supervisor ECB = 00000000
Supervisor SR  = 0000
Supervisor PC  = 00000000
$ #
$ # ...and this sequence tells what happened to the interrupt routine:
$ #
$ fst -a -u <unit_number>
Fault Diagnostic Information
Fault Status   = 00120001:
odd address error (OS/fault handler)
Access Addr    = 00000001
IR             = FFFC
Acc. Info      = 1101
User Fault PC  = 002B8222
D0-D7:  00000004 00000001 00000000 FFFF0004 FFFFFFFF FFFFFFFF 0000FFFF 00000001
A0-A7:  002B82C4 002C876E 00000001 00276BD4 00E47DE0 002B82D4 002C875E 002C874E
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 AQDEV with the -DB option, can be used to isolate the logic that caused the fault.

8.2.5 Mapping Buffers from the Interrupt Routine

Drivers for MULTIBUS 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 (refer to 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.

Note that DMA drivers for AT-compatible devices can use this same technique, except that they would call PBU_$DMA_START and PBU_$DMA_STOP instead of PBU[2]_$MAP and PBU[2]_$UNMAP.

8.3 Starting an I/O Operation

The 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. What needs to be said here is an explanation of why you might want to include an SIO routine in the interrupt side. Why, then? Mainly because the driver may have more data to transfer than can be handled in one I/O operation and because 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, then 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.4 (Pascal) and Appendix F, section F.5 (C).

Chapter 9

Shared Drivers

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

The general organization of a shared 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 shared 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 clean-up entry points, interrupt entry points, and other useful information.

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

See /DOMAIN_EXAMPLES/GPIO_EXAMPLES/SHARED_EXAMPLE for an example of a shared driver.

9.1 Controlling Multiple Processes

The chief design consideration of a shared driver is how to control multiple processes attempting to access the same procedure or data structure. Specifically, a shared 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 this:

```
procedure P (parameters);
var lock:  mutex_lock_rec_t;
begin
         if mutex_$lock(lock,wait_time) then
         begin
         .
         .
         .
         { body of procedure }
         .
         .
         .
         mutex_$unlock (lock);
         end
end {P};
```

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 (eventcount);
discard (mutex_$lock(lock));
```

9.1.2 Synchronization

As described in Chapter 8, GPI/O software provides one built–in eventcount per device as a means of synchronizing device operations with driver routines. But a shared 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 GPI/O 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 full 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. PBU_$GET_EC returns the pointer to the built-in eventcount in the device control table entry. This is advanced under control of the return value from the interrupt handler. The procedure PBU_$WAIT can be used to wait on this eventcount and a timeout. However, it should only be used in a shared 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 shared 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 shared driver differs from a global library in that a shared 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 shared 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 shared driver, you must specify RWS_$GLOBAL_POOL as an input parameter (for private drivers, specify RWS_$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 (i.e., 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 (i.e., at system initialization), and all cleanup occurs when the driver is unloaded (i.e., at system shutdown). In other words, there is no per-process initialization or cleanup for shared drivers. Each procedure in a shared driver must be so designed that it restores the module invariant (i.e., 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 shared driver, however, the fault is reflected back to the process that last touched the driver. The reason for this difference is that in a shared driver you don't want the fault to reflect back to the owning process, which is the DM or the SPM. 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, shared drivers must be loaded at system initialization. To load a shared driver, you place the DDF for the shared device in the directory /DEV/GLOBAL_DEVICES. Immediately after loading the global libraries, the system searches the directory /DEV/GLOBAL_DEVICES for shared device drivers and 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 or SPM 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 (i.e., the boot shell).

When the system exits, it calls the clean-up routine of each shared 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 GPI/O software package allows the same driver (either shared 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 individual device is specified by its own DDF. The DDF specifies the interrupt level, CSR page, entry points for the initialization and clean-up 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 clean-up entry points are called individually, for each device.

The interrupt handler has an input parameter, PBU_$UNIT_T, that identifies the unit that 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

```
FUNCTION Interrupt_handler (unit: pbu_$unit_t): pbu_$interrupt_return_t;
```

Chapter 10

Binding and Debugging

10.1 Binding the Device Driver

The purpose of *binding* is to create a single output object file out of the several modules that make up your 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, you use two bind operations to produce two separate executable modules. The first module is the call-side module (input for CALL_LIBRARY in the DDF); the other is 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, driver routines, and optional clean-up routine. In the sample shell script below, this module is called CALL_SIDE.BIN.

The interrupt-side module contains the interrupt-side routine(s), bound with the GPI/O source library /LIB/PBU_INT_LIB. The interrupt-side module also contains any communication areas (e.g., 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.

The sample shell script for device driver binding follows. If you use this script, substitute your own pathnames for those shown in angle brackets.

```
von
# bind the call side of the driver
#
bind -allmark <call_side.bin> -b  <call_lib_pathname> -map <map_pathname>
# bind the interrupt side
#
bind -allmark -sys -b <interrupt_pathname> -map <map_pathname> - <<!
<interrupt_side.bin>
/lib/pbu_int_lib
-und
-end
!
```

10.1.1 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 1K-byte area of memory called DMA_BUFFER, first you would declare it

```
VAR (buffer_sec)
      dma_buffer : ARRAY[0..1023]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.

The driver in /DOMAIN_EXAMPLE/GPIO_EXAMPLE/AT_EXAMPLE uses the -ALIGN option. For additional information, refer to the *Domain Binder and Librarian Reference.* For information about placing variables in sections, refer to the *Domain Pascal Language Reference* and to the discussion of C's #section command in the *Domain C Language Reference.*

10.1.2 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; refer to 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; refer to 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 runtime.)

If your driver requires 32–bit mathematics, you may get undefined references to the run–time 32–bit math package when binding the driver. If you see a reference to a global name that begins "M$," you should try rewriting the expression using 16–bit variables. If this is not suitable, you can bind your driver to a bindable copy of our math package called M$ARITH.BIN, which is provided for this purpose in the GPI/O package. Following is an example of a math package call generated by the compiler. The source code is

```
int x, y
foo ()
{ return x*y; }
```

When you bind this file specifying the –SYS option, you get the following:

```
$ bind junk.bin -sys
Undefined globals:

          m$mis$lll                First referenced in J.BIN
```

In this example, m$mis$lll is a compiler–generated call to the math package. To get this global resolved, bind your driver to the GPI/O copy of the math package, as follows:

```
$ bind junk.bin gpio/m$arith.bin -sys
All globals are resolved.
```

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

10.2 Debugging the Device Driver

You can use the high–level language debugging tool (DEBUG) on the call–side library by following the procedure outlined in subsection 10.2.1, 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. Using DEBUG on any interrupt–side routine may cause your system to crash during debugging or immediately after restarting.

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 DEBUG, you can copy them into the interrupt–side library where they belong.

- There is no way to set breakpoints 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.

CAUTION: Do not use DEBUG to examine or print any variable touched by the interrupt handler while interrupts are enabled. If you need to see such a variable, stop your call library routine at a point where it has disabled interrupts. You can then look around freely, but be sure that your routine touches the data again before enabling interrupts.

Although you may use DEBUG on shared drivers, there are special considerations when debugging in global space. These are discussed in subsection 10.2.2.

10.2.1 Using DEBUG on Call-Side Routines

Using DEBUG on call-side routines that are accessible from the application is simple and straightforward. Using it on the initialization and clean-up routines, however, takes some imagination, since PBU_$ACQUIRE calls each routine before you can stop it to set breakpoints.

One approach to bebugging the initialization routine is to place a read statement at the beginning of the routine, requesting input from the keyboard. Run DEBUG on the program that calls PBU_$ACQUIRE; this can be either AQDEV or your own device-acquisition program (refer to Chapter 12). At the point when your initialization routine begins executing, the read statement asks for input: type CTRL/Q to interrupt DEBUG. At this point, DEBUG is inside the initialization routine, allowing you to use DEBUG's TB and ENV commands and set any desired breakpoints. (To resume execution, type something at the keyboard to satisfy the read statement.)

For extensive debugging of your driver, it helps to prepare two copies of the call-side library. The primary copy is the one referenced by the DDF and is loaded by AQDEV or some other program you use to call PBU_$ACQUIRE. The second copy should be bound to an application that serves as a vehicle for debugging. The second copy must use the same data as the primary copy, but it provides a more convenient means to set breakpoints and call call-side routines that are accessible from the application.

To prepare the call-side library for a session with DEBUG, proceed as follows:

1. Make two copies of the driver's call-side library. In the copy that is to be bound with the application, declare the driver's data structure using the EXTERN clause in the VAR section, as follows:

   ```
   control_block : EXTERN control_block_t;
   ```

   This procedure ensures that the data structure you view in DEBUG is the same one that is being continuously updated by the device and not the static copy that DEBUG sees.

   NOTE: If your driver includes an interrupt-side library, you will have DEFINEd the data structure there; otherwise, you should DEFINE it in the primary copy of the call-side library (i.e., the copy referenced in the DDF). If you are programming in C, there is no need to specify the EXTERN clause with the data structure in one copy and the DEFINE clause in another, since all globals live in their own private sections; refer to Appendix C, section C.2.6.

2. Bind the prepared copy of the call-side library directly with the application. This enables you to access call-side routines from the application (see section 10.1 on binding).

3. Create the DDF, using the CRDDF command (see Chapter 11). When invoking the CRDDF command, be sure to specify the pathname of the primary copy of the call-side library.

4. Open two process windows on your node. We'll call one of these windows Process_1 and the other Process_2. In Process_1, acquire the device, using the AQDEV command (see Chapter 12). In Process_2, type the command

   ```
   debug -src -proc process_1
   ```

   The reason for running DEBUG in a separate process from the one in which the driver is running is that the driver must have been acquired *before* you start running DEBUG; otherwise, DEBUG takes PBU_$ACQUIRE rather than your driver as its source.

5. In Process_1, invoke the name you have given to the bound application and call-side library.

6. Return to Process_2 and step through your program, using DEBUG.

7. After debugging, insert the EOF mark to release the device.

For additional information on using DEBUG and the -PROC option, refer to the *DOMAIN Language Level Debugger Reference.*

10.2.2 Debugging the Shared Driver

A device driver that has been designed as globally sharable can always be loaded as a private, non-shared driver, which means that it can be debugged in the privacy of its own address space, like an ordinary, non-shared GPI/O driver. Thus, you can use the debugging procedure outlined in subsection 10.2.1.

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

- 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 -APOLLO switch on the debugger. This allows you to single step right into the driver in global space.

- Your specially bound version of the target application and call library; refer to subsection 10.2.1. This allows you to set breakpoints in the call library routines. Note, however, that this bound copy is in private space, even though it uses data in global space.

To debug the driver as a shared driver, check that it has already been loaded from the directory /DEV/GLOBAL_DEVICES. Next, open a process window for each application or instance of an application to use the device concurrently, and start a debugger for each of these processes. In the processes, load the specially bound versions of the applications with copies of the driver. Among other things, you will be able to track down deadlocks, observe synchronization problems, and notice shared access to data unprotected by locks.

NOTE: Since the data section of a shared driver is global, the debugger has the same data as the processes. Therefore, you don't have to worry about the debugger remapping variables and taking a page fault during interrupt handling.

The most reliable approach to debugging shared libraries with 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: 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 exercizing 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) stores static configuration information about a device, as well as information about the driver, that GPI/O 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; refer to 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 containing the user–written call–side routines. AQDEV 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 clean–up 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 (e.g., -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 also 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	Version 1 and 2 options
-NODE	any of the following:	plus any of the following:
-CALL_LIBRARY	-CSR_OFFSET	-AT
-INITIALIZATION_ROUTINE	-MEMORY_BASE (< 64K)	-DEBUG
	-MEMORY_SIZE (< 64K)	-VME
		-MEMORY_BASE (> 64K)
		-MEMORY_SIZE (> 64K)

*All Version 1 options are required for a Version 1 DDF; refer to 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.

NOTE: Do not use the shell comment character (#) within a shell script that builds a DDF.

A shell script called BUILD_BM_DDF.SH for the sample driver in BM_EXAMPLE appears below; 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
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 subdirectory. See 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. 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 *DOMAIN System Command Reference*.

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 GPI/O routines.

Certain CRDDF options (e.g., 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

GPI/O software creates a Version 2 DDF if you specify any or all of the following options: MEMORY_BASE (less than 64K), MEMORY_SIZE (less than 64K), 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

GPI/O software creates a Version 3 DDF if you specify any or all of the following options: AT, VME, DMA_CHANNEL, DEBUG, MEMORY_BASE (greater than 64K), or MEMORY_SIZE (greater than 64K). The following subsections present shell scripts for building DDFs for an AT-compatible device and a VME device. For a full description of all Version 3 options, refer to Appendix A.

11.3.1 DDF for an AT-Compatible Device

Following is a sample shell script that builds a DDF for an AT-compatible device. Note that the -csr_page iovas are supplied by the CVT_AT command (refer to Appendix A).

```
von
dlf /dev/at
crddf /dev/at - <<!
-at
-unit 4
-nodef *
-csr_page 200 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:

```
$ 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 VME Device

Following is a sample shell file that builds a DDF for a VME device:

```
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:

```
$ 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              "
```

Chapter 12

Acquiring and Releasing the Device

12.1 Acquiring the Device

PBU_$ACQUIRE 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_$ACQUIRE calls the device initialization routine specified in the DDF. For a full description of AQDEV and PBU_$ACQUIRE, refer to Appendixes A and B.

The application itself cannot call PBU_$ACQUIRE; the driver must be loaded *before* the application in order to resolve the application's references to driver entry points. There are, however, two ways to make the call:

- Invoking the AQDEV command
- Invoking a program that calls PBU_$ACQUIRE

The end result of either is the same; which one you use depends upon how many applications you are running.

12.1.1 Using AQDEV

If you plan to execute several application programs that use the device, you should 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 (i.e., 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.

12.1.2 Invoking a Program to Call PBU_$ACQUIRE

If you need to run only one application, such as a server, then you can create a device acquisition program that calls PBU_$ACQUIRE to load the driver and then invokes the application. The application cannot call PBU_$ACQUIRE because the driver must be loaded prior to the application so as to resolve the application's references to driver entry points.

The following program acquires a device, invokes the application, and releases the device:

```
program device_acquisition;
begin
      pbu_$acquire('/dev/my_dev'...);
      pgm_$invoke('/my_dir/real_application.bin'...);
      pbu_$release('/dev/my_dev'...);
      pgm_$exit;
end.
```

12.2 Releasing the Device

You can release a device by inserting the EOF mark (under the DM, the EOF mark is bound to CTRL/Z) or by calling PBU_$RELEASE. PBU_$RELEASE 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 clean-up routine, PBU_$RELEASE will call it during device release. The device acquisition program can call PBU_$RELEASE. But since PBU_$RELEASE unloads the driver library, device drivers should not call it. PBU_$RELEASE is fully described in Appendix B.

Appendix A

GPI/O Commands

This appendix describes the use, format, parameters, and options for the four GPI/O commands that the user can invoke: AQDEV, CRDDF, CVT_AT, and RLDEV.

AQDEV (ACQUIRE_DEVICE)—Acquires control of a peripheral device.

FORMAT

AQDEV pathname [-DB]

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

-DB	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.

DESCRIPTION

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 creates a new copy of the shell after it installs the device driver. To release the device, type CTRL/Z (i.e., insert the EOF mark), which causes the new shell to return to the AQDEV command. AQDEV 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.

EXAMPLE

```
$ AQDEV /DEV/MTO -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.
$
```

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 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, INTERRUPT_LIBRARY, INITIALIZATION_ROUTINE, NODE, and UNIT.

-CLEANUP_ROUTINE entry–name
Specifies the entry point name of a clean–up 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.
- 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).
- AT-compatible bus: Optional. If specified, may indicate a range (e.g., -csr_page 200 21F). If the second parameter is missing, a range of eight consecutive bytes is assumed (e.g., "-csr_page 200" assumes a range of 200–207). Use the CVT_AT command (described in Chapter 3, subsection 3.1.1) to derive properly aligned iovas.

-DEBUG Sets a flag (DDF.DEBUG) that can be used to turn on debugging logic in a driver. Specifying this option results in the generation of a Version 3 DDF.

-DISPLAY Displays the current contents of the DDF.

-DMA_CHANNEL channel-number
Specifies to the driver the DMA channel number that an AT-compatible controller will use. This option is not required, and any information it includes is not used by GPI/O software. 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 routine 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 with associated interrupt routines for each level.

If the -INTERRUPT_ROUTINE option is omitted, interrupts are processed at the level equal to the device's unit number.

-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 GPI/O routine PBU[2]_$MAP_CONTROLLER to associate a virtual address with the memory on the controller. Specifying this option with an iova less than 64K results in the generation of a Version 2 DDF; if the iova is greater than 64K, 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 64K results in the generation of a Version 2 DDF; if iova is greater than 64K, a Version 3 DDF will be generated.

-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.* -NODEF 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. PBU[2]_$MAP_CONTROLLER maps the shared controller into global address space, and PBU_$MEM_PTR returns its address. Unlike a non-shared controller, a shared controller is not automatically unmapped on abnormal termination of a device driver.

NOTE: We recommend 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).

-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 AT-compatible bus: Must lie in the range 0-15 and must not be equal to any of our devices.

-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 give 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 device descriptor 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 for a discussion of the different DDFs and examples showing 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 Version 1 DDF:

```
$ CRDDF /DEV/MT0 -
New DDF.
> -UNIT 3
> -NODE *
> -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 DDF:

```
$ CRDDF /DEV/MT0 -DISPLAY
DDF version: 1
device UID:  00030003 0000002F (unit 3, node 2F)
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. Modify existing DDF:

```
$ CRDDF /DEV/MT0 -UPDATE -INTERRUPT_ROUTINE 3 MT_$SIO
```

CVT_AT (CONVERT_AT_ADDRESSES)—Converts an AT-Compatible I/O address to a processor physical address.

FORMAT

CVT_AT [AT_addr [AT_addr] ...]   [AT_addr1—AT_addr2]

ARGUMENTS

AT_addr	An AT-compatible I/O address (in hexadecimal).
AT_addr1—AT_addr2	A range of AT I/O addresses (in hexadecimal).

DESCRIPTION

The CVT_AT command converts AT-compatible bus addresses to physical addresses in processor address space. The command reports any conflict between the AT address you specify and the address of any system-supplied device. It also supplies the iova for so-called AT (i.e., 16-bit) addresses; see example 2 below.

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 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

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 type CTRL/Z (i.e., 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

GPI/O Routines

This appendix describes the calling format, input and output parameters, and usage of the GPI/O 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 GPI/O 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.

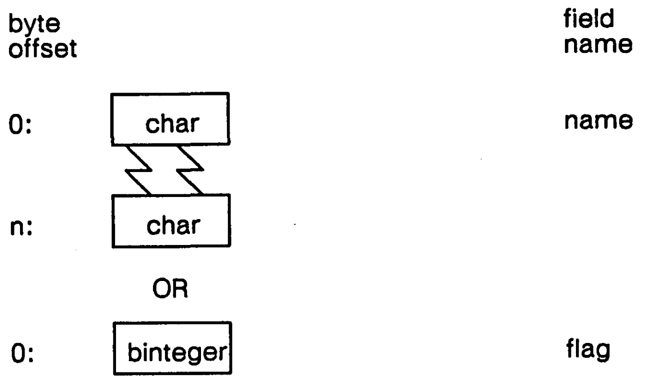
PBU_$DDF_EP_NAME_LEN	32	Maximum length of entry point names in DDF.
PBU_$DDF_MAX_LEN	1024	Maximum length of whole DDF.
PBU_$NO_CSR_IOVA	-1	Indicates no CSR page (version 2).
PBU_$MT_UNIT_NUMBER	3	Magtape unit number.
PBU_$SM_UNIT_NUMBER	4	Storage module device unit number.
PBU_$SMD_UNIT_NUMBER	4	Storage module device unit number.
PBU_$LPR_UNIT_NUMBER	6	Line printer unit number.
PBU_$IIC_UNIT_NUMBER	6	Internet interface controller unit number.
PBU_$MT_UNIT2_NUMBER	7	Placeholder for second magtape unit number.
PBU_$MAX_UNIT	7	Maximum allowable unit number.
PBU_$MIN_VME_UNIT	8	Minimum VME unit number.
PBU_$MAX_VME_UNIT	15	Maximum VME unit number.
PBU_$MAX_AT_UNIT	15	Maximum AT unit number.
BYTES_PER_PAGE	1024	Bytes per page.
PBU_$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_$CSR_PAGE_PTR_T	A 4-byte integer. A pointer to the CSR page.
PBU_$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 diagram below illustrates this data type:



Field Description

name
Interrupt routine entry point.

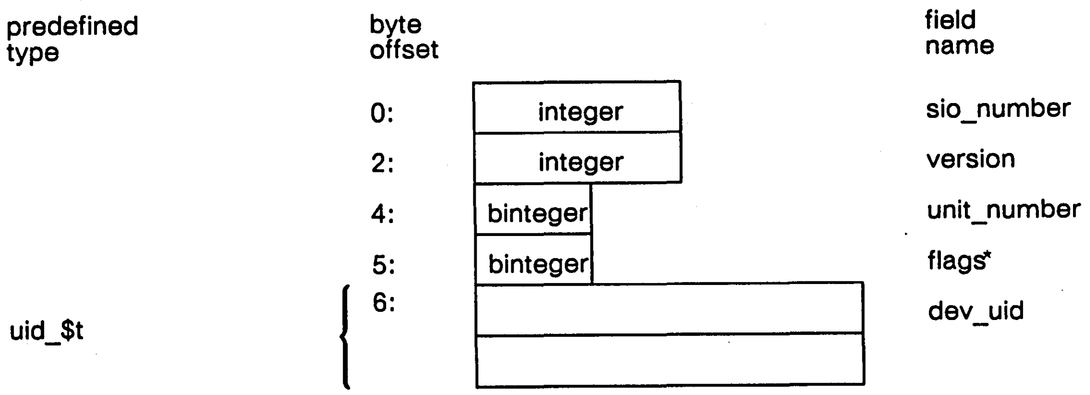
flag
For internal use only.

PBU_$DDF_PTR_T

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

PBU_$DDF_T

A Device Descriptor File (DDF). The diagram below illustrates the PBU_$DDF_T data type:



*See the following "Field Description" 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 containing Boolean values indicating device attributes. The following table lists the bit numbers within the mask, the record field names, and a sort description of each attribute:

Bit #	Field Name	Description
Bit 0	large	20–bit controller
Bit 1	share	Memory–mapped controller mapped in global address space
Bit 2	vme	VME device
Bit 3	sys	Reserved
Bit 4	debug	User debug flag
Bit 5	at	AT–compatible device
Bit 6	multiple	Driver supports multiple devices
Bit 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 clean–up routine.

int_list
Interrupt request level and name.

csr_page_iova
Address of device CSR page.

stack_size
Size (in bytes) of interrupt stack.

Field Description (cont.)

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 VME device CSR page.

memory_iova2
Memory-mapped controller base.

memory_size2
Memory-mapped controller memory size.

dma_channel
AT-compatible device channel number.

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

int_list2
Interrupt request level and name (VME and AT-compatible devices).

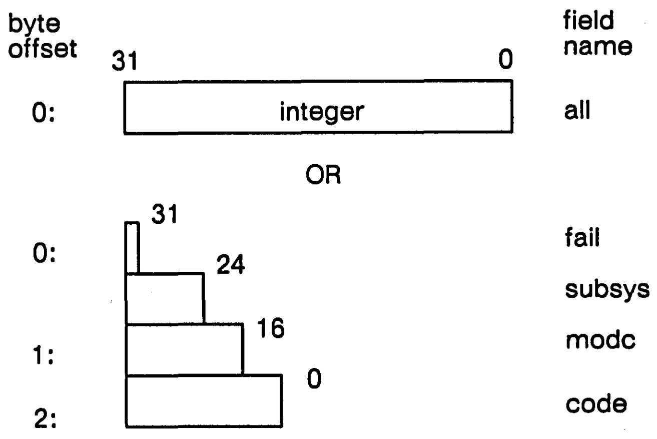
PBU_$DMA_CHANNEL_T	A 2-byte integer. The DMA channel number used by AT-compatible devices. Possible values are integers between 0 and 7.

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 AT-compatible controller reads processor memory. PBU_DMA_WRITE Processor memory writes to the AT-compatible controller.
PBU_$DMA_OPTS_T	A 2-byte integer. Specifies various DMA modes on the AT-compatible bus. Any combination of the following predefined values: PBU_DMA_ADR_DECR DMA hardware decrements the address to or from which data is transferred. The default is to increment. PBU_DMA_AUTO_INIT DMA hardware reinitializes itself after completing data transfer. PBU_DMA_CASCADE Sets DMA channel in cascade mode; use with devices that can request bus mastership doing DMA with their own DMA hardware. PBU_DMA_EXT_MEM DMA to AT-compatible or XT-compatible extension memory.
PBU_$GET_EC_KEY_T	A 2-byte integer. Specifies the eventcount to get. Currently, only the following predefined value is supported: PBU_$GET_DEVICE_EC Get device EC.
PBU_$INTERRUPT_FLAGS_T	A 2-byte integer. Flags returned from the device driver's interrupt routine specifying actions that 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_$IOVA_T	A 2-byte integer. A physical address on the I/O bus.
PBU2_$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_$PA_LIST_T	An array of up to 64 UNIV_PTRs. A list of physical addresses locating the buffer in processor memory.
PBU_$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 diagram below illustrates the 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 diagram below illustrates the UID_$T data type:

byte offset		field name
0:	integer32	high
4:	integer32	low

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 GPI/O Procedures and Functions

The GPI/O calls described in this section are listed in Table B-1, along with the type of bus each call supports.

Table B-1. GPI/O Procedures and Functions

GPI/O Call	Supported Bus
PBU_$ACQUIRE	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU_$ADVANCE_EC	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU_$ALLOCATE_EC	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU_$ALLOCATE_MAP	MULTIBUS
PBU_$CONTROL	MULTIBUS
PBU_$DEVICE_INTERRUPTING	MULTIBUS and AT-Compatible Bus
PBU_$DISABLE_DEVICE	MULTIBUS and AT-Compatible Bus
PBU_$DMA_START	AT-Compatible Bus
PBU_$DMA_STOP	AT-Compatible Bus
PBU_$ENABLE_DEVICE	MULTIBUS and AT-Compatible Bus
PBU_$FREE_MAP	MULTIBUS
PBU_$GET_EC	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU_$MAP	MULTIBUS
PBU_$MAP_CONTROLLER	MULTIBUS
PBU_$MEM_PTR	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU_$READ_CSR	MULTIBUS and VMEbus
PBU_$RELEASE	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU_$RELEASE_EC	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU_$UNMAP	MULTIBUS
PBU_$UNMAP_CONTROLLER	MULTIBUS
PBU_$UNWIRE	MULTIBUS
PBU_$WAIT	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU_$WIRE	MULTIBUS

Table B-1 (cont.). GPI/O Procedures and Functions

GPI/O Call	Supported Bus
PBU_$WIRE_SPECIAL	AT-Compatible Bus and VMEbus
PBU_$WRITE_CSR	MULTIBUS and VMEbus
PBU2_$ALLOCATE_MAP	MULTIBUS
PBU2_$FREE_MAP	MULTIBUS
PBU2_$MAP	MULTIBUS
PBU2_$MAP_CONTROLLER	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU2_$UNMAP	MULTIBUS
PBU2_$UNMAP_CONTROLLER	MULTIBUS, VMEbus, and AT-Compatible Bus
PBU2_$UNWIRE	MULTIBUS and AT-compatible Bus
PBU2_$WIRE	MULTIBUS, VMEbus, and AT-Compatible Bus

PBU_$ACQUIRE—Acquires control of a peripheral device.

FORMAT

PBU_$ACQUIRE (pathname, namelen, debug_flag, unit, status)

INPUT PARAMETERS

pathname	The pathname of the DDF for the device to be acquired. Specify this parameter as an array of characters.
namelen	The length in characters of the specified pathname. This is a 2-byte Pascal integer or a C unsigned short integer.
debug_flag	A Boolean value that indicates whether load information is printed. See the AQDEV command in Appendix A.

OUTPUT PARAMETERS

unit	The unit number in PBU_$UNIT_T format for use in subsequent calls to PBU routines.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$ACQUIRE 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:

- Locates the device driver routines and maps them into user-process address space.
- Wires the interrupt stack and associated interrupt code and data.
- Maps the CSR page for the device into user-process address space.

Normally, the AQDEV command calls PBU_$ACQUIRE, but user-written routines can also call it. However, PBU_$ACQUIRE cannot be called directly by device drivers or by application programs.

PBU_$ADVANCE_EC—Advances an eventcount.

FORMAT

PBU_$ADVANCE_EC (unit, ptr, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or C unsigned short integer.
ptr	The eventcount pointer, in EC2_PTR_T format, returned from the GPI/O call PBU_$ALLOCATE_EC. This is a 4-byte integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$ADVANCE_EC advances an eventcount from a special pool of eventcounts in wired memory. It enables the interrupt handler of a shared driver to selectively advance a particular eventcount based on the type of interrupt. See also the descriptions of PBU_$ALLOCATE_EC and PBU_$RELEASE_EC as well as the discussion of shared drivers in Chapter 9, section 9.1.

PBU_$ALLOCATE_EC—Allocates a new eventcount.

FORMAT

ptr := PBU_$ALLOCATE_EC (unit, status)

INPUT PARAMETER

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or C unsigned short integer.

OUTPUT PARAMETERS

ptr	Return pointer to a new eventcount, in EC2_PTR_T format. This is a 4-byte integer.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$ALLOCATE_EC allocates an eventcount from a special pool of eventcounts in wired memory. It is designed for use with shared drivers occupying global memory. See also the descriptions of PBU_$ADVANCE_EC and PBU_$RELEASE_EC as well as the discussion of shared drivers in Chapter 9, section 9.1.

PBU_$ALLOCATE_MAP—Allocates an area of MULTIBUS address space.

FORMAT

returned_iova := PBU_$ALLOCATE_MAP (unit, length, force_flag, iova, status)

INPUT PARAMETERS

unit	The unit number of the device in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
length	The length in bytes of MULTIBUS address space for which an area of the I/O map is to be allocated. This is a 2-byte Pascal integer or a C unsigned short integer.
force_flag	A Boolean value that indicates whether or not a specific MULTIBUS address is to be assigned. For C programs, refer to Appendix C, subsection C.2.4 for information about using Boolean values in C.
iova	If the force_flag parameter is true, the MULTIBUS address in PBU_$IOVA_T format to be assigned as the starting address of the portion of MULTIBUS address space to be allocated.

OUTPUT PARAMETERS

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.

DESCRIPTION

PBU_$ALLOCATE_MAP 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.

PBU_$CONTROL—Specifies mapping controls.

FORMAT

PBU_$CONTROL (unit, opts, old_opts, reserved, status)

INPUT PARAMETERS

unit	The unit number of the device in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.

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, i.e., 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.

reserved Reserved for future use; pass in 0.

OUTPUT PARAMETERS

old_opts	The previous setting of the options in PBU_$OPTS_T format.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$CONTROL modifies the byte-swapping and protection hardware on 20-bit MULTIBUS implementations. The byte-swapping options are fully described in Chapter 1, section 1.4.

PBU_$DEVICE_INTERRUPTING—Checks for device interrupts.

FORMAT

boolean := PBU_$DEVICE_INTERRUPTING (unit, status)

INPUT PARAMETER

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETERS

boolean	A value that indicates (if true) that the device's interrupt line is asserted. For C programs, refer to Appendix C, subsection C.2.4 for information about using Boolean values in C.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$DEVICE_INTERRUPTING reads the current state of the device's interrupt request line. This routine can be called from a user-written interrupt routine. Since 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.

PBU_$DEVICE_INTERRUPTING cannot be used with VME devices.

PBU_$DISABLE_DEVICE—Disables interrupts from a peripheral device.

FORMAT

PBU_$DISABLE_DEVICE (unit, status)

INPUT PARAMETER

unit	The unit number of the device in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$DISABLE_DEVICE prevents a device from requesting interrupts by setting its interrupt mask bit.

The system automatically disables interrupts from a device

- After it has been acquired
- During interrupt processing
- When the device is released

PBU_$DISABLE_DEVICE cannot be used with VME devices.

PBU_$DMA_START—Starts a DMA operation.

FORMAT

PBU_$DMA_START (unit, channel, direction, buffer, length, opts, status)

INPUT PARAMETERS

unit
The unit number of the device in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.

channel
A 2-byte Pascal integer or a C unsigned short integer, in PBU_$DMA_CHANNEL_T format. Specifies the number (0 through 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

buffer
The buffer to be mapped, specified as a universal array of characters, in PBU_$BUFFER_T format. It must be page aligned.

length
The length of the buffer in bytes. This is a 2-byte Pascal integer or a C unsigned short integer and must be greater than 0 and less than or equal to 1024.

opts
Specifies one or both 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*, Order No. 230843-002.
- 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 DMA hardware on the mother board in cascade mode so that an AT-compatible device with demand-DMA capability can use its own DMA hardware (i.e., can request bus mastership). It is a way of arbitrating for the 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 AT-compatible or XT-compatible extension memory, not processor memory.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$DMA_START and PBU_$DMA_STOP are paired functions for use with AT–compatible devices. They should surround each DMA operation, whether successful or not. PBU_$DMA_START prepares the system DMA hardware for the controller's operation. The driver must call this routine before issuing any I/O commands to the device. After PBU_$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.

PBU_$DMA_START can be called from the driver's interrupt side.

For devices that are bus masters, 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.

For addtional information on using this call, refer to Chapter 3, subsection 3.6.1.

PBU_$DMA_STOP—Stops a DMA operation.

FORMAT

resid_cnt := PBU_$DMA_STOP (unit, channel, status)

INPUT PARAMETERS

unit	The unit number of the device in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
channel	A 2-byte Pascal integer or a C unsigned short integer, in PBU_$DMA_CHANNEL_T format. Specifies the number (0 through 7) of the channel to be stopped.

OUTPUT PARAMETERS

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 always be 0.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$DMA_START and PBU_$DMA_STOP are paired functions for use with AT-compatible devices. They should surround each I/O operation, whether successful or not. PBU_$DMA_START prepares DMA hardware for the controller's operation. After the controller has completed its operation, the driver must next call PBU_$DMA_STOP to get status from DMA hardware to ensure that the hardware has 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.

PBU_$DMA_STOP can be called from the driver's interrupt side.

For addtional information on using this call, refer to Chapter 3, section 3.6.1.

PBU_$ENABLE_DEVICE—Enables interrupts from a peripheral device.

FORMAT

PBU_$ENABLE_DEVICE (unit, status)

INPUT PARAMETER

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$ENABLE_DEVICE 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.

PBU_$ENABLE_DEVICE cannot be used with VME devices.

PBU_$FREE_MAP—Releases the I/O map area previously allocated to a device.

FORMAT

PBU_$FREE_MAP (unit, status)

INPUT PARAMETER

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$FREE_MAP releases the area of the I/O map previously allocated by the GPI/O call PBU_$ALLOCATE_MAP.

PBU_$GET_EC—Retrieves the eventcount associated with a device.

FORMAT

PBU_$GET_EC (unit, key, ecp, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
key	The key that specifies which eventcount to get in PBU_$GET_EC_KEY_T format. Currently, the only value allowed is PBU_$GET_DEVICE_EC.

OUTPUT PARAMETERS

ecp	A pointer to the eventcount for the device in EC2_$PTR_T format.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$GET_EC returns an eventcount identifier that the driver or the application can place into a list of eventcount identifiers that they pass to EC2_$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 has occurred. Refer to Chapter 6, subsection 6.3.2.

PBU_$MAP—Maps an I/O buffer.

FORMAT

returned_iova := PBU_$MAP (unit, buffer, length, iova, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
buffer	The buffer to be mapped. Specify the buffer as an array of characters.
length	The length in bytes of the buffer. This is a 2-byte Pascal integer or a C unsigned short integer.
iova	A page-aligned MULTIBUS address within the I/O map area allocated by PBU_$ALLOCATE_MAP in PBU_$IOVA_T format. This is a 2-byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETERS

returned_iova	The MULTIBUS address that marks the start of the buffer in MULTIBUS address space in PBU_$IOVA_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$MAP establishes the mapping between the buffer in processor address space and MULTIBUS address space. Drivers must call this routine before using the buffer for I/O operations and only after they have called PBU_$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.

PBU_$MAP_CONTROLLER—Maps controller memory to processor address space.

FORMAT

address := PBU_$MAP_CONTROLLER (unit, iova, length, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
iova	The MULTIBUS address that marks the start of controller memory in PBU_$IOVA_T format. This is a 2-byte Pascal integer or a C unsigned short integer. The address must lie on a page boundary and must be smaller than 32K bytes.
length	The length in bytes of controller memory. This is a 2-byte Pascal integer or a C unsigned short integer. The length to be mapped must be between 0 and 32K bytes, and the sum of the length and the MULTIBUS address must be less than 32K bytes.

OUTPUT PARAMETERS

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.2.5.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$MAP_CONTROLLER maps controller memory to processor address space. Device drivers can map only one area of controller memory per device at a time. Possible errors include:

- The specified unit number is invalid (PBU_$BAD_UNIT).
- The device has not been acquired (PBU_$NOT_ACQUIRED).
- Controller memory has already been mapped (PBU_$ALREADY_MAPPED).
- The specified MULTIBUS address is larger than 32K bytes or causes the sum of length and address to exceed 32K (PBU_$BAD_IOVA).
- The specified length is not between 0 and 32K bytes or causes the sum of length and address to exceed 32K (PBU_$BAD_LEN).

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.

It should be noted that 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.

PBU_$MEM_PTR—Returns the address and length of a shared controller.

FORMAT

address := PBU_$MEM_PTR (pathname, ddf_length, length, status)

INPUT PARAMETERS

pathname	The pathname of the DDF for the shared controller. Specify this parameter as an array of characters.
ddf_length	The length in characters of the specified pathname. This is a 2-byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETERS

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, section C.2.5.
length	The length in bytes of the area for the mapped controller. This is a 4-byte Pascal integer or C unsigned long integer.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$MEM_PTR returns the address of a shared memory-mapped controller mapped in global address space to any application that wants to use it. The following example shows how to use PBU_$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
```

PBU_$READ_CSR—Reads a device's control and status register (CSR).

FORMAT

PBU_$READ_CSR (unit, csr, value, word_flag, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
csr	The control and status register to be read in universal character format (Pascal type UNIV char or C type char). Refer to Appendix C, section C.1 for more information.
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.2.4 for information about using Boolean values in C.

OUTPUT PARAMETERS

value	The result of the read, located in the low-order (right-hand) byte if a byte read was performed. This is a 2-byte Pascal integer or a C unsigned short integer.
status	Completion status in STATUS_$T format.

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 lie 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 lie within the area of processor address space to which the memory has been mapped.

PBU_$READ_CSR is typically used in the initialization routine, but other call-side routines can call it.

NOTE: Drivers for 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.

PBU_$RELEASE—Releases an acquired device.

FORMAT

PBU_$RELEASE (unit, force_flag, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
force_flag	A Boolean value that indicates whether or not the clean-up routine can abort the device release operation. If this parameter is set to true, the device is released regardless of the status returned by the clean-up routine. If this parameter is set to false, the clean-up routine can abort the release procedure by returning a nonzero status code. Upon receipt of the status, PBU_$RELEASE aborts device release and returns to its caller.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

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 clean-up 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.

PBU_$RELEASE_EC—Returns an eventcount to the system.

FORMAT

PBU_$RELEASE_EC (unit, ptr, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2–byte Pascal integer or C unsigned short integer.
ptr	Returns a pointer to a new eventcount, in EC2_PTR_T format. This is a 4–byte integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$RELEASE_EC releases an eventcount allocated by PBU_$ALLOCATE_EC to a special pool of eventcounts in wired memory. It is designed for use with shared drivers occupying global memory. See also the descriptions of PBU_$ADVANCE_EC and PBU_$ALLOCATE_EC as well as the discussion of shared drivers in Chapter 9, section 9.1.

PBU_$UNMAP—Unmaps an I/O buffer.

FORMAT

PBU_$UNMAP (unit, buffer, length, iova, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2–byte Pascal integer or a C unsigned short integer.
buffer	The buffer to unmap. Specifies the buffer as an array of characters.
length	The length in bytes of the area to be unmapped. This is a 2–byte Pascal integer or a C unsigned short integer.
iova	The MULTIBUS address that marks the start of the buffer in PBU_$IOVA_T format. This address must be the address that PBU_$MAP returned (the actual start of the buffer).

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$UNMAP unmaps the buffer from MULTIBUS address space and invalidates the I/O map for the space occupied by the buffer.

Device drivers are under not required to unmap previously mapped buffers; another call to PBU_$MAP that specifies the same area of the I/O map effectively unmaps the previously mapped buffer. PBU_$UNMAP is used primarily to protect a buffer from erroneous references by a controller.

PBU_$UNMAP can be called from interrupt–side routines.

PBU_$UNMAP_CONTROLLER—Unmaps a controller's memory from processor address space.

FORMAT

PBU_$UNMAP_CONTROLLER (unit, address, length, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2–byte Pascal integer or a C unsigned short integer.
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.2.5.
length	The length in bytes of the area to be unmapped. This is a 2–byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$UNMAP_CONTROLLER unmaps from processor address space the controller memory mapped by PBU_$MAP_CONTROLLER. The whole mapped length must be unmapped. Possible errors can include the following:

- The specified unit number is invalid. (PBU_$BAD_UNIT).
- The specified device has not been acquired (PBU_$NOT_ACQUIRED).
- Controller memory has not been mapped (PBU_$NOT_MAPPED).

PBU_$UNWIRE—Unwires an I/O buffer.

FORMAT

PBU_$UNWIRE (unit, buffer, length, modify_flag, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
buffer	The buffer to be unwired, specified as a universal array of characters.
length	The length in bytes of the buffer. This is a 2-byte Pascal integer or a C unsigned short 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, see Chapter 7, subsection 7.1.4 ("Unwiring the I/O Buffer"). For C programs, refer to Appendix C, subsection C.2.4 for information about using Boolean values in C.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$UNWIRE makes a buffer previously wired into processor memory with PBU_$WIRE available for MMU paging operations.

NOTE: Buffers that are part of a driver's interrupt side must never be unwired.

PBU_$WAIT—Waits for device interrupt or timeout.

FORMAT

index := PBU_$WAIT (unit, time-out, quit_enable, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
time-out	The length of time in milliseconds that the routine is to wait. This is a 4-byte integer (C or Pascal).
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 will 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.2.4 for information about using Boolean values.

OUTPUT PARAMETERS

index A 2-byte Pascal integer or C short integer that corresponds to the event that caused PBU_$WAIT to return, in PBU_$WAIT_INDEX_T format. Possible values are

0	= Eventcount advanced by the System Interrupt Handler
1	= Timeout
2	= Quit fault (CTRL/Q)

status Completion status in STATUS_$T format.

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 has advanced it since the last time PBU_$WAIT was called. If an advance has 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.

PBU_$WIRE—Wires an I/O buffer.

FORMAT

PBU_$WIRE (unit, buffer, length, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2–byte Pascal integer or a C unsigned short integer.
buffer	The buffer to be wired, specified as a universal array of characters.
length	The length in bytes of the buffer. This is a 2–byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$WIRE makes the buffer's pages permanently resident in processor memory in preparation for an I/O operation. Drivers must wire I/O buffers before mapping them with PBU_$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 7.1.2.

PBU_$WIRE returns an error if any page of the specified buffer has already been wired.

PBU_$WIRE_SPECIAL—Wires a buffer and returns its physical addresses.

FORMAT

PBU_$WIRE_SPECIAL (unit, opts, buffer, length, pa_list, max_cnt, cnt, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2–byte Pascal integer or a C unsigned short integer. The unit number must refer to a VME or demand–DMA AT–compatible device.
opts	Specify one of the following options in PBU_$WIRE_SPEC_T format: • PBU_$WIRED_BUFFER: Verifies that buffer is already wired and return error message if it is not. • []: Wires buffer.
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 Pascal integer or a C unsigned short integer.
max_cnt	The length (number of entries) in the pa_list array. This is a 2_byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETERS

pa_list	An array of physical addresses in PBU_$PA_LIST_T format.
cnt	The number of entries returned in PA_LIST. This is a two byte Pascal integer or a C unsigned short integer.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU_$WIRE_SPECIAL is provided for VME controllers and demand–DMA AT–compatible controllers that use physical addresses to access processor memory. (Demand–DMA controllers can request external bus mastership.) Like PBU[2]_$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.

PBU_$WRITE_CSR—Writes to a device's control and status register.

FORMAT

PBU_$WRITE_CSR (unit, csr, value, word_flag, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
csr	The control and status register to be written in universal character format (Pascal type UNIV char or C type char). Refer to Appendix C, section C.1 for more information.
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 2-byte Pascal integer or a C unsigned short 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.2.4 for information about using Boolean values.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

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 lie 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 lie within the processor address space to which the memory has been mapped.

NOTE: Drivers for 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.

PBU2_$ALLOCATE_MAP—Allocates area of MULTIBUS address space for a 20-bit controller.

FORMAT

returned_iova := PBU2_$ALLOCATE_MAP (unit, length, force_flag, iova, status)

INPUT PARAMETERS

unit	The unit number of the device in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
length	The length in bytes of MULTIBUS address space for which an area of the I/O map is to be allocated. This is a 4-byte integer (in C and Pascal).
force_flag	A Boolean value that indicates whether or not a specific MULTIBUS address is to be assigned. For C programs, refer to Appendix C, subsection C.2.4 for information about using Boolean values.
iova	If the force_flag parameter is true, the MULTIBUS address in PBU2_$IOVA_T format to be assigned as the starting address of the portion of MULTIBUS address space to be allocated. This is a 4-byte integer (in C and Pascal).

OUTPUT PARAMETERS

returned_iova	The MULTIBUS address in PBU2_$IOVA_T format that marks the start of MULTIBUS address space allocated by PBU2_$ALLOCATE_MAP. This is a 4-byte integer (in C and Pascal).
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU2_$ALLOCATE_MAP reserves an area of the MULTIBUS address space for subsequent DMA transfers from a 20-bit controller. 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 for 20-bit controllers can allocate a second area of the I/O map for a device by calling PBU2_$MAP_CONTROLLER.

PBU2_$FREE_MAP—Releases I/O map area previously allocated to a 20-bit controller.

FORMAT

PBU2_$FREE_MAP (unit, status)

INPUT PARAMETER

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU2_$FREE_MAP releases the area of the I/O map previously allocated by the call PBU2_$ALLOCATE_MAP.

PBU2_$MAP—Maps an I/O buffer.

FORMAT

returned_iova := PBU2_$MAP (unit, buffer, length, iova, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
buffer	The buffer to be mapped. Specifies the buffer as an array of characters.
length	The length in bytes of the buffer. This is a 4-byte integer (in C and Pascal).
iova	A page-aligned MULTIBUS address within the I/O map area allocated by PBU2_$ALLOCATE_MAP in PBU2_$IOVA_T format. This is a 4-byte integer (in C and Pascal).

OUTPUT PARAMETERS

returned_iova	The MULTIBUS address that marks the start of the buffer in MULTIBUS address space in PBU2_$IOVA_T format. This is a 4-byte integer (in C and Pascal).
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU2_$MAP establishes the mapping between the buffer in processor address space and MULTIBUS address space. Drivers must call this routine before using the buffer for I/O operations and only after they have called PBU2_$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 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.

PBU2_$MAP_CONTROLLER—Maps 20-bit MULTIBUS, AT-compatible, or VME controller memory to processor address space.

FORMAT

address := PBU2_$MAP_CONTROLLER (unit, iova, length, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
iova	The bus address that marks the start of controller memory in PBU2_$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).

OUTPUT PARAMETERS

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.2.5.
status	Completion status in STATUS_$T format.

DESCRIPTION

PBU2_$MAP_CONTROLLER maps 20-bit MULTIBUS, AT-compatible, or VME controller memory to processor address space. Device drivers can map only one area of controller memory per device at a time. Possible errors include:

- The specified unit number is invalid (PBU_$BAD_UNIT).
- The device has not been acquired (PBU_$NOT_ACQUIRED).
- Controller memory has already been mapped (PBU_$ALREADY_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 MULTIBUS addresses.

It should be noted that memory that is mapped with PBU2_$MAP_CONTROLLER *must* be unmapped with PBU2_$UNMAP_CONTROLLER.

Refer to Chapter 7, section 7.2 for information on referencing controller memory.

PBU2_$UNMAP—Unmaps an I/O buffer.

FORMAT

PBU2_$UNMAP (unit, buffer, length, iova, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
buffer	The buffer to unmap. Specifies the buffer as an array of characters.
length	The length in bytes of the area to be unmapped. This is a 4-byte integer (in C and Pascal).
iova	The MULTIBUS address that marks the start of the buffer in PBU2_$IOVA_T format. This address must be the address that PBU2_$MAP returned (the actual start of the buffer).

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU2_$UNMAP 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 PBU2_$MAP that specifies the same area of the I/O map effectively unmaps the previously mapped buffer. PBU2_$UNMAP is used primarily to protect a buffer from erroneous references by a controller.

PBU2_$UNMAP can be called from the interrupt side.

PBU2_$UNMAP_CONTROLLER—Unmaps a 20-bit MULTIBUS, AT-compatible, or VME controller's memory from processor address space.

FORMAT

PBU2_$UNMAP_CONTROLLER (unit, address, length, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
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.2.5.
length	The length in bytes of controller memory. This is a 4-byte Pascal integer or C unsigned long integer.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU2_$UNMAP_CONTROLLER unmaps from processor address space the controller memory mapped by PBU2_$MAP_CONTROLLER. The whole mapped length must be unmapped. Possible errors can include the following:

- The specified unit number is invalid (PBU_$BAD_UNIT).
- The specified device has not been acquired (PBU_$NOT_ACQUIRED).
- Controller memory has not been mapped (PBU_$NOT_MAPPED).

PBU2_$UNWIRE—Unwires an I/O buffer.

FORMAT

PBU2_$UNWIRE (unit, buffer, length, modify_flag, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
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, see Chapter 7, subsection 7.1.4 ("Unwiring the I/O Buffer"). For information about using Boolean values in C, refer to Appendix C, subsection C.2.4.

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU2_$UNWIRE makes a buffer previously wired into processor memory with PBU2_$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.

PBU2_$WIRE—Wires an I/O buffer.

FORMAT

PBU2_$WIRE (unit, buffer, length, status)

INPUT PARAMETERS

unit	The device unit number in PBU_$UNIT_T format. This is a 2-byte Pascal integer or a C unsigned short integer.
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).

OUTPUT PARAMETER

status	Completion status in STATUS_$T format.

DESCRIPTION

PBU2_$WIRE makes the buffer's pages permanently resident in processor memory in preparation for an I/O operation. Drivers must wire I/O buffers before mapping them with PBU2_$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 7.1.2.

PBU2_$WIRE returns an error if any page of the specified buffer has already been wired.

B.3 Error Messages

Following are possible error messages that can be returned by GPI/O 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 MULTIBUS 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_$BUS_TIMEOUT
Read/write CSR caused bus timeout (PBU_$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_$CLEANUP_ROUTINE_MISSING
Clean-up routine not in driver (PBU_$ACQUIRE).

PBU_$CSR_PAGE_IN_USE
CSR page in use.

PBU_$DDF_TOO_BIG
DDF greater than 1K byte in length.

PBU_$DEVICE_NOT_SHARED
PBU_$MEM_PTR called for non-shared memory-mapped controller (PBU_$MEM_PTR).

PBU_$DEVICE_NOT_MAPPED
Controller not mapped (PBU[2]_$UNMAP_CONTROLLER).

PBU_$DEVICE_TIMEOUT
MULTIBUS device got bus timeout.

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_$INIT_ROUTINE_MISSING
Intialization routine not in driver library (PBU_$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_$INT_LIB_TOO_BIG
Interrupt library larger than 32K bytes (PBU_$ACQUIRE).

PBU_$INT_VECTOR_IN_USE
VME interrupt vector in use (PBU_$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_$NO_ROOM
No room in I/O map (PBU[2]_$ALLOCATE_MAP).

PBU_$NO_MORE_ECS
No more eventcounts available (PBU_$ALLOCATE_EC).

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_$NOT_VME
Operation valid for VME device only (PBU_$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.

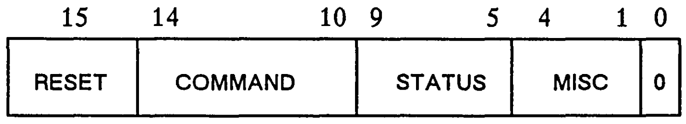
C.1 The CSR Page

In general, use data types of integer and char (for Pascal) or char (for C) 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 Pascal set, Pascal or C enumerated type, or C struct, follow these steps:

1. Declare the register as type char and integer (for Pascal) or char (for C) to ensure proper byte alignment.
2. Copy the register into local storage that contains a variant (in C, a union) 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 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 boolean of
       TRUE  :      (all : integer);
       FALSE :      (reset : boolean;
                     cmd : 0..31;
                     status : 0..31;
                     misc : 0..15;
                     mbz  : 0..0);
                     end; {of csr_t}
```

The definition of the copy of this register in C would be:

```
typedef union {
      short all;
      struct {
            unsigned int reset :1;
            unsigned int cmd :5;
            unsigned int status :5;
            unsigned int misc :4;
            unsigned int mbz :1;
}           fields;
}           csr_t #attribute[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. If you are writing your driver in C, see subsection C.2.3 for more information about using char types.

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 timeout 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* and the *DOMAIN C Library (CLIB) Reference* for complete information about our version of the C language. Use the suggestions in the following subsections to supplement the information in those manuals. An example of a driver coded in C appears in Appendix F.

C.2.1 Insert Files

The GPI/O insert file for C programmers is /SYS/INS/PBU.INS.C. 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* contains a table of corresponding data types in the various languages.

C.2.3 Type char

When you use a char type in C, you must be aware of the effects of the STD_$CALL keyword. This keyword signals the compiler that the C program and external routines exchange data according to the Domain system standard, which is equivalent to that used in Pascal and FORTRAN. The *DOMAIN C Language Reference* describes this keyword fully.

In the C insert file /SYS/INS/PBU.INS.C, GPI/O routines that take arguments of type char contain the appropriate STD_$CALL keyword. Because of the keyword, the compiler treats an array of char differently from the way it treats a pointer to a buffer containing char data. Suppose, for example, that you declare a CSR register as an array of char:

```
        char csr[8];
```

When you use csr as an argument in the routines PBU_$READ_CSR and PBU_$WRITE_CSR, the compiler passes to the routines a pointer to the characters in the array csr.

On the other hand, suppose you wish to use a pointer to a buffer of char data to declare the CSR register. As a C programmer, you might expect to use the pointer name in the invocation. Because of the effects of the STD_$CALL keyword on the routines that take csr as an argument, however, using the pointer name would make the C compiler pass a pointer to the pointer, instead of a pointer to the character or string. This would be incorrect.

Because of the effects of $STD_CALL, you must precede the pointer name with C's indirection operator (*). The indirection operator tells the compiler to pass the address of the buffer rather than the address of a pointer to the buffer. For example:

```
        char *CP
```

If you use CP instead of *CP, the compiler passes a pointer to a pointer, which is incorrect.

In summary, if you wish to use a pointer to a buffer to declare a CSR, pass the pointer as "*pointer_name" rather than just "pointer_name." The compiler then properly passes the address of the buffer to the routines PBU_$READ_CSR and PBU_$WRITE_CSR.

C.2.4 Boolean Values

Although C does not support a Boolean type, certain GPI/O 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 /SYS/INS/BASE.INS.C. Remember to include this file in your device driver program, as described in the *DOMAIN C Language Reference*.

In C, any nonzero value is defined as "true"; in Pascal, only a value of FF (hex) is "true." For "true," the GPI/O 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 GPI/O routines may not recognize as "true" the value the C compiler gives as "true."

C.2.5 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.2.6 Defining Globals

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. But in C, all globals live in their own private sections. Therefore, you need not be so concerned about where to define a global in C, since it is unrelated to the module in which the definition is made. As shown in the sample C driver in Appendix F, the data structure BMCB is globally declared in bm_global.c, and wherever it is referenced elsewhere, it is declared with the EXTERN keyword.

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 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 release notes shipped with the compiler software, or to the online version of compiler release notes. See also the *Language Reference* 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 given 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
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 To Copy or to Wire

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

```
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
```

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 msec; mapping five 1-page buffers takes 2.765 msec. 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, 512K), 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 GPI/O operations for the DN400, DN560, DN3000, DSP80, and DSP160 as of SR9.5. 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 given 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.

*Refer to *DOMAIN System Call Reference*

Table D-2. Timing for DN400, 560, 570-T, 580-T, 3000 and DSP80, 160 Workstations

Model	Operation	Times
DN400 (SR9.5)	page copy	0.000 (SVC overhead) + 1.879/page
	pbu2_$wire	0.289 (SVC overhead) + 0.509/page
	pbu2_$unwire	0.260 (SVC overhead) + 0.271/page
	pbu2_$map	0.252 (SVC overhead) + 0.150/page
	pbu2_$unmap	0.383 (SVC overhead) + 0.005/page
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
DN570-T (SR9.5.1)	page copy	0.000 (SVC overhead) + 0.346/page
	pbu2_$wire	0.061 (SVC overhead) + 0.106/page
	pbu2_$unwire	0.079 (SVC overhead) + 0.108/page
	pbu2_$map	Unsupported Call
	pbu2_$unmap	Unsupported Call
DN580-T (SR9.5.1)	page copy	0.000 (SVC overhead) + 0.328/page
	pbu2_$wire	0.067 (SVC overhead) + 0.077/page
	pbu2_$unwire	0.073 (SVC overhead) + 0.085/page
	pbu2_$map	Unsupported Call
	pbu2_$unmap	Unsupported Call
DN3000 (SR9.5)	page copy	0.000 (SVC overhead) + 0.296/page
	pbu2_$wire	0.120 (SVC overhead) + 0.136/page
	pbu2_$unwire	0.105 (SVC overhead) + 1.277/page
	pbu2_$map	Unsupported Call
	pbu2_$unmap	Unsupported Call
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

Appendix E

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 on-line 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, subsection 4.3.3, 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 E.3). An identical version of this driver coded in C is listed in Appendix F.

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

E.1 BM.PVT.PAS

BM.PVT.PAS 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 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.) }

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 }

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 }
```

E.2 BM.INS.PAS

BM.INS.PAS 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 OF.)  }

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;
```

E.3 BM_LIB.PAS

BM_LIB.PAS consists of the call-side routines that perform initialization (BM_$INIT), clean-up (BM_$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
(peripheral 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
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:

    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 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 THEN error_$print(st);
    bmcb.flags.buffer_wired := false;

END;    { of unwire_buffer }
%eject;

{    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. }

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 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 forceably 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_$acquire  to  device-dependent
initialization.  (Note: pbu_$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  command
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
immediately. }

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_$timeout;
{ If index = 2, the user typed CTRL/Q while we were waiting. Note:  the
standard system fault catcher will blast us directly back to shell command
level, so we'd never get here. But just in case the fault catcher chooses
to ignore the quit, we'll handle it. }
        2:   status.all := bm_$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;
```

E.4 BM_INT_LIB.PAS

BM_INT_LIB.PAS 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_$unmap here to unmap
the last buffer, but choose not to: if another portion of the buffer needs
to be transferred, mapping the new portion (see bm_$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 controller 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
operation.  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 transfer }
    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_$sio(bmcb.sio_status);      { call internal sio routine to start
                                     up controller }
    IF bmcb.sio_status.all <> 0 THEN BEGIN    { oops -- bm_$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
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. }

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
initialization 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;
```

Appendix F

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. It differs from the Pascal version in following the C convention of devoting each routine to a single function. Hence, this C version consists of more files than does the Pascal version. The "makefile" script in section F.10 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, subsection 4.3.3, and Figure 4–2. For additional information about the driver and the hypothetical bulk–memory controller it supports, refer to section F.1. For information about writing device drivers in C, refer to Appendix C, section C.2.

> **NOTE:** Unlike Pascal, the C programming language is casesensitive; therefore, all system procedure names (such as GPI/O routines) must be lowercase, consistently with their appearance in the system insert files. Likewise, any global names in C that are accessed by GPI/O routines must be lowercase.

The driver consists of nine files (plus the makefile):

- bm_ins.c
- bm_global.c
- bm_init.c
- bm_command.c
- bm_sio.c
- bm_wait.c
- unwire_buffer.c
- bm_int.c

- bm_cleanup.c
- makefile

F.1 bm_ins.c

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 megabyte can be transferred with one call, but since the MULTIBUS 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 portions of size BM_$BLOCK_LEN. Note that BM_$BLOCK_LEN is not the maximum possible, which is 64K bytes. The reason for not allowing 64K-byte 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 int bm_$buf_len_t;            /* bm buffer dimension */
typedef int bm_$buf_t[bm_$max_len];
typedef bm_$buf_t *bm_$buf_ptr_t;
typedef int bm_$bm_address_t;        /* address of block in bulk memory */
typedef union {
    bm_$buf_ptr_t   p;
    int             i;
} bm_$both_t;                          /* for handling buffer pointers */


/**************** ALL PRIVATE DEFINITIONS FOLLOW *****************/

/* commands 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
                                                    */

/*
 * 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 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.
 */

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.
 */

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;

    /* define communications area */

    typedef struct {
        pbu_$unit_t          pbu_unit_number; /* number of this pbu (peripheral
                                               * bus unit) device
                                               /*
        bm_$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_$iova_t          bm_iova;      /* start of our area of i/o
                                            * address space
                                            */
        bm_$both_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 called from
                                            * int side
                                            */
    bm_$both_t          io_addr;      /* address of last i/o transfer */
    bm_$buf_len_t       io_len;       /* length of last i/o transfer */
} bm_$bmcb_t;
```

F.2 bm_global.c

The bm_global.c file contains all global data for the call side of the driver.

```
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pbu.ins.c";
#include "bm.ins.c"

bm_$bmcb_t      bmcb;     /* The bulk memory control block is globally
                           * defined in this file only. All other files
                           * referencing bmcb do so using the keyword extern.
                           */
```

F.3 bm_init.c

The bm_init.c routine is called from PBU_$ACQUIRE and performs device-dependent initialization. Since this routine is called by GPI/O software, its parameters are passed in as pointers and must therefore be dereferenced with the indirection operator (*).

```
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pbu.ins.c"
#include "bm.ins.c"

/* Since this routine is called from Pascal, all parameters are passed by
 * reference.
 */

void
bm_$init(unit, ddf_ptr, csr_ptr, status)
pbu_$unit_t             *unit;            /* pbu unit number */
pbu_$ddf_ptr_t          *ddf_ptr;
pbu_$csr_page_ptr_t     *csr_ptr;
status_$t               *status;
{
    extern bm_$bmcb_t       bmcb;

    printf("unit = %d\n", *unit); /* dereference unit to print the value */

    /* Save 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 */
            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.  Because status was previously
                        /* declared as a pointer, it must be dereferenced.
                         */


    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_$allocate_map(bmcb.pbu_unit_number, /* number of
                                                            * this device
                                                            */
                bm_$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;
   }

   /*
    * We could enable interrupts from the controller here, but we'll
    * wait until we actually start an operation -- see bm_command.
    */

   bmcb.flags.b.init = 1;   /* note we're initialized */
}
```

F.4 bm_command.c

The bm_command.c routine performs argument validation, wires down the user's buffer, and calls the internal bm_sio.c routine to start the transfer.

```
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pbu.ins.c"
#include "bm.ins.c"

void
bm_command(command, buffer, len, bm_address, status)
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;
{
        extern bm_$bmcb_t       bmcb;

        /*
         * 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.i < 0) || (bmcb.bufaddr.i+len >
                                        pbu_$max_virtual_address)) {
                status->all = bm_$bad_address;
                return;
        }

        if ((bm_address < 0) || (bm_address + len > bm_$max_address)) {
                status->all = bm_$bad_bm_address;
                return;
        }

        /* Wire down the buffer. */

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

        pbu_$wire(bmcb.pbu_unit_number, buffer, bmcb.buflen, status);

        if (status->all != 0) {
                status->s.fail = 1;
                return;
        }

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

        /*
         * Buffer is all ready. Call the internal sio routine to map the
         * buffer and load the controller registers.
         */

        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_$sio(status);                /* start up the operation */
        if (status->all != 0) {
            status->fail = 1;
            unwire_buffer();
        }
        /* Enable interrupts from the bm controller. */
        pbu_$enable_device(bmcb.pbu_unit_number, *status);
}
/* These next two routines should probably be macros for maximum
 * performance.
 */
/* This routine reads a block of memory from the bulk memory device into
 * processor memory.
 */
void
bm_$read(buffer, buflen, bm_addr, s)
bm_$buf_t               buffer;
bm_$buf_len_t           buflen;
bm_$bm_address_t        bm_addr;
status_$t               *s;
{
    bm_command(BM_READ_CMD, buffer, buflen, bm_addr, s);
}
/* This routine writes a block of processor memory out to the bulk memory
 * device.
 */
void
bm_$write(buffer, buflen, bm_addr, s)
bm_$buf_t               buffer;
bm_$buf_len_t           buflen;
bm_$bm_address_t        bm_addr;
status_$t               *s;
{
    bm_command(BM_WRITE_CMD, buffer, buflen, bm_addr, s);
}
```

F.5 bm_sio.c

The bm_sio.c routine starts the I/O operations for the bulk memory device. It maps (a part of) the buffer and loads the controller registers to start an I/O operation.

```
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pbu.ins.c"
#include "bm.ins.c"
void
bm_$sio(status)
status_$t       *status;
{
    extern bm_$bmcb_t       bmcb;
    bmcb.csr_ptr.c->bm_address = bmcb.bm_address; /* tell controller
                                                    * where to start in
                                                    * bulk memory
                                                    */
    /*
     * If the buffer length is less than or equal to bm_$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_$map(bmcb.pbu_unit_number,    /* number of
                                                              * device
                                                              */
                 bmcb.bufaddr,   /* virtual address of buffer */
                 bmcb.io_len,    /* length of buffer */
                 bmcb.bm_iova,   /* iova we got from pbu_$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
                                              */
}
```

F.6 bm_wait.c

The bm_wait.c routine waits for the completion of a read or write operation. Note that for bm_wait.c a time-out value of 0 means wait forever. This is unlike PBU_$WAIT, for which a time-out value of 0 means return immediately.

```
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pbu.ins.c"
#include "bm.ins.c"

void
bm_$wait(timeout, bm_status, rem_len, status)
short           timeout;
bm_$status_t    *bm_status;     /* controller status */
bm_$buf_len_t   *rem_len;       /* residual count */
status_$t       *status;
```

```
{
     extern bm_$bmcb_t        bmcb;

     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);

          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->fail = 1;
              return;
          }
     } else
          index = 0;   /* transfer already complete */

     switch (index) {
     case 0:
         /* the operation completed. Get ending status and length
          * transferred for caller.
          */
         bm_status->all = bmcb.status.all;
         if (bmcb.status.all == bm_$sio_error)
             *status = bmcb.sio_status;
         else if (bmcb.status.all != bm_$status_ok)
             status->all = bm_$io_error;
         *rem_len = bmcb.rem_len;   /* residual count */
         break;
     case 1:
         /* operation did not complete in time. */
         status->all = bm_$timeout;
         break;
```

```
        case 2:
            /*
             * 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 index value %d\n", index);
            break;
        }
        /* Unmap and unwire the buffer. */

        pbu_$unmap(bmcb.pbu_unit_number,
            *bmcb.bufaddr.p,        /* 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 */
    }
```

F.7 unwire_buffer.c

Th unwire_buffer.c routine unwires a buffer.

```
    #include "/sys/ins/base.ins.c"
    #include "/sys/ins/error.ins.c"
    #include "/sys/ins/pbu.ins.c"
    #include "bm.ins.c"

    void
    unwire_buffer()
    {
        extern bm_$bmcb_t       bmcb;
        status_$t               st;

        if (!bmcb.flags.b.buffer_wired)
            return;                         /* nothing to do */

            pbu_$unwire(bmcb.pbu_unit_number,
                *bmcb.bufaddr.p,
                bmcb.buflen,
                bmcb.command = BM_READ_CMD,  /* touch pages if read command */
                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 = 0;
}
```

F.8 bm_int.c

Th bm_int.c routine handles interrupts is the driver's interrupt routine.

```
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pbu.ins.c"
#include "bm.ins.c"

pbu_$interrupt_return_t
bm_$int ()
{
    extern bm_$bmcb_t        bmcb;
    extern bm_$bmcb_t        bm_$sio();

    /*
     * We're called from the internal pbu interrupt handler when an
     * interrupt is received from the device. (Note: we could call
     * pbu_$unmap here to unmap the last buffer, but choose not to: if
     * another portion of the buffer needs to be transferred, mapping the
     * new portion (see bm_$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 portion 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 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.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 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 */
    }
```

F.9 bm_cleanup.c

The bm_cleanup.c routine is called by PBU_$RELEASE when the user issues the RLDEV command. Since this routine is called by GPI/O software, its parameters are passed in as pointers and must therefore be dereferenced with the indirection operator (*).

```
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pbu.ins.c"
#include "bm.ins.c"

void
bm_$cleanup (unit, force, status)
pbu_$unit_t     *unit;
char            *force;
status_$t       *status;
{
    extern bm_$bmcb_t       bmcb;

    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 forceably reset the controller. */
    if (force)
        bmcb.csr_ptr.c->command = BM_INIT_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 */
```

F.10 makefile

The makefile script organizes the files that make up the driver into the call and interrupt libraries when the driver is bound.

```
all: bm_call.lib bm_int.lib
bm_call.lib: bm_init.bin bm_cleanup.bin bm_command.bin
      bind -allmark bm_init.bin bm_cleanup.bin bm_command.bin
 -b bm_call.lib -map -sys
bm_int.lib: bm_sio.bin bm_int.bin bm_global.bin
      bind -allmark bm_sio.bin bm_int.bin bm_global.bin /lib/pbu_int_lib
-b bm_int.lib -map -sys
bm_cleanup.bin: bm_cleanup.c bm.ins.c
      /com/cc bm_cleanup.c -ndb
bm_command.bin: bm_command.c bm.ins.c
      /com/cc bm_command.c -ndb
bm_read.bin: bm_read.c bm.ins.c
      /com/cc bm_read.c -ndb
bm_write.bin: bm_write.c bm.ins.c
      /com/cc bm_write.c -ndb
bm_sio.bin: bm_sio.c bm.ins.c
      /com/cc bm_sio.c -ndb
bm_int.bin: bm_int.c bm.ins.c
      /com/cc bm_int.c -ndb
bm_global.bin: bm_int.c bm.ins.c
      /com/cc bm_global.c -ndb

listing:
      pr -e bm.ins.c bm_global.c bm_int.c bm_sio.c | prf -s //goliath
```

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 (MULTIBUS, VMEbus, and 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**.
clean–up 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 clean–up 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/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.**
demand–DMA	The capability of certain AT–compatible devices to request external bus mastership. Such devices have on–board DMA hardware.

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 clean-up 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 (GPI/O) 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 memory to processor memory. Each entry within the I/O map maps one page of MULTIBUS 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 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.**
non-bus-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 system.
paging	Moving pages of virtual memory to and from physical memory. The MMU controls paging operations.
PBU	Peripheral bus unit, synonymous with MULTIBUS device.
PBU Manager	The collection of routines that are internal to the operating system and manage GPI/O 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

The letter *f* means "and the following page"; the letters *ff* mean "and the following pages". Entries beginning with numbers are listed first.