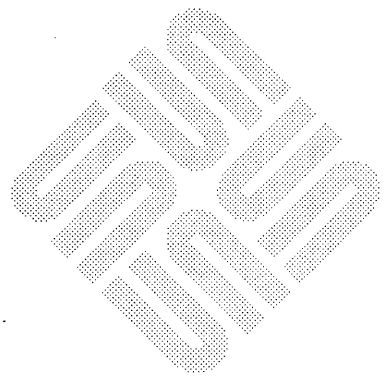
Writing Device Drivers

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1780-10
Revision A, of 24 April 1989

PART ONE: Regular Device Drivers



2

Hardware Context

Computer I/O architectures are far more dependent upon bus structure than they are upon CPU type, and device drivers, oriented as they are towards I/O, must have intimate knowledge of the bus characteristics of the machines on which they are running. For example, many Multibus machines do not support vectored interrupts [1] and thus drivers for interrupt driven devices which are intended to run on Multibus machines must provide polling facilities. Fortunately, the Sun kernel provides facilities (described in the *Other Kernel/Driver Interfaces* section of the *Overall Kernel Context* chapter) by which a driver can determine the type of the machine upon which it's running.

2.1. Multibus Machines

Multibus Memory Address Space and I/O Address Space

The MC680X0 family of processors does all its I/O via a process known as "memory mapping." What this means is that the processor sees no difference between memory and peripheral devices — all input-output operations are performed by storing data and fetching data from the same memory space. The Multibus, on the other hand, was originally designed for processors, like those of the Intel 8080 family, which have two separate address spaces. Such processors have one kind of instruction for storing data in memory or fetching data from memory (instructions such as MOV), and another, different kind of instruction (such as IN and OUT) for transferring data to or from peripheral devices. Reflecting the architecture of such processors, the Multibus has two address spaces.

Multibus memory space
: is used for memory or devices that look like memory. Many devices — commonly known as "memory mapped" devices — are designed to be accessed as memory, and drivers for such devices can "map" them into user virtual memory space and then perform device I/O by simply reading and writing the device's memory as part of normal address space. Such memory-mapped drivers tend to be quite simple, and so it's notable that devices not explicitly designed to be memory mapped can, under a restricted set of circumstances, be driven by memory mapping. The restrictions are,

[1] The Multibus itself, as it turns out, actually does support vectored interrupts, but not in a way that can reasonably be used with the MC680X0 family of processors.

however, fairly severe. Such drivers cannot, for example, have `xxioctl()` routines. See the *Mapping Devices Without Device Drivers* section of the *Driver Development Topics* manual for more details. The Sun-2 Color Board is a good example of a device that *is* designed to be memory mapped, and a listing of its driver can be found in the *Sample Driver Listings* appendix.

Multibus I/O address space
is another "space" entirely separate from normal memory. Typically used as an area to which device registers can be mapped, I/O space was originally introduced to keep such registers out of limited primary address space by providing a means of making peripherals, rather than system memory, respond to the bus whenever given I/O control lines were asserted by the CPU. (Such a setup also reduces hardware costs by keeping the number of address lines small.) Devices which have their control and status registers mapped to Multibus I/O address space are said to be "I/O mapped" devices.

The MC680X0 family, of course, no longer suffers the addressing limitations that made the dual-space architecture of the Multibus so attractive. The VMEbus, in similar regard, is no longer structured around separate "memory" and "I/O" spaces. (The term "I/O space" does continue to be used, from time to time, with reference to VMEbus-based systems and devices. Such use, however, is largely by way of analogy with Multibus systems, and it shouldn't be taken too literally).

Be aware that generic Multibus memory space can be either 20-bit or a 24-bit. (Sun normally uses 20-bit Multibus memory addresses, though when a Multibus card is installed in a VMEbus system with a VMEbus/Multibus adapter, 24-bit addresses are used). In similar regard, a generic Multibus can provide either an 8-bit or 16-bit I/O space, and Sun uses only the 16-bit Multibus I/O space. Note, however, that some older Multibus boards accept only 8-bit Multibus I/O addresses.

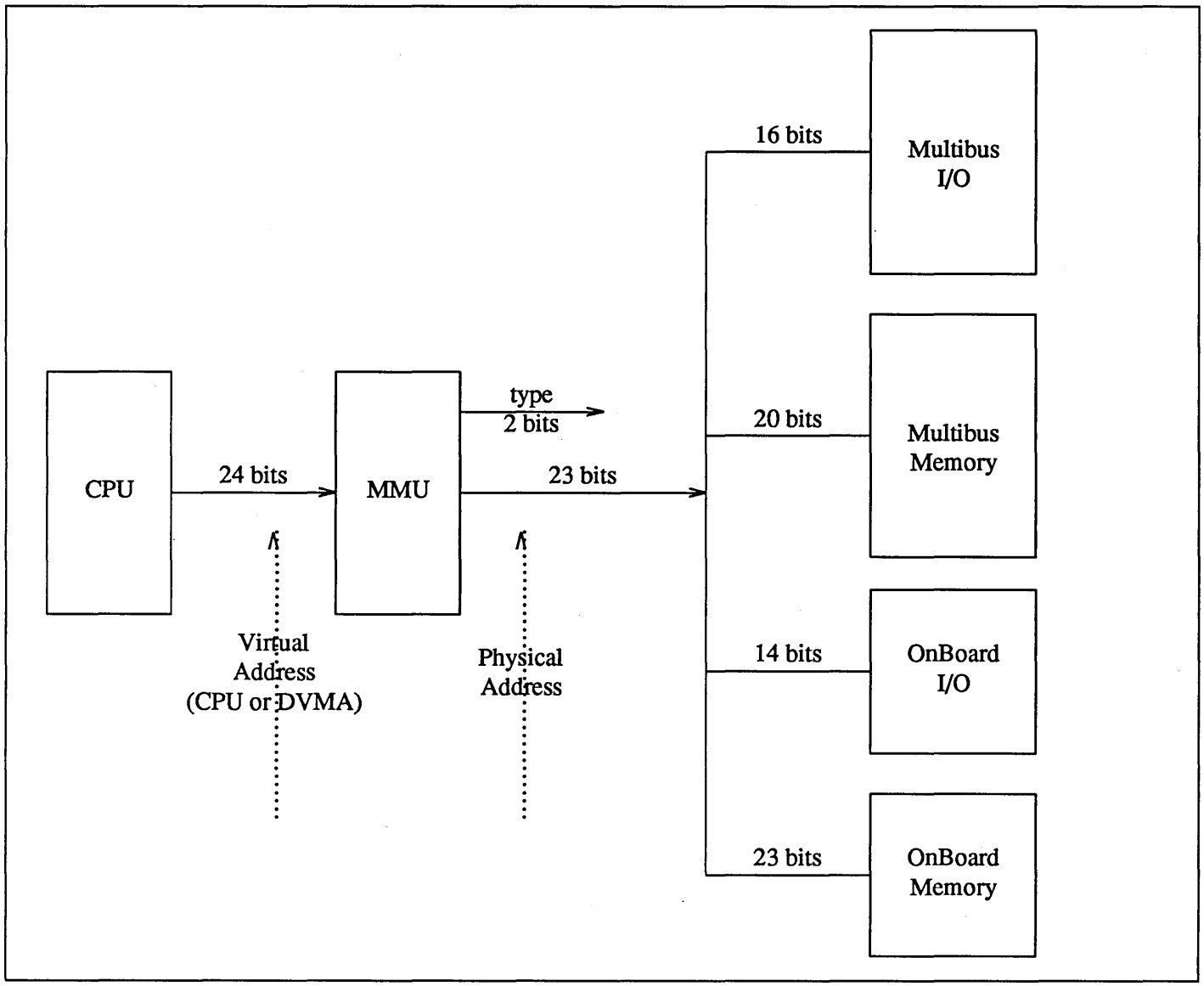
Sun Multibus systems actually have four "address spaces," corresponding to the four types of memory (each type has an identifying number associated with it, a number which is used by the MMU in computing PTE's (Page Table Entries). See the *Sun-2 Address Mapping* section of the *Driver Development Topics* chapter for details. Though you will seldom deal with the on-board address spaces, you're best off understanding what they are. The following table thus contains not only the two Multibus spaces, but the "on board" memory and I/O spaces as well. It's within these spaces, resident on the CPU board itself, that SunOS is run.

Table 2-1 *Sun-2 Multibus Memory Types*

Type	*Description*	*Address Size*	*Address Range*
0	On-Board Memory	23 bits	0x0 – 0x7FFFFF
1	On-Board I/O Space	14 bits	0x0 – 0x3FFF
2	Multibus Memory	20 bits	0x0 – 0xFFFFF
3	Multibus I/O Space	16 bits	0x0 – 0xFFFF

The following schematic view of the Sun-2 Multibus may help the driver developer to visualize the larger hardware context within which drivers operate (when running on a Sun-2 Multibus machine.)

Figure 2-1 *Sun-2 Multibus Address Spaces*



Note some significant aspects of addressing layout as indicated in this table.

- The Memory Management Unit is at the *center* of the picture, a position that reflects its importance in the addressing scheme of all Sun machines, VMEbus based as well as Multibus based. (The centrality of the MMU will become quite clear when you later set out to allocate a physical address to your device, and then examine/set it with the PROM monitor.)

- Secondly, the input address of the MMU is a 24-bit *virtual address.* It may originate with the CPU, or come from a DMA bus master; it makes no difference.
- The output is a 23-bit *physical address* and a 2-bit *address type.* The address type specifies one of the four address spaces indicated at the right of the diagram.
- The four address spaces are to the right. The space corresponding to the incoming virtual address is a function of both the address and the memory type. Note that only the top two memory spaces (Multibus I/O and Multibus Memory) are accessible by way of the Multibus; the two On-Board memory spaces are accessed directly and are seldom of concern to non-Sun driver developers.

Programs can only reference driver address spaces in terms of virtual addresses which are then translated by the MMU into physical addresses within the appropriate physical address space.

Allocation of Multibus Memory

Here are some notes about the allocation of Multibus Memory resources in the Sun system.

No devices may be assigned addresses below `0x40000` in Multibus memory space since the CPU uses these addresses for DVMA. (See the end of this chapter for a discussion of DVMA).

The table on the next page shows a map of how Multibus Memory space is laid out in the Sun system. Note that this memory map, as well as all of those that follow, is only a general guide. To be sure that you are not installing a device at a location that will put it in conflict with existing devices, it's necessary to check the configuration of the specific systems into which it will be installed. The best way to do so is to check the local config file for the physical addresses of the devices installed within the bus of interest. This will probably give you enough information, but if you still think that there may be a conflict, and if you have a Sun source license, you can check the driver header files to determine the amount of space consumed on the bus by existing devices. With the exception of the Sky board, these devices can be rearranged. Also note the possibility that your machine will have devices attached to it, and taking up bus space, even though those devices do not appear in the config file. This possibility exists because the `xxmmap()` system call can sometimes be used to drive a device without installing it in the formal sense — see the *Mapping Devices Without Device Drivers* section of the *Driver Development Topics* chapter for more details.

Table 2-2 *Sun-2 Multibus Memory Map*

Address	*Device*
0x00000 – 0x3FFFF	DVMA Space (256 Kilobytes)
0x40000 – 0x7FFFF	Sun Ethernet Memory (#1) (256 Kilobytes)
0x80000 – 0x83800	SCSI (#1) (16 Kilobytes)
0x84000 – 0x87800	SCSI (#2) (16 Kilobytes)
0x88000 – 0x8B800	Sun Ethernet Control Info (#1) (16 Kilobytes)
0x8C000 – 0x8F800	Sun Ethernet Control Info (#2) (16 Kilobytes)
0x90000 – 0x9F800	*** **FREE** *** (64 Kilobytes)
0xA0000 – 0xAF800	Sun Ethernet Memory (#2) (64 Kilobytes)
0xB0000 – 0xBF800	*** **FREE** *** (64 Kilobytes)
0xC0000 – 0xDF800	Sun Model 100/150 Frame Buffer (128 Kilobytes)
0xE0000 – 0xE1800	3COM Ethernet (#1)
0xE2000 – 0xE3800	3COM Ethernet (#2)
0xE4000 – 0xE7C00	*** **FREE** *** (16 Kilobytes)
0xE8000 – 0xF7800	Reserved for Color Devices (64 Kilobytes)
0xF8000 – 0xFF800	*** **FREE** *** (16 Kilobytes)

Allocation of Multibus I/O Space

Multibus I/O address space is specified in the config file as `mbio`. From the PROM monitor, Multibus I/O space begins at `0xEB0000`, and extends to `0xEC0000`.

Prior to Sun Release 3.0, the system made the assumption that any address lower than `0x10000` that it found in its config file was a Multibus I/O address. With current releases this is no longer true; now the bus type of every address must be explicitly given.

The following table of generic Multibus I/O usage, like the table above, is intended only as a guide.

Table 2-3 *Sun-2 Multibus I/O Map*

Address	*Device Type*
0x0040 – 0x0047	Interphase Disk Controllers
0x00A0 – 0x00A3	CPC TapeMaster Controllers
0x0200 – 0x020F	Archive Tape Drives
0x0400 – 0x047F	Ikon 10071-5 Multibus/Versatec Interface
0x0480 – 0x057F	Systech VPC-2200 Versatec/Centronics Interfaces
0x0620 – 0x069F	Systech MTI-800/1600 terminal Interface
0x2000 – 0x200F	Sky Board
0xEE40 – 0xEE4F	Xylogics 450/451 Disk Controller
0xEE60 – 0xEE6F	Xylogics 472 Multibus Tape Controller

2.2. VMEbus Machines

VMEbus machine architecture is generally more complex than Multibus machine architecture — it makes no distinction between I/O space and Memory space, but on the other hand it supports multiple address spaces. It does so for reasons of both cost and flexibility. The VMEbus was designed to be cost-effective for a range of applications. It is expensive (in terms of money, power, and board space) to provide the hardware for a full 32-bit address space. If installed devices only respond to 16-bit addresses, it makes sense to be able to put them all into a 16-bit address space and save the cost of 16-bits' worth of address decoders and the like. The 24 and 32-bit address spaces are similar compromises between cost and flexibility.

The driver writer has to understand which address space his board uses (generally, this is completely out of his/her control), and make an appropriate entry in the config file. For DMA devices, the driver writer has to know the address space that the board uses for its DMA transfers (this is usually a 32 or 24-bit space).

Sun-2 VMEbus Address Spaces

The Sun-2 VMEbus machines are based upon the 24-bit subset of the generic VMEbus — they support only a 16-bit and a 24-bit address space. These address spaces are known as `vme16d16` (16 address bits and 16 data bits respectively) and `vme24d16` (24 address bits and 16 data bits). Sun-2 VMEbus machines also contain on-board memory and I/O space, of course, but these aren't accessed by way of the VMEbus and are only barely relevant to the driver developer.

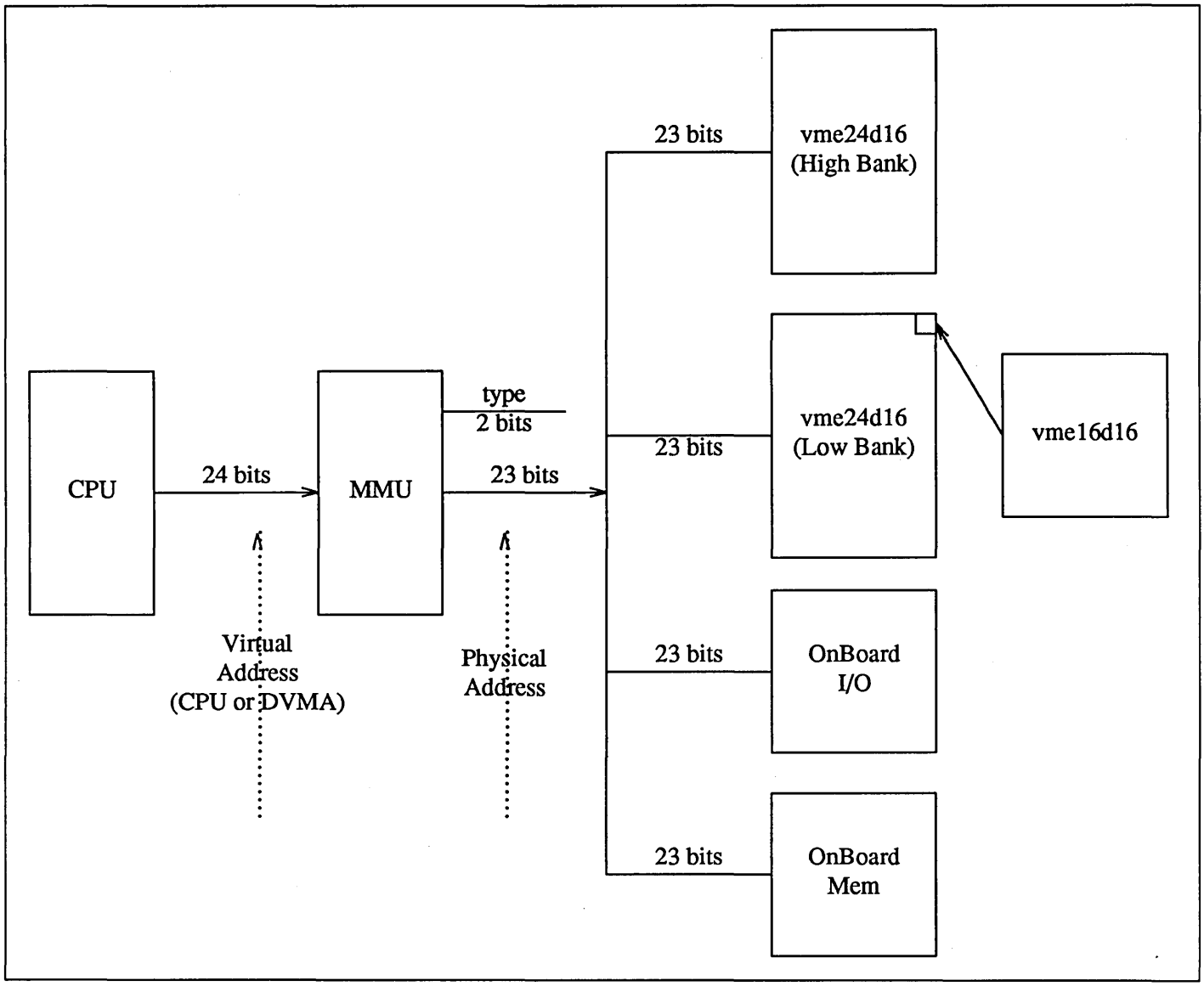
There are four types of memory on Sun-2 VMEbus machines:

Table 2-4 *Sun-2 VMEbus Memory Types*

Description	*Address Size*	*Address Range*
On-Board Memory	23 bits	0x0 - 0x7FFFFF
On-Board I/O Space	23 bits	0x0 - 0x7FFFFF
`vme24d16`	23+1 bits	0x0 - 0xFEFFFF
`vme16d16` — Stolen from top 64K of `vme24d16` (0x0 - 0xFFFF)		

The four address spaces are laid out as follows:

Figure 2-2 *Sun-2 VMEbus Address Spaces*



Note a few details:

- In all Sun-2 machines (as in Sun-3's, Sun-3x's and Sun-4's), the address input into the MMU is a virtual address, and may originate with either the CPU or a DVMA (Direct Virtual Memory Access) bus master. (See the *Sun Main-Bus DVMA* section, later in this chapter, for a discussion of DVMA).
- Unlike Sun-2 Multibus systems, in which each memory type maps cleanly to one address space, `vme24d16` maps to two different memory banks. Addresses from `0x0` to `0x7FFFFF` are "type 2" memory, while those from `0x800000` and up are "type 3". This is because Sun-2 VMEbus machines have only 23 output address bits, and this trick is necessary to generate the full range of a 24-bit address space. (See *Sun-2 Address Mapping* in the

Driver Development Topics chapter for more details).

- Multibus boards, connected to VMEbus to Multibus adapters, can be plugged into physical memory anywhere within `vme24d16` (which means that they can also be in `vme16d16`).
- The 24 bits in the `vme24d16` address space are referred to in the above table as 23+1 bits. This is because, as should be clear in the diagram above, the Sun-2 MMU outputs only the lower 23 bits of the address, and the 24th bit is actually one of the MMU's type bits.
- Note especially that `vme16d16` *is stolen from* `vme24d16`. It's selected by addresses in the form `0xFFXXXX`, that is, addresses which have the 8 high bits set.

Sun-3/Sun-3x/Sun-4 Address Spaces

Sun-3, Sun-3x and Sun-4 machines are all based on the full 32-bit VMEbus, so let's begin their discussion with a listing of the address types supported by the generic VMEbus. In all these memory references, we are referring to virtual VMEbus addresses, not Sun physical memory locations.

Table 2-5 *Generic VMEbus (Full Set)*

VMEbus-Space Name	*Address Size*	*Data Transfer Size*	*Physical Address Range*
`vme32d16`	32 bits	16 bits	`0x0 – 0xFFFFFFFF`
`vme24d16`	24 bits	16 bits	`0x0 – 0xFFFFFF`
`vme16d16`	16 bits	16 bits	`0x0 – 0xFFFF`
`vme32d32`	32 bits	32 bits	`0x0 – 0xFFFFFFFF`
`vme24d32`	24 bits	32 bits	`0x0 – 0xFFFFFF`
`vme16d32`	16 bits	32 bits	`0x0 – 0xFFFF`

Not all of these spaces are commonly used, but they are all nevertheless supported by the Sun-3 and Sun-4 lines. The following table indicates their sizes and physical address mappings.

Table 2-6 *Sun-3/Sun-4 VMEbus Address types*

Type	*Address-Space Name*	*Address Size*	*Address Range*
0	On-board Memory	32 bits	`0x0 – 0xFFFFFFFF`
1	On-board I/O	24 bits	`0x0 – 0xFFFFFF`
2	`vme32d16`	32 bits	`0x0 – 0xFEFFFFFF`
3	`vme32d32`	32 bits	`0x0 – 0xFEFFFFFF`
2	`vme24d16` — Stolen from top 16M of `vme32d16` (0x0 - 0xFEFFFF)		
2	`vme16d16` — Stolen from top 64K of `vme24d16` (0x0 - 0xFFFF)		
3	`vme24d32` — Stolen from top 16M of `vme32d32` (0x0 - 0xFEFFFF)		
3	`vme16d32` — Stolen from top 64K of `vme24d32` (0x0 - 0xFFFF)		

The Sun-3x is different than the Sun-3 and Sun-4 in that the hardware does not use page table entries (PTE's) with a type identifier to map the devices into physical memory. The Sun-3x uses absolute physical address when mapping devices.

Therefore the type field is not used as an identifier of physical address mapping. The next two tables show the virtual VME addresses and the corresponding physical addresses for the specific ranges. Note for the Sun-3x there is no `vme32d16` entry and there is a hole in the address space usage from the end of the on-board I/O area to the beginning of the vme16d16 area.

Table 2-7 *Sun-3x VMEbus Address types*

Type	*Address-Space Name*	*Address Size*	*Address Range*
0	On-board Memory	32 bits	`0x0 – 0xFFFFFFFF`
1	On-board I/O	24 bits	`0x0 – 0x00FFFFFF`
2	`vme24d16`	32 bits	`0x0 – 0xFEFFFFFF`
3	`vme32d32`	32 bits	`0x0 – 0xFEFFFFFF`
2	`vme16d16` — Stolen from top 64K of `vme24d16` (0x0 - 0xFFFF)		
3	`vme24d32` — Stolen from top 16M of `vme32d32` (0x0 - 0xFEFFFF)		
3	`vme16d32` — Stolen from top 64K of `vme24d32` (0x0 - 0xFFFF)		

Table 2-8 *Sun-3x Physical Address map*

Type†	*Address-Space Name*	*Address Size*	*Address Range*
	On-board Memory	32 bits	`0x00000000 – 0x57FFFFFF`
	On-board I/O	32 bits	`0x58000000 – 0x6EFFFFFF`
	`vme16d16`	32 bits	`0x7C000000 – 0x7C00FFFF`
	`vme16d32`	32 bits	`0x7D000000 – 0x7D00FFFF`
	`vme24d16`	32 bits	`0x7E000000 – 0x7EFFFFFF`
	`vme24d32`	32 bits	`0x7F000000 – 0x7FFFFFFF`
	`vme32d32`	32 bits	`0x80000000 – 0xFFFFFFFF`

†*Types are not used with the Sun-3x architecture.*

Sun-3/Sun-3x/Sun-4 space overlays are much more complex than those of the Sun-2, as is evident from both the tables above and the diagrams below. The principle, however, is the same — when a space overlays a larger space, its memory is stolen from that larger space and is considered by the MMU to be in the the overlaid space. One simply cannot address above `0xFF000000` in 32-bit VMEbus space or above `0x00FF0000` in 24-bit VMEbus space.

As the following diagrams illustrate, Sun-3 and Sun-4 addressing schemes are almost identical. They differ only in the size of the virtual address which — output by the CPU or a DVMA Bus Master — is fed to the MMU.

The Sun-3x, which has the MMU on the CPU chip, is a different hardware architecture than the Sun-3's and Sun-4's. There is a full 32 bit input to the MMU from the CPU, and all 32 bits are used for input to the OnBoard and vme modules. No devices use the `vme32d16` so it is not part of the memory map.

Figure 2-3 *Sun-3 VMEbus Address Spaces*

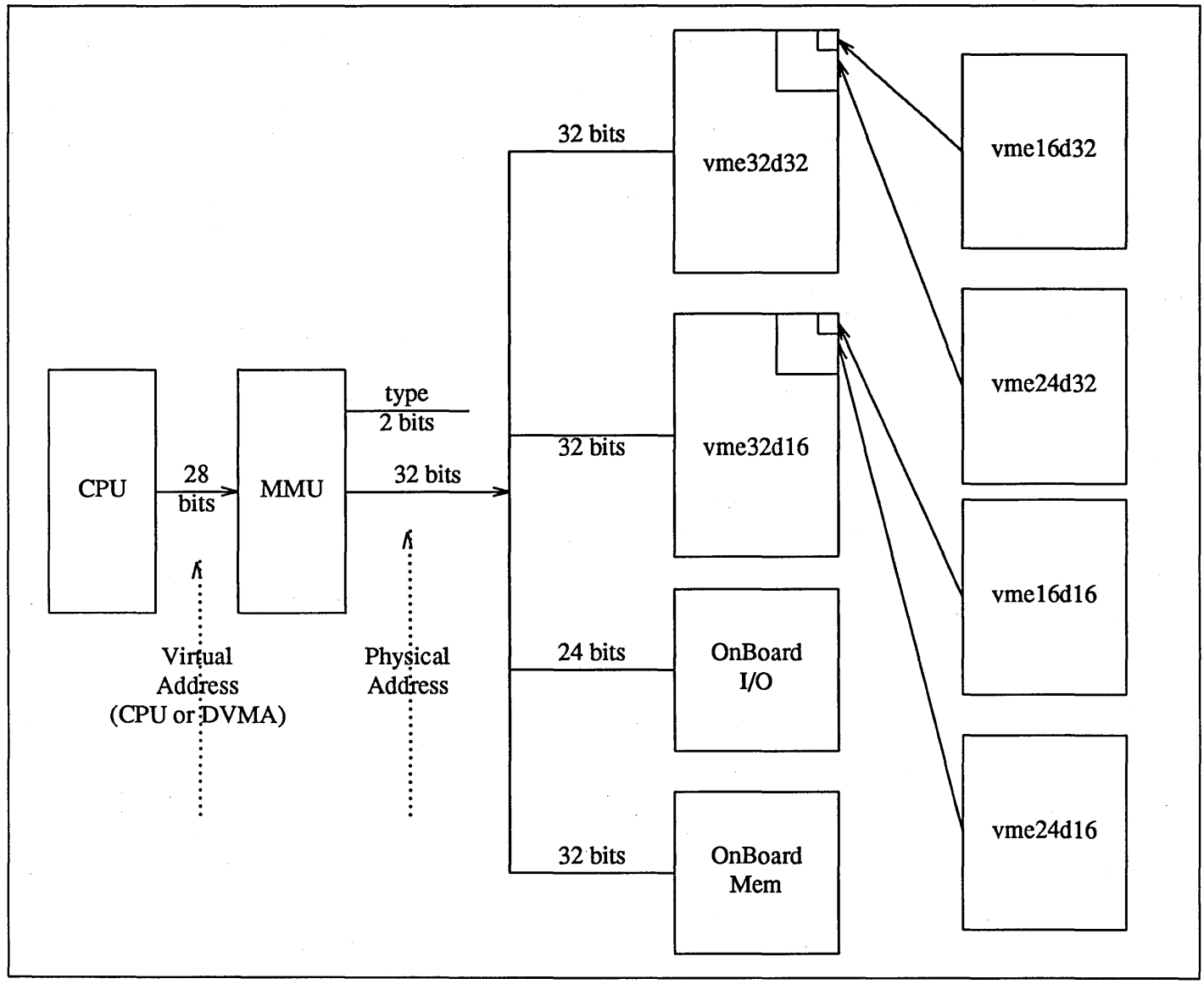


Figure 2-4 *Sun-3x VMEbus Address Spaces*

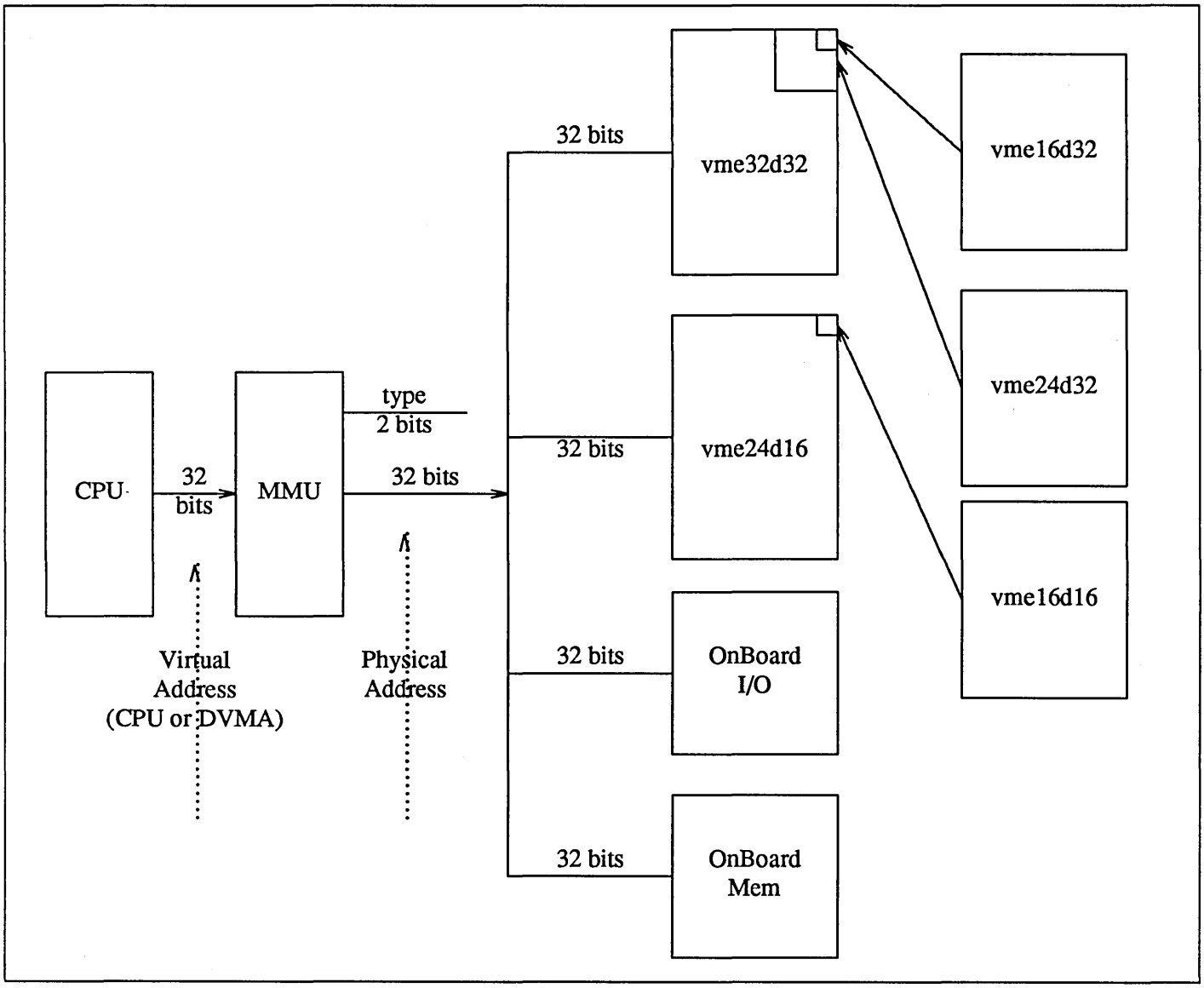
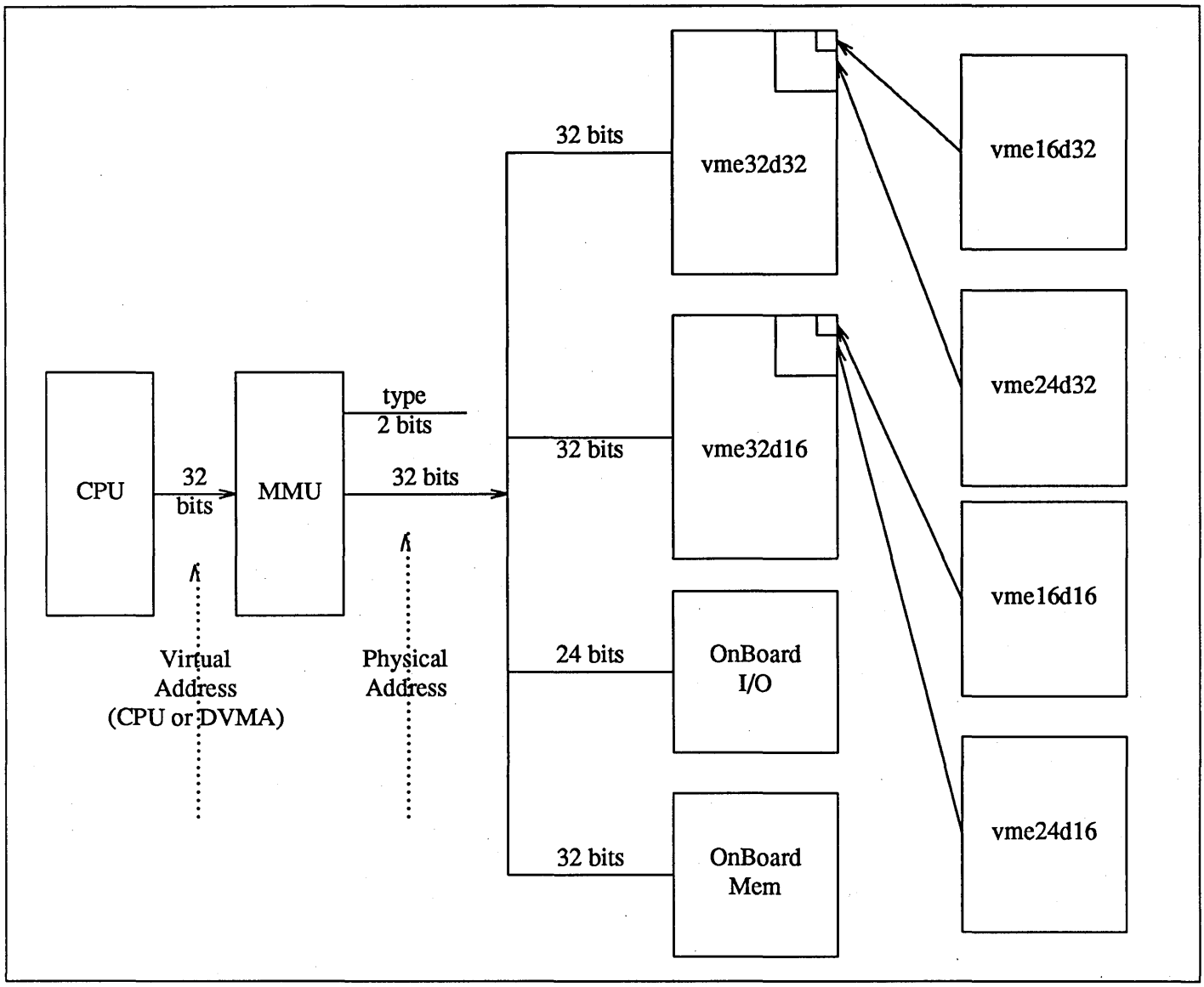


Figure 2-5 *Sun-4 VMEbus Address Spaces*



Allocation of VMEbus Memory

This section summarizes the typical use of the 16, 24 and 32-bit VMEbus address spaces by Sun devices. Note that the usages summarized here are only for the generic configuration, and there's no guarantee that they match the exact usage on your machine. They will, however, help you to decide where to attach your device. The "Allocated From" field shows whether bus space is allocated from the high end of the given range or from the low end. The idea is to keep the maximum size "hole" in the middle in case the boundary needs to be shifted later.

Table 2-9 *16-bit VMEbus Address Space Allocation*

Address Range	*Allocated From*	*Description of Use*
0x0000-0x7FFF	Low	Reserved for OEM/user devices
0x8000-0xFFFF	High	Reserved for Sun devices

16-bit VMEbus space is mapped into the topmost 64K of 24-bit VMEbus space at 0x00FF0000 to 0x00FFFFFF (on Sun-2s) or 0xFFFF0000 to 0xFFFFFFFF (on Sun-3's, Sun-3x's, and Sun-4's). Note: The Multibus/VMEbus Adapter will map the Multibus I/O addresses of Multibus cards that use Multibus I/O into the same addresses in the 16-bit VMEbus space. This may place the standard Multibus addresses for some cards into the OEM/user area in the above table. These addresses can be changed, if necessary, by physically readdressing the device and then changing its entry in the config file.

Table 2-10 *24-bit VMEbus Address Space Allocation*

Address Range	*Allocated From*	*Description of Use*
0x000000-0x0FFFFF		CPU board DVMA space
0x100000-0x1FFFFF		Reserved by Sun
0x200000-0x2FFFFF	Low	Reserved for small Sun devices
0x300000-0x3FFFFF	High	Reserved for large Sun devices
0x400000-0x7FFFFF	(Taken)	Reserved for huge Sun devices
0x800000-0xBFFFFF	High	Reserved for huge OEM/user devices
0xC00000-0xCFFFFF	Low	Reserved for large OEM/user devices
0xD00000-0xDFFFFF	High	Reserved for small OEM/user devices
0xE00000-0xEFFFFF		Multibus-to-VMEbus memory space
0xF00000-0xFEFFFF		Reserved for the Future
0xFF0000-0xFFFFFF		Stolen by 16-bit VMEbus space

Table 2-11 *32-bit VMEbus Address Space Allocation (Sun-3's, Sun-3x's, and Sun-4's)*

Address Range	*Description of Use*
0x00000000 - 0x000FFFFF	DVMA Space
0x00100000 - 0x7FFFFFFF	Reserved by Sun
0x80000000 - 0xFEFFFFFF	Reserved for OEM/user devices
0xFF000000 - 0xFFFFFFFF	Stolen by vme24d16

These same assignments apply to both 16-bit-data and 32-bit-data VMEbus accesses. Note that, at least in the GENERIC kernel, there are some Sun devices (`tm0, tm1, vpc0, vpc1 and mti0-4`) installed in the OEM/user area. It's always best to check, when choosing an installation address, that you aren't going to conflict with an already installed device.

Table 2-12 *VMEbus Address Assignments for Some Devices*

Device	*Addressing*	*Addresses Used*
VMEbus SKY Board	`vme16d16`	0x8000 - 0x8FFF (Sun-2 only)
VMEbus SCSI Board	`vme24d16`	0x200000 - 0x2007FF
VMEbus TOD Chip	`vme24d16`	0x200800 - 0x2008FF (Sun-2 only)
Graphics Processor	`vme24d16`	0x210000 - 0x210FFF
Sun-2 Color Board	`vme24d16`	0x400000 - 0x4FF7FF

The VMEbus Sky board occupies addresses `8000-8FFF` in 16-bit address space, and it requires that the high nibble of the address be '8'. Unlike other pre-installed devices, it cannot be moved.

This table is, of course, not complete. There is always a variety of devices on the bus, as can be easily determined by examining the config file. This table, however, does include the standard devices that use a significant amount of space on the VMEbus. Note that, in machines which came after the Sun-2 line, several of these devices have been replaced by on-board devices and have thus disappeared from the VMEbus address space.

The Sun VMEbus to Multibus Adapter

Multibus devices that are to be attached to VMEbus machines must be attached to a VMEbus to Multibus adapter. (The Adapter works for most, but not all, Multibus boards). An adapter can be used to take over *one and only one* chunk of `vme24d16`. However, that chunk can overlap all or part of `vme16d16` (because `vme16d16` is a proper subset of `vme24d16`). In any case, the adapter must be told how much space the board attached to it actually expects, for by default it will take over a full megabyte. Note that the Multibus Adapter supports fully vectored interrupts, and that drivers for Multibus devices attached by way of adapters need not poll, since the adapters contain switches by which Multibus devices can be assigned vectors.

Interrupt Vector Assignments

The table below shows the assignments of interrupt vectors for those devices that can supply interrupts through the VMEbus vectored interrupt interface. To pick one for your device, examine the kernel config file for an unused number in the range reserved for customer use, `0xC8` to `0xFF`.

Table 2-13 *Vectored Interrupt Assignments*

Vector Numbers	*Description*
0x40 *thru* 0x43	sc0, sc? si0, si? — SCSI Host Adapters
0x48 *thru* 0x4B	xyc0, xyc1, xyc? — Xylogics Disk Controllers
0x4C *thru* 0x5F	future disk controllers
0x60 *thru* 0x63	tm0, tm1, tm? — TapeMaster Tape Controllers
0x64 *thru* 0x67	xtc0, xtc1, xtc? — Xylogics Tape Controllers
0x68 *thru* 0c6F	future tape controllers
0x70 *thru* 0x73	ec? — 3COM Ethernet Controller
0x74 *thru* 0x77	ie0, ie1, ie? — Sun Ethernet Controller
0x78 *thru* 0x7F	future ethernet devices
0x80 *thru* 0x83	vpc? — Systech VPC-2200
0x84 *thru* 0x87	vp? — Ikon Versatec Parallel Interface
0x88 *thru* 0x8B	mti0, mti? — Systech Serial Multiplexors
0x8C *thru* 0x8F	dcp1, dcp? — SunLink Comm. Processor
0x90 *thru* 0x9F	zs0, zs1 — Sun-3/Sun-3x Terminal/Modem Controller
0xA0 *thru* 0xA3	future serial devices
0xA4 *thru* 0xA7	pc0, pc1, pc2, pc3 — SunIPC
0xA8 *thru* 0xAB	future frame buffer devices
0xAC *thru* 0xAF	future graphics processors
0xB0 *thru* 0xB3	sky0, ? — SKY Floating Point Board
0xB4 *thru* 0xB7	SunLink Channel Attach
0xB8 *thru* 0xC7	Reserved for Sun Use
0xC8 *thru* 0xFF	Reserved for Customer Use

2.3. ATbus Machines

The Intel 80386 processor handles I/O devices placed in either memory space or in I/O space. On the 80386, memory-mapped I/O provides additional programming flexibility. Any memory instruction can access any I/O port located in the memory space. For example, the MOV instruction transfers data between any register and any port. The AND, OR, and TEST instructions manipulate bits in the internal registers of a device.

On some devices, reading a register will not read back what was written. Therefore, instructions such as AND, OR, and TEST can, in some cases, produce unexpected results because the instruction reads a good location, changes it, and writes it back. See the *Other Device Peculiarities* section, ahead.

Memory-mapped I/O can use the full complement of instructions. The 16 MB memory of AT memory exists in the 4 GB physical address space of the Sun386i at `0xE000 0000`. For example, a device that, on an AT, shows up in memory at `D0 0000` will show up in the Sun386i physical memory at `0xE0D0 0000`. Virtual addresses are assigned during the autoconfiguration process.

If an I/O device is mapped into the I/O space then the IN, OUT, INS, and OUTS instructions are used to communicate to and from the device. All I/O transfers

are performed via the AL (8-bit), AX (16-bit), or EAX (32-bit) registers. The first 256 bytes of the I/O space are directly addressable. The entire 64 Kbyte I/O space is indirectly addressable through the DX register.

The Sun386i has 21 interrupt channels, but only 11 are available to devices on the AT bus. The following list of interrupt channel assignments shows all of the interrupt channels.

Table 2-14 *Interrupt Channel Assignments*

*AT Channel**	*Assignee*
3	AT Pin B25
4	AT Pin B24
5	AT Pin B23
6	Not available (system diskette)
7	Not available (parallel port)
8	SCSI
9	AT Pin B04
10	AT Pin D03
11	AT Pin D04
12	AT Pin D05
13	Not available (Ethernet)
14	AT Pin D07
15	AT Pin D06

* *Available to AT Cards*

When you add an AT card to the AT bus, you must select one of the values in the Channel column for the AT card's jumpers. For example, if you select channel 10 for a serial card, the "device" line in the config file might look as follows:

```
device  ns0 at atio ? csr 0x3f8 irq 10 priority 6
```

The Sun386i does not permit two AT cards to use the same interrupt channel.

Some cards will also use DMA and will have jumpers to select a DMA channel to use. The following list shows that DMA channels 0-3 and channel 5 are available for AT cards. Note that channel 0 and 5 can be used with 16-bit DMA devices; 1, 2, and 3 can be used only with 8-bit DMA devices. Note also that channels 4, 6, and 7 are pre-assigned.

Table 2-15 *Sun386i DMA Channel Assignments*

Channel	*Assignee*	*Size (bits)*
0	AT Bus	16
1	AT Bus	8
2	AT Bus	8
3	AT Bus	8
4	Software	Not Available
5	AT Bus	16
6	Ethernet	16
7	SCSI	16

For example, you might set up a controller that uses DMA channel 3. For this, the "controller" line in the config file might look like: this:

```
controller wds0 at atio ? csr 0x320 dmachan 3 irq 3 priority 3
```

The Sun386i does not permit two AT cards to use the same DMA channel.

In these examples, "priority" refers to the `spl` levels used in the driver. That is, the phrase "priority 3" implies that the driver uses `spl3()` to protect its critical regions.

Loadable Drivers

On Sun386i machines, device drivers can be dynamically loadable. That is, they can be attached to a system without rebuilding its kernel and without having to bring the system down and restart it. See the *Adding and Removing Loadable Drivers* section of the *Configuring the Kernel* chapter for details.

DOS and SunOS Environments

The Sun386i system supports both DOS drivers and SunOS drivers.

You can attach a DOS device driver in the standard DOS way, but it will be usable only from within the DOS environment. Usually, all you need to do is to first plug in an add-in board. Then you insert an installation diskette (which comes with the board) into Drive A> and re-boot the system. The device driver is already compiled and linked. Generally, the diskette contains programs called "INSTALL" or something similar. You execute this program by typing its name. It copies the driver file from the diskette to the hard disk. At the same time, this procedure will modify the disk's `config.sys` file.

The DOS system must be re-booted. The device driver will automatically be loaded into memory, its options will be parsed, and the driver will be initialized.

NOTE *The DOS driver on the Sun386i is running under SunOS and DOS, but the driver is unaware of this. SunOS might switch control to another task during device operation, so strict timing dependencies could fail. Real time devices, for example, may not work properly. If a peripheral and controller have strict timing requirements, their drivers should be written in the standard SunOS style. DOS drivers do not run at the elevated priority of SunOS drivers.*

SunOS drivers, of course, are parts of the system kernel. Thus the timing requirements of most devices can be met under SunOS. SunOS drivers are accessible from the DOS environment.

2.4. Hardware Peculiarities to Watch Out For

There is a variety of device peculiarities that the driver developer must be aware of. The most common of them are related to the Multibus and Multibus-based devices, but there are others as well.

Multibus Device Peculiarities

The IEEE Multibus is a source of problems for two separate reasons. The first of these, discussed immediately below, is the fact that the Multibus has a different notion of byte order than does the either Motorola MC680X0 family or the Sun SPARC processor (the reduced instruction set CPU upon which Sun-4 machines are built). The second is simply that the Multibus has been around for a long time, and thus brings with it a variety of older devices, many of which have addressing limitations and other characteristics which make for a less than perfect fit with the Sun architecture.

Multibus Byte-Ordering Issues

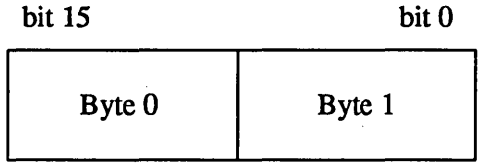
The Sun-2, Sun-3, and Sun-3x processors are members of the Motorola MC680X0 family, while Sun-4 processors are based on the SPARC CPU. All of these processors address bytes within words by what we shall call *IBM conventions* — the most significant byte of a word is stored at the lowest addressed byte of the word. The Multibus, on the other hand, uses *DEC conventions* — the least significant byte of a word is stored at the lowest address, and significance increases with address.

This class of byte-addressing conventions leads to two separate problems, with two separate solutions:

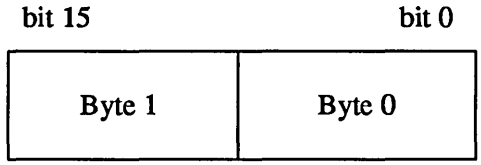
- The first problem occurs when you're moving a single *byte* across the interface between the MC680X0/SPARC and the IEEE Multibus. Because the two devices don't agree about the end of the word that the byte actually appears in, you have to change the byte address before the move — what you want to do, in effect, is move every byte to the other side of the word which it occupies — the most CPU-efficient way of doing so is to toggle the least significant bit of every byte address.
- The second problem, also related to the Multibus, is a higher level version of the first. It occurs when machine *words* with significant internal structure (or structures that contain words) are moved across the bus interface. (If you write only words, and the device uses only words, there's no problem). The Multibus byte-ordering incompatibility will cause structures to be scrambled when they're moved across the bus interface, unless the bytes within them are physically swapped first.

Here are a few pictures describing the problems in detail:

Motorola (IBM) Byte Ordering



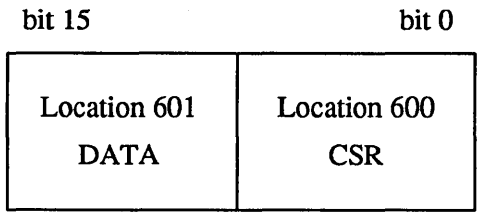
Multibus (DEC) Byte Ordering



That is, the MC680X0 and SPARC CPUs place byte 0 in bits 8 through 15 of the 16-bit word, whereas the Multibus places byte 1 in those bits. If you did everything with the CPU, or everything on the Multibus, there wouldn't be any conflict, since things would be consistent. However, as soon as you cross the boundary between them, the byte order is reversed. Thus, you have to toggle the least significant bit of the address of any *byte* destined for the Multibus — this will have the effect of swapping adjacent addresses and thus reordering the bytes.

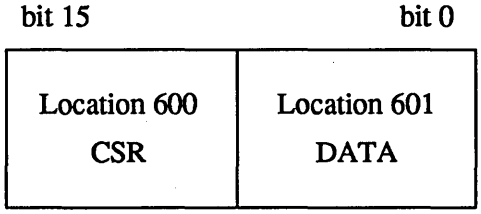
To clarify this, consider an interface for a hypothetical Multibus board containing only two 8-bit I/O registers, namely a control and status register (csr) and a data register (we actually use this design later on in our example of a simple device driver). In this board, we place the command and status register at Multibus byte location 600, and the data register at Multibus byte location 601. The Multibus picture of that device looks like this:

Hypothetical Board Registers



But the MC680X0 and SPARC processors view that device as looking like this:

Hypothetical Board Registers



so that if you were to read location 600 from the point of view of the processor, you'd really end up reading the DATA register off the Multibus instead. So, when we define the *skdevice* data structure for that board, we define it by starting with the register definition in the device manual, and then swapping bytes to take account of the expected byte swapping:

```
struct skdevice {
    char    sk_data;    /* 01: Data Register */
    char    sk_csr;     /* 00: command(w) and status(r) */
};
```

This rule (flipping the least significant bit of the address) holds good for all *byte* transfers which cross the line between the MC680X0/SPARC CPU and the Multibus.

Other Multibus-related Peculiarities

- Many Multibus device controllers are geared for the 8-bit 8080 and Z80 style chips and don't understand 16-bit data transfers. Because of this, such controllers are quite happy to place what's really a word quantity (such as a 16-bit address which must be two-byte aligned in the MC680X0) starting on an odd byte boundary. Some devices use 16-bit or 20-bit addresses (many don't know about 24-bit addresses), and it often happens that you have to chop an address into bytes by shifting and masking, and assign the halves or thirds of the address one at a time, because the device controller wants to place word-aligned quantities on odd-byte boundaries. Note also that many Multibus boards are geared for the 8086 family with its segmented address scheme. An 8086 (20-bit) address really consists of a 4-bit segment number and a 16-bit address; you usually have to deal with the 4-bit part and the 16-bit part separately. For a good example of what we're talking about here, see the code for `vp.c` in the *Sample Driver Listings* appendix to this manual.
- Although there are a myriad of vendors offering Multibus products, remember that the Multibus is a "standard" that evolved from a bus for 8-bit systems to a bus for 16-bit systems. Read vendors' product literature *carefully* (especially the fine print) when selecting a Multibus board. The memory address space of the Multibus is *supposed to be* 20 or 24 bits wide and the I/O address space of the Multibus is *supposed to be* 16 bits wide. In practice, some older boards are limited to 16 bits of address space and 8 bits

of I/O space. In particular, watch for the following addressing peculiarities:

- For a memory-mapped board, ensure that the board can actually handle a full twenty bits of addressing. Older Multibus boards often can only handle sixteen address lines. The Sun system assumes there is a 20-bit Multibus memory space out there. If the Multibus board you're talking to can only handle 16-bit addresses, it will ignore the upper four address lines, and this means that such a board "wraps around" every 64K, which means that on a Sun the addresses that such a board responds to would be replicated sixteen times through the one-megabyte address space on the Multibus. This may conflict with some other device.
- Some Sun-2 Multibus systems, notably Sun-2/170s, have a backplane structure that complicates the installation of 24-bit memory-mapped devices. The internal "bus" on these systems (often called the P2 bus) is divided into multiple segments, each mapped to a portion of the backplane slots. In such systems, 24-bit memory-mapped devices must be installed in a different segment than that used by standard Sun-2 devices. See the *Sun-2/170 Configuration Guide* for more information.
- For an I/O-mapped board (one that uses I/O registers), make sure that the board can handle 16-bit I/O addressing. Some older boards support only 8-bit I/O addressing. In our system, the address spaces of such boards would find themselves replicated every 256 bytes in the I/O address space. Trying to fit such a board into the Sun system would severely curtail the number of I/O addresses available in the system.

- Finally, watch out for boards containing PROM code that expects to find a CPU bus master with an Intel 8080, 8085, or 8086 on it. Such boards are of course useless in the Sun system.

Sun-4/SPARC Peculiarities

There are two peculiarities which are specific to machines built upon the Sun SPARC CPU (currently, just Sun-4's) which can impact device drivers. For more information about the Sun-4 machine architecture, see *Porting C, Fortran and Pascal Programs to the Sun-4*.

- The first problem is structure alignment. In MC680X0 family processors, structures are aligned on half-word boundaries, but on Sun-4's, the structure-alignment requirements are imposed by the most strictly-aligned structure components. For example, a structure containing only bytes and characters has no alignment restrictions, while a structure containing a `double` word must be constructed so as to guarantee that that this word falls on a 64-bit boundary.

  Programmers must be aware of these rules when writing drivers, for Sun-4 compilers will pad structures to enforce them, and such padding will not always be correct for structures intended to map to device registers. Also, structures must be carefully designed if drivers are to be portable across machine architectures.

- The second problem is data alignment. In MC680X0 family processors, characters are aligned on byte boundaries, while integers of all sizes are

aligned on 16-bit boundaries. In Sun-4 machines, in contrast, all quantities must be aligned on their "natural" boundaries: 16-bit half words on 16-bit boundaries, 32-bit words on 32-bit boundaries and 64-bit `double` words on 64-bit boundaries.

In normal programs, details such as these are handled by the compiler. In drivers, however, more care must be taken. SPARC (unlike the MC68010) doesn't break down 32-bit transactions into successive 16-bit transactions. Thus, there are times when 32-bit entities have to be broken down by the driver if they are to get across the bus correctly. More specifically, 32-bit or 64-bit alignment is not possible in the 16-bit VMEbus spaces, and thus 32-bit and 64-bit data access does not exist. In the 32-bit VMEbus spaces, all data paths exist.

Other Device Peculiarities

There are other device peculiarities of interest to the driver developer. These peculiarities are particularly unfortunate in that they tend to require special handling of various kinds — byte swapping, bit shuffling, timing delays, etc. — whenever the driver contacts the device. Such special handling precludes the most obvious and desirable means of interfacing the driver to the device, by mapping the device registers into a C-structure declaration and then accessing them by way of references to structure fields.

- One of the most infuriating of these peculiarities is internal sequencing logic. Devices with this strange characteristic (a vestige of microcomputer systems with extremely limited address space) map multiple internal registers to the same externally addressable address. There are various kinds of internal sequencing logic:
  - The Intel 8251A and the Signetics 2651 alternate the same external register between *two* internal mode registers. Thus, if you want to put something in the first mode register of an 8251, you do so by writing to the external register. This write will, however, have the invisible side effect of setting up the sequencing logic in the chip so that the next read/write operation refers to the alternate, or second, internal register.
  - The NEC PD7201 PCC has multiple internal data registers. To write a byte into one of them, it's necessary to first load the first (register 0) with the number of the register into which the following byte of data will go — you then send that byte of data and it goes into the specified data register. The sequencing logic then automatically sets up the chip so that the next byte sent will go into data-register 0.
  - Another chip of a similar ilk is the AMD 9513 timer. This chip has a data pointer register for pointing at the data register into which a data byte will go. When you send a byte to the data register, the pointer gets incremented. The design of the chip is such that you *can't read the pointer register to find out what's in it*!
- In fact, it's often true that device registers, when read, don't contain the same bits that were last written into them. This means that bitwise operations (like `register &= ~XX_ENABLE`) that have the side effect of

generating register reads must be done in a software copy of the device register, and then written to the real device register.

- Another problem is timing. Many chips specify that they can only be accessed every so often. The Zilog Z8530 SCC, which has a "write recovery time" of 1.6 microseconds, is an example. This means that a delay has to be enforced (with DELAY) when writing out characters with an 8530. Things can get worse, however, for there are instances when it's unclear what delays are needed, and in such cases it's left to the driver developer to determine them empirically.
- And peripheral devices can contain chips that use a byte-ordering convention different from that used by the Sun system into which they're installed. The Intel 82586, for example, supports DEC byte-ordering conventions; this makes it perfectly compatible with Multibus-based, but not VMEbus-based, Sun machines. Drivers for such peripheral devices will have to swap bytes, as indicated above, and to take care that, in doing so, they don't inadvertently reorder the bits in any control fields greater than 16 bits in length.
- Finally, there are some common interrupt-related peculiarities worth noting:
  - When a controller interrupts, it does *not* necessarily mean that both it *and* one of its slave devices are ready. Some controllers are designed in this way, but others interrupt to indicate that the controller or one of its devices *but not necessarily both* is ready.
  - Not all devices power up with interrupts disabled and then start interrupting only when told to do so.
  - While there should be a way to determine that a board has actually generated an interrupt — an attention bit or something equivalent — some devices have no such thing.
  - Finally, an interrupting board should shut off its interrupts when told to do so (and also after a bus reset). Not all do.

2.5. DMA Devices

Many device controller boards are capable of what is known as Direct Memory Access or DMA. This means that the CPU can tell the device controller for such devices the address in memory where a data transfer is to take place and the length of the data transfer, and then instruct the device controller to start the transfer. The data transfer then takes place without further intervention on the part of the processor. When it's complete, the device controller interrupts to say that the transfer is done.

Sun Main-Bus DVMA

NOTE *Sun-2, Sun-3, Sun-3x, and Sun-4 machines use Direct Virtual Memory Access (DVMA) to allow devices on the Main Bus (either a VMEbus or a Multibus) to perform DMA transfers from and to system virtual address space. In the Sun386i system, however, the Memory Management Unit (MMU) is incorporated directly on the Intel 80386 chip itself; devices need to use physical addresses. Sun386i*

DMA is discussed in the next Section.

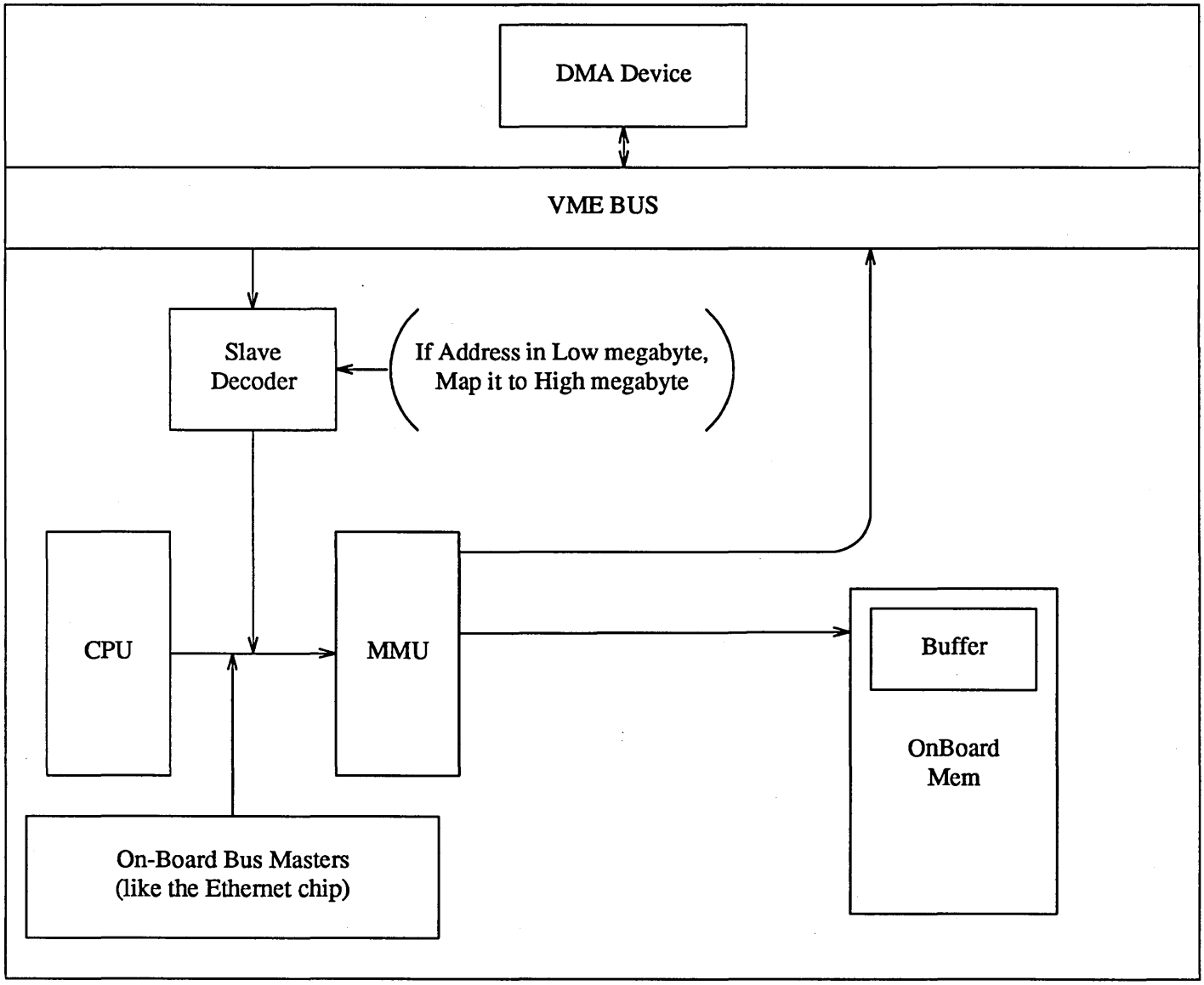
Direct Virtual Memory Access (DVMA) is a mechanism provided by the Sun Memory Management Unit to allow devices on the Main Bus (either a VMEbus or a Multibus) to perform DMA directly to Sun processor memory. It also allows Main Bus master devices to do DMA directly to Main Bus slaves without the extra step of going through processor memory. DVMA works by ensuring that the addresses used by devices are processed by the MMU, just as if they were virtual addresses generated by the CPU. This allows the system to provide the same memory protection and mapping facilities to DMA devices as it does to the system CPU (and thus to programs).

When setting up a driver to support DMA, it's necessary to know the device's DMA address size. This address size is the primary factor used in determining which of the system address spaces will host the device. Multibus devices generally have a DMA address size of 20 bits, while VMEbus devices generally have a 24 or 32-bit DMA address size.

- Since, on Sun-2 Multibus machines, DMA addresses are generally 20-bits long, the system DVMA hardware responds to the first 256K of Multibus address space (`0x0` to `0x3FFFF`). When an address in this range appears on the bus, the DVMA hardware adds `0xF00000` to it (the system places the Multibus memory address space at `0xF00000` in the system's virtual address space) and then uses the MMU to map to the location in physical memory that will be used for the data transfer.
- On Sun-2 VMEbus systems, the DVMA hardware responds to the entire lower megabyte of VMEbus address space (`0x0` to `0xFFFFF`). The system maps addresses in this range into the most significant megabyte of system virtual address space (`0xF00000` to `0xFFFFFF`).
- On the Sun-3, Sun-3x, and Sun-4 systems the DVMA hardware responds to the lowest megabyte of VMEbus address space *in both the 24-bit and 32-bit VMEbus spaces.* It maps addresses in this megabyte into the most significant megabyte of system virtual address space (`0x0FF00000` to `0xFFFFFFFF` for the Sun-3 and `0xFFF00000` to `0xFFFFFFFF` for the Sun-3x and Sun-4). The Sun-3, Sun-3x, and Sun-4 DVMA hardware use supervisor access for checking protection.

The driver writer must account for these mappings, as should be evident from the diagram below.

Figure 2-6 *System DVMA*



Devices can only make DVMA transfers in memory buffers which are from (or redundantly mapped into — see below) the low-memory areas reserved as DVMA space. The memory-management hardware will then recognize references to these areas and map them into the high megabyte of system virtual address space, an area known as DVMA space. Likewise, if a driver needs to allocate space for a DMA transfer, it must do so by way of a mechanism that guarantees its allocation from DVMA space. There are several ways of making this guarantee:

- `rmalloc()` can be used with the `iopbmap` argument. This will get a small block of memory from the beginning of the DVMA space. Such small blocks of memory are usually used for control information, and not for large

blocks of data.

- For a large buffer, the driver can statically declare a `buf` structure (which is a buffer header that contains a pointer to the data) and then use `mbsetup()` to allocate a buffer for it from DVMA space. This mechanism is primarily intended for block devices but is perfectly adaptable for use by character devices that need large DMA buffers.

When dealing with addresses which are in DVMA space, the driver must strip off the high bits by subtracting the external variable `DVMA`, which contains the address of DVMA (declared as an array of characters). `DVMA` is initialized by the system to either `0x00F00000` (for Sun-2s) or `0xFFF00000` (for Sun-3's, Sun-3x's and Sun-4's). If the driver fails to make this adjustment, the device will attempt to use a null address — in the high megabyte — and the CPU board will not respond to it.

NOTE *Addresses received by way of* `mbsetup()` *(and* `MBI_ADDR()`*) do not have to be adjusted in this fashion, as* `mbsetup()` *will have already adjusted them to be relative to the start of DVMA space.*

When the device, in turn, uses the address, the address reference comes down the bus and through a slave decoder, which adds the machine-specific offset to it to map it back into the high megabyte of system virtual memory.

Sun DMA is called DVMA because the addresses which the device uses to communicate with the kernel are virtual addresses like any others. The driver, as part of the kernel, is privy to implementation dependent information, and knows that it must chop off the high-bits of any address intended for the device. This allows the MMU to recognize the addresses destined for the Main Bus and to act accordingly. The device, however, knows nothing of this except that its buffers are mapped to the high megabyte of system virtual memory.

User processes, it should be noted, cannot do DVMA directly into their own address spaces. The kernel, however, provides a way of getting around this limitation by supporting the redundant mapping of physical memory pages into multiple virtual addresses. In this way, a page of user memory (or, for that matter, a page of kernel memory) can be mapped into DVMA space in such a way that transferred data immediately appears in (or immediately comes from) the address space of the process requesting the I/O operation. All that a driver need do to support such direct user-space DVMA is to set up the kernel page maps with the routine `mbsetup()` — the details of the mapping will then be automatically handled by the kernel.

If you wish to do DMA over the Main Bus, you must make the appropriate entries in the kernel memory map. There are two functions, `mbsetup()` and `mbrelse()`, to help with this chore.

DMA on ATbus Machines

The Sun386i uses the Intel 80386 chip. This chip has an integrated MMU, so the I/O devices cannot access the Sun MMU address-translation facility and therefore must use physical addresses to access memory directly.

To do DMA on the Sun386i, you must make certain changes in the kernel's memory map (its page tables). Use the `mbsetup()`, `dma_setup()`,

`mbrelse()`, and `dma_done()` routines to make these changes. The changes you must make to the kernel memory map are described with these routines in the *Kernel Support Routines* appendix.

5

Driver Development Topics

5.1. Installing and Checking the Device

The central processor board (CPU) of the Sun Workstation has a set of PROMs containing a program generally known as the "Monitor". (See the appropriate *PROM Commands* chapter of the *PROM User's Manual* for detailed descriptions of the monitor commands and their syntax). The monitor has three basic purposes:

1) To bring the machine up from power on, or from a hard reset (monitor `k2` command).

2) To provide an interactive tool for examining and setting memory, device registers, page tables and segment tables.

3) To boot SunOS, stand-alone programs, or the kernel debugger `kadb`.

If you simply power up your computer and attempt to use its monitor to examine your device's registers, you will likely fail. This is because, while you may have correctly installed your device (a process that includes specifying its virtual memory mapping in the config file) those mappings are SunOS specific, and don't become active until SunOS is booted. The PROM will, upon power up, map in a set of essential system devices — like the keyboard — but your device is almost certainly not among them.

When installing a new device, you will use the monitor primarily as a means of examining and setting device registers. But before even beginning the development of your driver, it's a good idea to attach your device to the system bus and use the monitor to manually probe and test it. This will give you a chance to become familiar with the details of its operation, and to ensure that it works as you expect it to.

Setting the Memory Management Unit

Upon power-up, the PROM monitor:

- Maps the beginning of on-board memory, up to 6 megabytes, to low virtual addresses starting at virtual `0x0`.
- *Sun-2 machines only.* Maps the bus spaces into virtual address space, for the purpose of supporting Multibus devices. Multibus IO space is mapped from `0xEB0000` to `0xEBFFFF` on Sun-2 Multibus machines. On Sun-2 VMEbus machines, `vme16d16` is mapped from `0xEB0000` to `0xEBFFFF` so that Multibus cards attached by way of VMEbus adapter cards can be accessed. These two address spaces, Multibus I/O and `vme16d16`, are *not*

remapped by the SunOS kernel. This means, for example, that kernel virtual address `0xEBEE40` can be used to talk to a device at `0xEE40` in Multibus IO space without setting up a mapping. (This shortcut is *only* possible for the two 16-bit Sun-2 spaces).

Later, using the autoconfiguration process, SunOS makes a pass through the config file (actually, through the `ioconf` file that was produced as output by `config` when it processed the config file). For each device, SunOS selects an unused virtual address (using an algorithm that doesn't presently concern us) and maps it into the device's physical address as specified in the config file.

SunOS then calls the *xx*`probe()` routine for each device, passing it the chosen virtual address. In this way, *xx*`probe()` is kept from having any knowledge of the physical address to which the device is mapped. *xx*`probe()` then determines whether or not the device is present. If it isn't, the virtual address can be reused.

To test a device, ignore the SunOS mappings and use the monitor to manually set the MMU to map your device registers to a known address in physical memory. Then you can use the monitor to verify its proper operation. This verification process will consist primarily of using the monitor's `O` (open a byte), `E` (open a word) and `L` (open a long word) commands to examine and modify the device's registers. Note that, in Sun-4 machines, words and long words are both 32 bits in length.

The process of setting up the device for initial testing consists of three discrete steps.

- The selection of an appropriate virtual address for the testing of the device.
- The determination of the physical address of the device, as well as the address space that it occupies.
- The use of the monitor to map the system's virtual address to the device's physical address. Detailed discussion of these three steps follow.

Since SunOS initializes the MMU in the course of its autoconfiguration process, it's possible to test a device by actually installing it, and then booting and halting SunOS. (You can halt SunOS by pressing the 'L1' and 'A' keys simultaneously, or, on a terminal console, by hitting the <BREAK> key). Having gotten to the monitor by this route, the MMU will be initialized to its SunOS run-time state. You can then use the monitor to test the device, or, if you wish, boot `kadb`. *(A hard reset — the monitor's* `k2` *command — will set the to MMU to its pre-SunOS power-up state). But while using the SunOS memory maps may occasionally be useful, it's not what you want to do during the first stages of device integration.*

Selecting a Virtual Address

First, understand that the MMU, when mapping a virtual address to a physical address, is actually mapping to a page of physical memory and an offset within that page. The low-order bits of a virtual address, those that specify the offset, *do not get mapped* — an address that is X bytes from the beginning of its virtual page will be X bytes from the beginning of whatever physical page it gets

mapped into.

The mapping mechanism is essentially the same for all Sun systems, although the details of address size and page mapping differ. This can be seen in the following diagrams:

Figure 5-1 *Sun-2 Address Mapping*

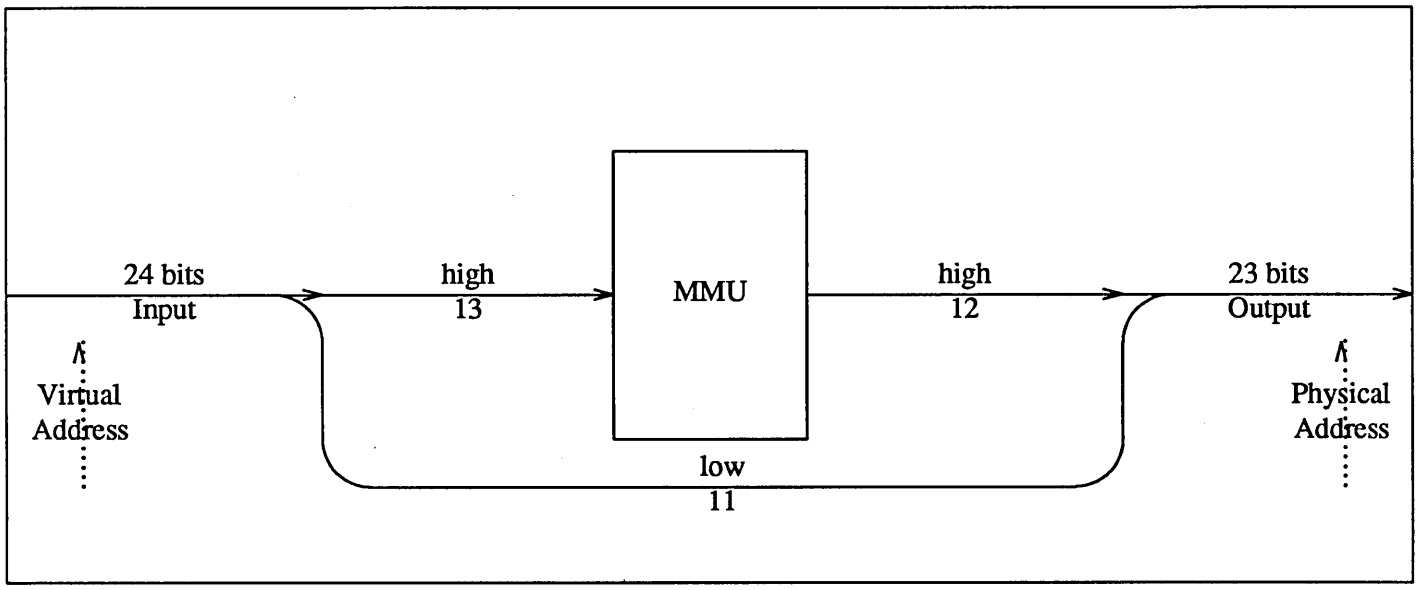


Figure 5-2 *Sun-3 Address Mapping*

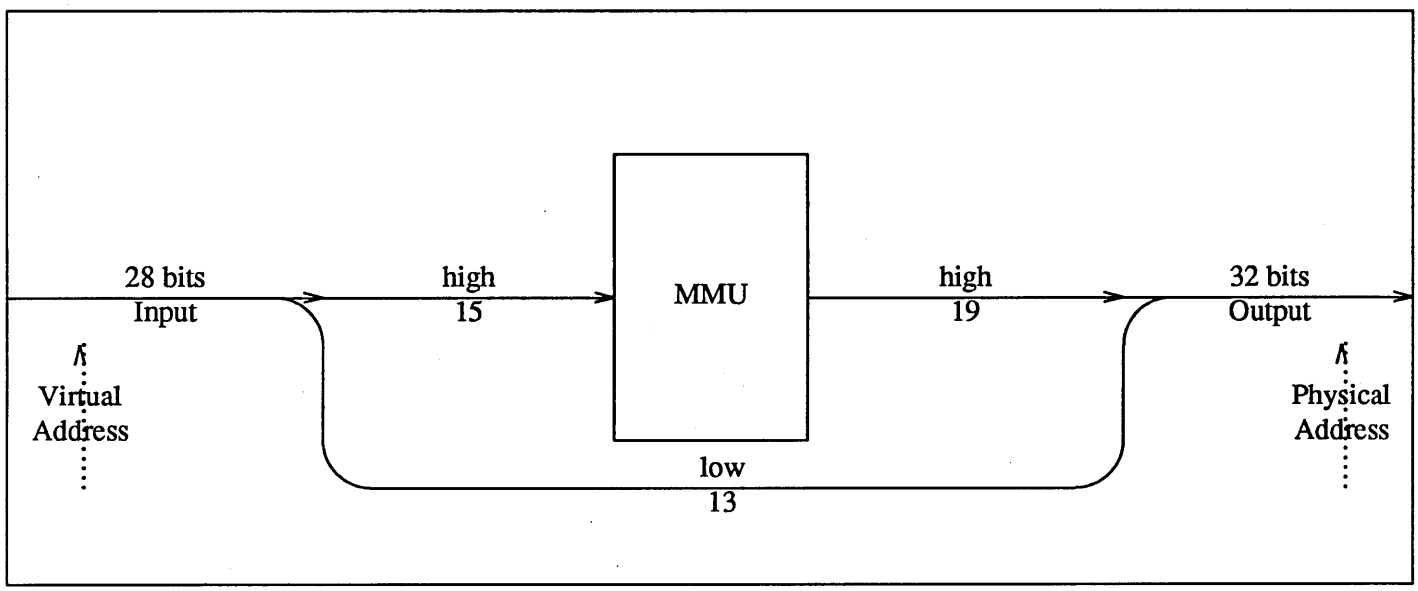


Figure 5-3 *Sun-3x/Sun-4 Address Mapping*

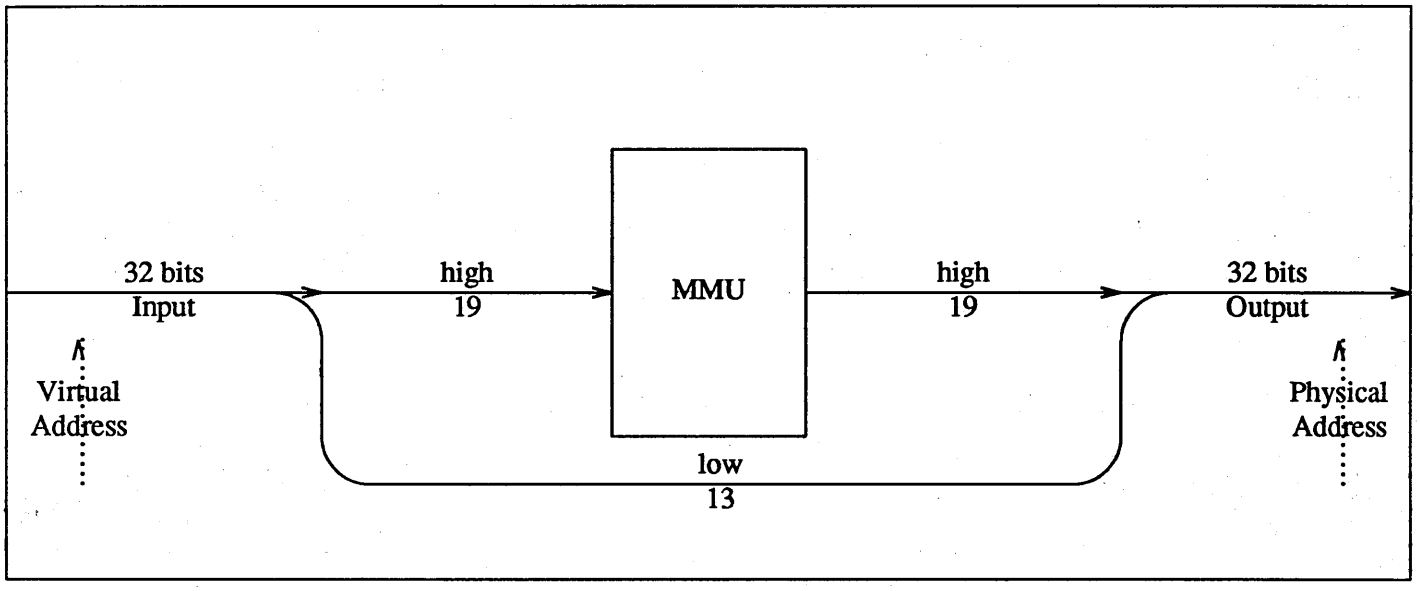
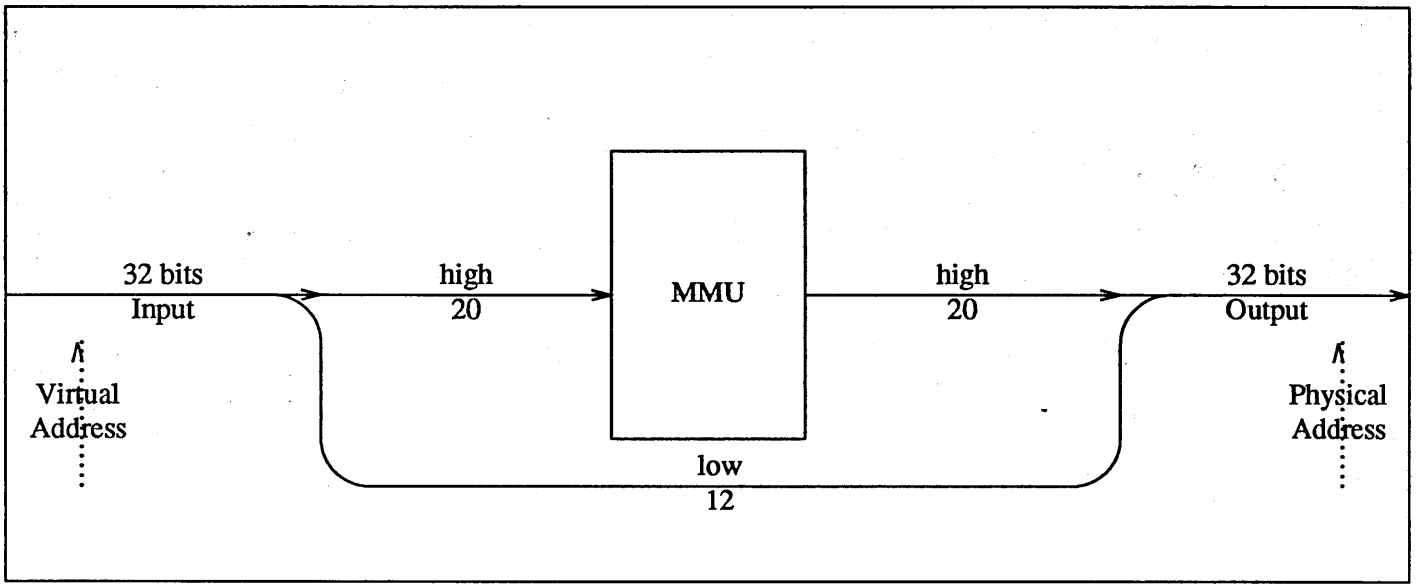


Figure 5-4 *Sun386i Address Mapping*



The easiest way to select a virtual address for PROM-monitor testing is to use one between `0x4000` and `0x100000` on Sun-2, Sun-3, Sun-3x and Sun-4 systems, or `0x20000` and `0x100000` on Sun386i systems. Addresses in these ranges are unused by the monitor in the respective Sun models, and are thus

available. (Note that these addresses, while convenient for testing, are not those that the kernel will choose when your device is finally installed).

It's most convenient to select a virtual address which has only zero's in its low-order bits. This way you select the first address in a virtual page. The low-order bits in the address you choose will remain unchanged. With `'X'` representing the unmapped low-order bits (11 for a Sun-2, 13 for a Sun-3, Sun-3x or Sun-4, 12 for a Sun386i) the test address `0x4000` is, in binary:

```
   Sun-2 :             0000 0000 0010 0XXX XXXX XXXX  (24 bits)
   Sun-3 :        0000 0000 0000 100X XXXX XXXX XXXX  (28 bits)
   Sun-3x:  0000 0000 0000 0000 100X XXXX XXXX XXXX  (32 bits)
   Sun-4 :  0000 0000 0000 0000 100X XXXX XXXX XXXX  (32 bits)
 Sun386i :  0000 0000 0000 0000 0100 XXXX XXXX XXXX  (32 bits)
```

Finding a Physical Address

Your board may be preconfigured to some address. If it is, then use that address unless it conflicts with the address of an already installed device. If it does, you will have to find an unused physical address at which you can install your device. To do so, examine the kernel config file for the system upon which you are working. Tables in the *Hardware Context* chapter show memory layouts corresponding to typical configurations, but if your system has departed at all from the norm, you will have to consult your kernel's config file (to determine where devices have been installed) and the header files for the corresponding device drivers (to determine how much space they consume on the bus).

Selecting a Virtual to Physical Mapping

When selecting a virtual to physical mapping, it's best if you understand a bit about the internals of the Memory Management Unit. The Sun-2, Sun-3, and Sun-4 all use the same proprietary MMU architecture. The Sun-3x uses the MMU that is on the same chip as the CPU. This MMU works completely different than the Sun MMU.

The following description is about the Sun MMU operation as it pertains to the Sun-2, Sun-3 and Sun-4. There is also an example of how to perform a mappings using sample numbers. The Sun-3x description follows the Sun-2/ Sun-3/Sun-4 description and includes a page mapping example.

Sun-2/Sun-3/Sun-4 Virtual to Physical Mapping

Up to this point we've only stressed that the MMU maps the top bits of the virtual address, leaving the offset bits unchanged. Now it will be necessary to explain the mapping process in more detail.

Some new concepts are necessary to discuss the details of virtual to physical memory mapping.

- The *context register (of real concern only on the Sun-2)* is a register specifying which of memory *contexts* should be used when mapping virtual addresses to physical addresses. Sun-2 and Sun-3 Context Registers contain 3 bits, and specify one of eight memory contexts; Sun-4/260 Context Registers contain four bits, and specify one of 16 memory contexts. Each SunOS *process segment* (containing either code, data or stack) is kept within a single memory context.

- Sun-3s have user and kernel address spaces in the same hardware context. That is to say, there is only one virtual address space, a portion of which is used by the kernel and the rest by user processes. Sun-4 virtual address spaces are divided into two chunks. One of them is at the top of the addressable virtual memory space and the other is at the bottom. The size of the unused space between these two spaces varies with the model — in the Sun-4/260 each of the two virtual address spaces is 512 megabytes in size, and the space between them consumes 15 Gigabytes.
- Sun-2s, on the other hand, segregate kernel and user processes into separate hardware contexts with separate address maps. Kernel processes are run in the supervisor context (context 0) and only processes in context 0 have access to the I/O devices.

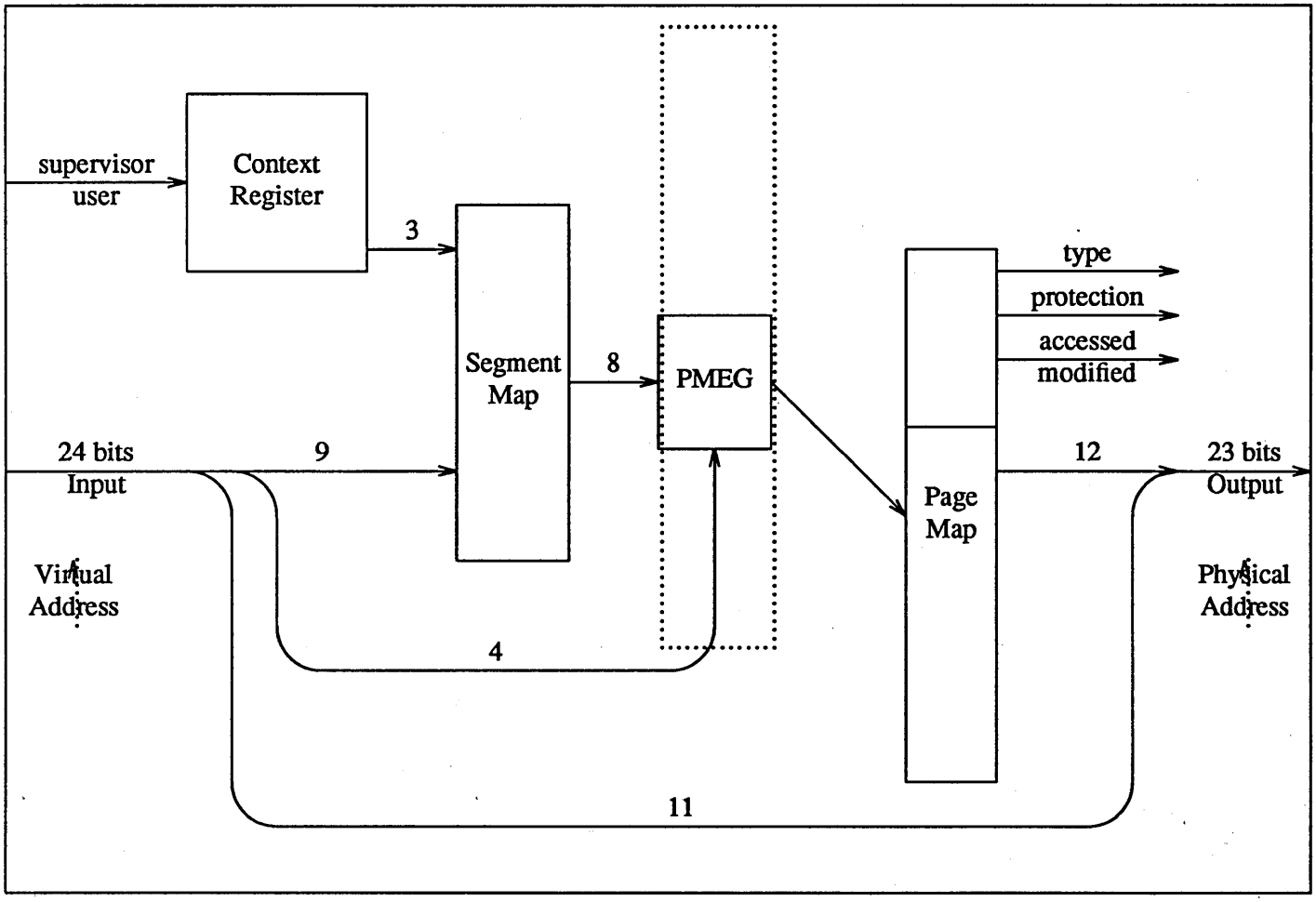
- The *segment map* is used in conjunction with the *context register* to select the *page map entry group* (PMEG) corresponding to the virtual address being mapped. The eight bits in the segment register specify one of a group of 256 PMEGs.
- Within each *page map entry group* there are 16 *page table entries.*
- The *page map* maps the PMEG returned from the segment mapping with a second subfield of the incoming virtual address to exactly specify a single *page table entry* describing the physical page within which the virtual address is mapped.
- The *page table entry* (PTE) is the final output of the MMU. A PTE specifies the physical address of a page, as well as its type (e.g., on-board memory space), protection, and the state of its *access* and *modified* flags.

Note (for Sun-2 machines only): when testing your device, it's necessary to ensure both that you are in supervisor state and that you are in context zero (the kernel context). The monitor normally initializes to supervisor state, but if you enter it by way of an abort from SunOS, you will remain in whatever context you were in at the time of the abort. To be on the safe side, begin all of your monitor sessions with the command `S5`*. This will put you into supervisor data state, where you want to be. Note one important exception to this rule: if you've* `mmap()`*'ed the device into your (user) program's address space and want to check that this worked, you must use the* `S1` *command instead of the* `S5` *command. This will cause user function codes to be used when accessing page maps and data.*

Sun-2 Address Mapping

Note the following diagram of the Sun-2 MMU:

Figure 5-5 *Sun-2 MMU*

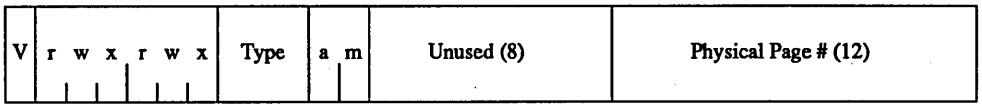


Note that:

- The lower 11 bits of the incoming virtual address are passed through the MMU without being mapped — these are the bits that specify the position within the page, regardless of whether that page is physical or virtual.
- Multiple segment maps can specify the same PMEG, and often do.
- The PTE, on the output side of the MMU, specifies a variety of kinds of status information for the specified page, as well as the top bits of its physical address.

The process of mapping a virtual to a physical address consists, in practice, of plugging the right number into the right PTE. The monitor provides a simple means of addressing the right PTE, but you will have to calculate the right value to plug into it.

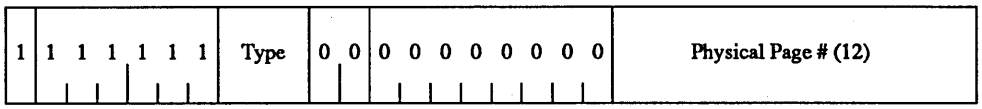
On Sun-2 systems, hardware PTEs are 32-bit numbers with the following structure.



Most of the PTEs that we will deal with will have similar structures, and so we can begin by making a "template" bit mask that we can use to construct our standard PTEs. One acceptable mask will assume values as follows:

```
V (valid) = 1
rwxrwx = 111111
(a/m) accessed/modified = 00
unused = 00000000
```

Thus, we can see that our template will be:



This gives us a mask of `0xFE000000` (if we assume that the type field is `0000`). Now, as already mentioned, there are four types of memory, represented in the PTE by values of 0, 1, 2 and 3 in the type field indicated above. (Types 0 and 1 have the same meaning in both Multibus and VMEbus machines, but types 2 and 3 do not.) Type 2 is used, on Sun-2 VMEbus machines, to designate the first 8 megabytes of the 24-bit VMEbus space — `0x0` to `0x7FFFFF` — and type 3 is used to designate the second 8 megabytes — `0x800000` to `0xFFFFFF`. (But remember that the top 64K of the 24-bit space is stolen for the 16-bit space). This use of two memory types to designate physical memory is necessary because the Sun-2 physical address size, 23 bits, is not sufficient to address all 16 megabytes of `vme24d16`.

Table 5-1 *Sun-2 PTE Masks*

Type	Description	Mask
0	On Board Memory	0xFE000000
1	On Board I/O Space	0xFE400000
2	(Multibus) Memory Space	0xFE800000
3	(Multibus) I/O Space	0xFEC00000
2	(VMEbus) VMEbus Low	0xFE800000
3	(VMEbus) VMEbus High	0xFEC00000

To determine the value which we need to plug into the PTE, we must add the appropriate mask to the appropriate physical page number, thus giving us the full 32-bit number that we need. Here, we will cease to explain details and simply give a series of rules for calculating physical page numbers.

```
If Sun-2 Multibus:

    If Multibus I/O Space, use Type-3 Template
    If Multibus Memory Space, use Type-2 Template

    Physical Page Number = Physical Address >> 11
```

```
If Sun-2 vme24d16:

    If Physical Address >= 0x800000
        Use Type-3 Template
        Physical Page Number =
            (Physical Address - 0x800000) >> 11

    If Physical Address < 0x800000
        Use Type-2 Template
        Physical Page Number = Physical Address >> 11
```

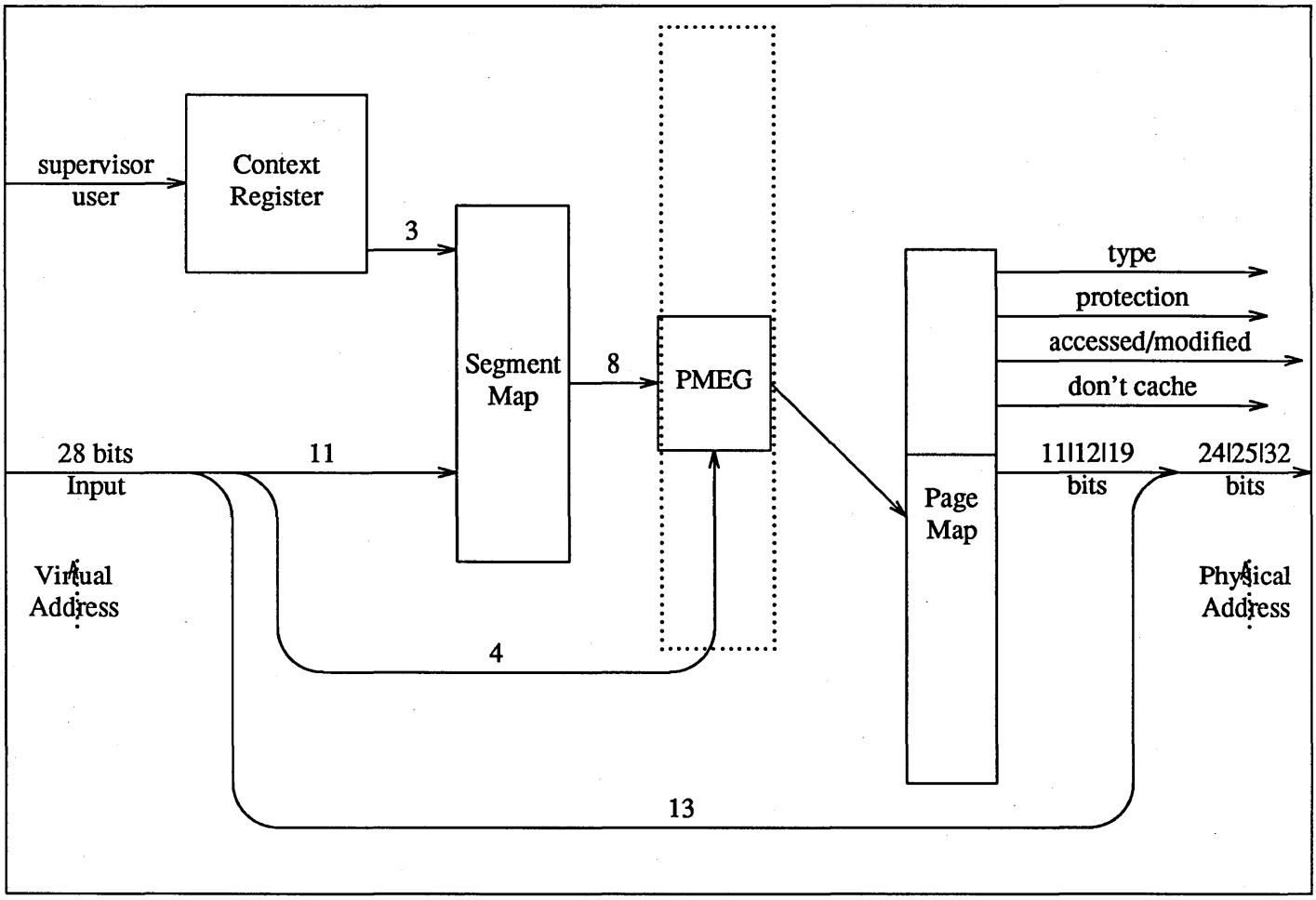
```
If Sun-2 vme16d16

    Use Type-3 Template
    Physical Page Number =
        (Physical Address + 0x7F0000) >> 11
```

Sun-3 and Sun-4 Address Mapping

Consider the following diagram of address mapping on the Sun-3.

Figure 5-6 *Sun-3 MMU*

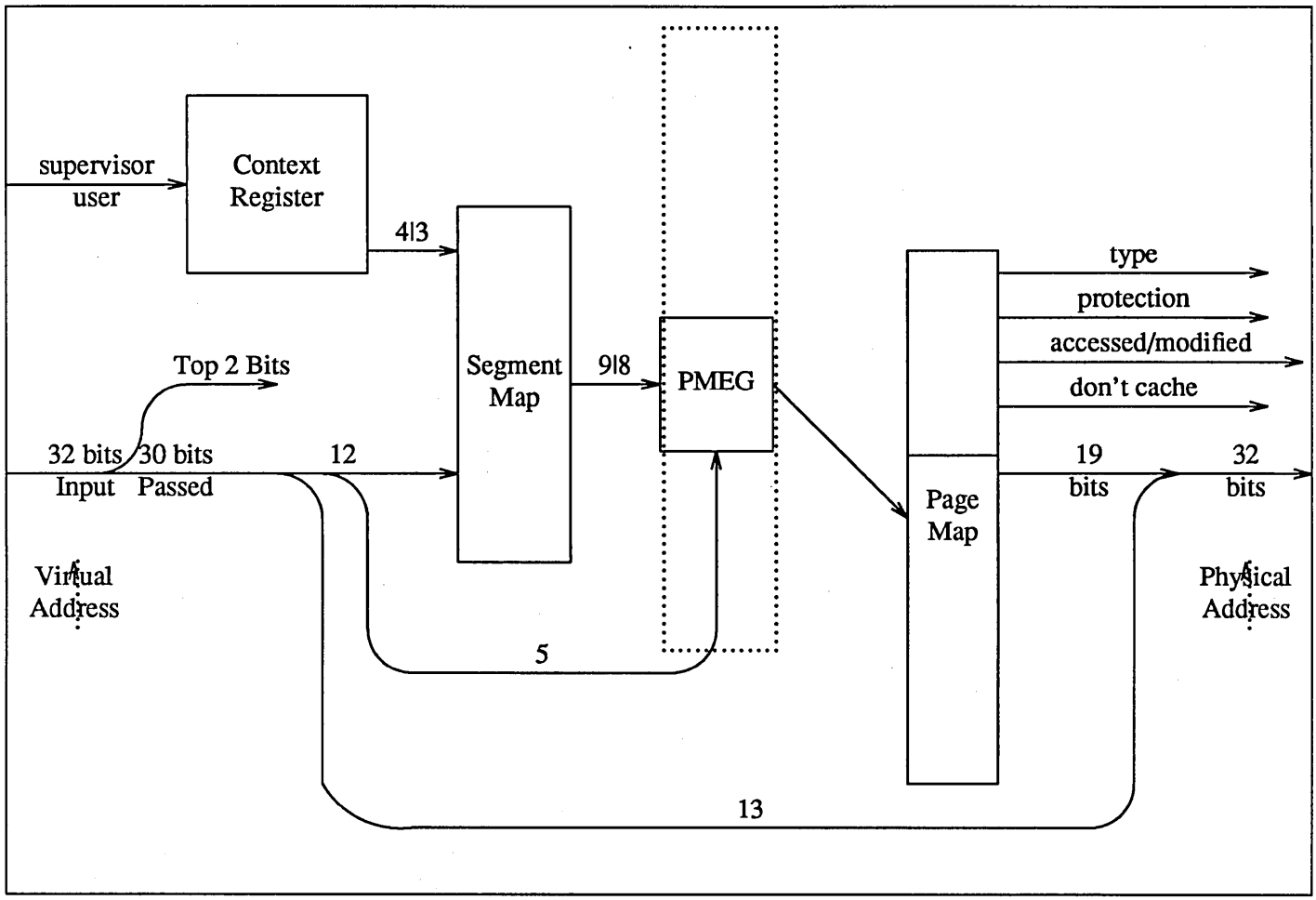


As you can see, the general scheme is the same as it was in the Sun-2, but the details have changed:

- The MMU is getting a 28-bit virtual address as its input, as opposed to a 24-bit address in the Sun-2.
- The number of mode and permission bits in the PTE has been reduced.
- The number of high-order bits reported out of the MMU, and thus the size of the physical address, is variable. The address size is fixed for any given Sun-3 machine, and varies only with the model — there are different kinds of Sun-3 machines and they have different physical address sizes.

The Sun-4 MMU is almost the same:

Figure 5-7 *Sun-4 MMU*



As you can see, the Sun-4 MMU is largely identical to the Sun-3 MMU. The differences are that:

- The Sun-4 MMU gets a 32-bit virtual address as its input, as opposed to a 28-bit address in the Sun-3. The top two bits are immediately shunted off. They must be either `00` or `11`, and are used to specify one of the two "chunks" in the virtual address space. (See *Selecting a Virtual to Physical Mapping* above).
- The number of bits coming off the Context Register is 4 (to specify one of 16 contexts) on Sun-4/260s and 3 (to specify one of 8 contexts) on Sun-4/110s.
- The number of bits coming off the Segment map is 9 for Sun-4/260s and 8 for Sun-4/110s.

On both Sun-3 and Sun-4 systems, PTEs are 32-bit numbers with the following structure.

V	w	s	c	Type	a	m	Unused (5)	Physical Page Number (19)

As we did with Sun-2 PTEs, we will make a "template" bit mask that we can use to construct our standard PTEs. One acceptable mask assumes values as follows:

```
V (valid) = 1
w/s (write ok/supervisor only) = 11
c (don't cache) = 1
(a/m) accessed/modified = 00
unused = 00000
```

(A one (1) in the don't cache position only disables caching if the type is zero (0), since other types of pages are never cached). With the above values, our template will be:

1	1	1	1	Type	0	0	0	0	0	0	Physical Page Number (19)

This gives us a mask of `0xF0000000` (if we assume that the type field is `00`). Thus, the four masks for the four types of memory are:

Table 5-2 *Sun-3/Sun-4 PTE Masks*

Type	Description	Mask
0	On Board Memory	0xF0000000
1	On Board I/O Space	0xF4000000
2	vme16d16	0xF8000000
2	vme24d16	0xF8000000
2	vme32d16	0xF8000000
3	vme16d32	0xFC000000
3	vme24d32	0xFC000000
3	vme32d32	0xFC000000

To determine the value to be plugged into the PTE, we must add the appropriate mask to the appropriate physical page number, thus giving us the full 32-bit number that we need. Here, again, we will give rules instead of details.

```
If vme16d16
   or vme24d16
   or vme32d16

    Use Type-2 Template
```

```
If vme16d32
   or vme24d32
   or vme32d32

    Use Type-3 Template
```

```
If vme32d16
   or vme32d32

    Physical Page Number = Physical Address >> 13
```

```
If vme24d16
   or vme24d32

    Physical Page Number =
        (Physical Address +0xFF000000) >> 13
```

```
If vme16d16
   or vme16d32

    Physical Page Number =
        (Physical Address +0xFFFF0000) >> 13
```

Sun-3x Virtual to Physical Mapping

In the previous CPU board designs, such as the Sun-3 architecture, a discrete MMU was designed and implemented to handle Demand Paging (off chip). That MMU was implemented mostly in hardware, with a dedicated register for the Context and separate high speed RAM for the Segment and Page values. In the Sun-3x architecture where the MC68030 is used as the CPU, a fully programmable Memory Management Unit (MMU) integrated into the silicon (on the 68030 chip) will be used to handle demand paging. A similar MMU has been offered by Motorola for some time (the MC68851 MMU) but was not used by Sun due to certain architectural incompatibilities.

This Memory Management Unit is drastically different in operation from the popular discrete version of its processors. Some of the MMU's most significant changes involve how the Translation Tables are initialized, accessed, and updated and also the way the Address Translation procedure, or Table Walk, is completed. This next discussion provides the process of how the firmware builds, initializes, and updates the entries in the MMU Translation Tables, how the Table Walk is accomplished, and how the MMU performs Address Translation. An example is shown how to use the monitor to map virtual addresses into physical addresses to access devices through the PROM.

The MMU handles the translation of addresses from virtual to physical using translation tables stored at arbitrary locations in memory. The MMU has an

Address Translation Cache (ATC) that holds recently used virtual to physical address translations. When the CPU passes a virtual address to the MMU for translation, it will first search the ATC for the corresponding physical address. If the requested entry is not in the ATC, the processor will search the translation tables in main memory for the information. An ATC access operates in parallel with the other on-chip caches, namely the CPU's Instruction Cache and Data Cache. In order for the MMU to operate correctly, its internal registers must be initialized to a known state.

The MMU has several internal registers that are initialized to known values before the MMU is Enabled (Address Translation Enabled) and during various Reset (k2 or power-on) operations. These registers include the CPU Root Pointer (CRP), the Supervisor Root Pointer (SRP), and the Translation Control (TC) register, all of which are initialized while the MMU is Disabled (Translation Disabled). The CRP and SRP are discussed in the Motorola 68030 Manual, but for now it is important to say that these registers contain the starting addresses for the MMU's table walk.

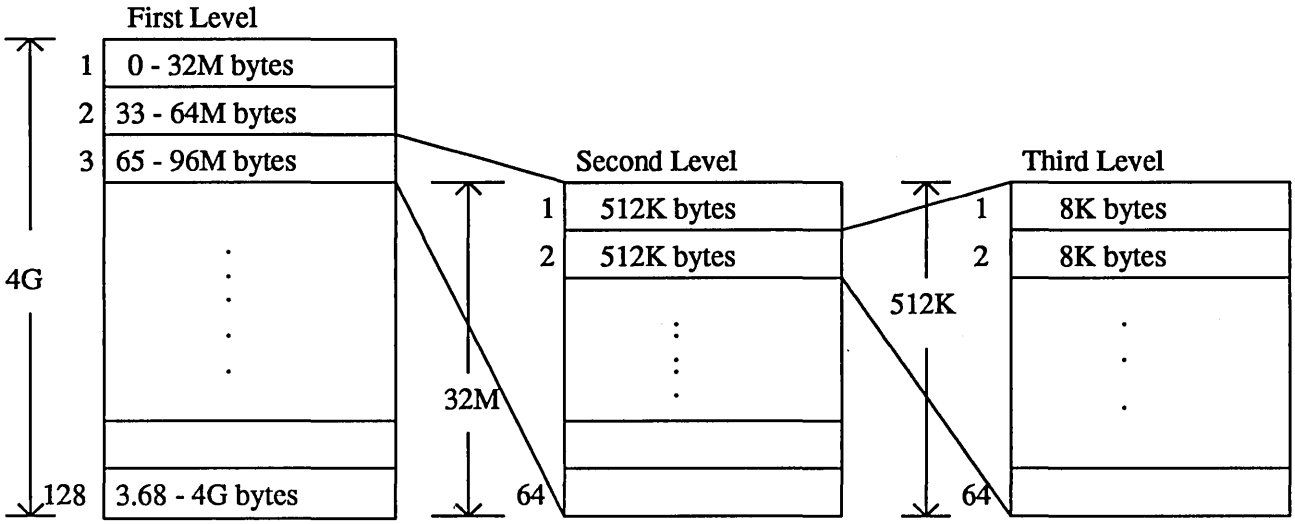
The Table Walk

The MMU's principal function is address translation, which involves converting a virtual or logical address to a physical address. This process is known as a Table Walk. For the Sun-3x architecture a three level MMU has been designed and requires that a three level table walk be initiated to perfrom address translation. This process terminates when either an INVALID Entry or PAGE Descriptor is encountered. The three levels of address translation are referred to as TIA, TIB, and PAGE respectively.

The three level table walk is needed to evenly divide the four gigabyte addressing range of the MC68030. This could have been accomplished several different ways, but a specified design goal was to have the Firmware, the Executive Diagnostic and the Unix Operating System all use the same Translation Table format.

The first level of lookup, the TIA table entry, must be able to map in the entire four gigabyte addressing range all at once. The largest block of virtual memory that is required at any one time will be 32 megabytes. By dividing 4 gigabytes by 32 megabytes we get 128 entries for the first level of address translation. For the second level of translation, the TIB entries will take each of the 32 megabyte TIA entries and divide them by 64. This will allow each TIA entry to be accessed as 64 separate 512 Kbytes (1/2 megabyte) blocks. Each of the 64 TIB entries are then divided into 64 again which results in 8 Kbyte page sizes.

It is because of this table traverse that the name Table Walk is used. Each virtual address is translated to a physical one by taking parts of the virtual address and using them as indexes into the three tables, the resulting output being a Page Table Entry (PTE) which will determine that exact physical address. See the table below for how the entire virtual address range is divided into 8 Kbyte ranges.



The beginning of the table walk starts with a pointer to the location of the MMU tables in main memory. The PMMU has two pointers, one that is used by the CPU (CPU Root Pointer), and one that is used by the CPU while in supervisor state (Supervisor Root Pointer). For the firmware's use, both the CRP and the SRP are initialized to the same value, which means they will both point to the base of the MMU tables.

When the MMU is Enabled, the CPU will pass virtual addresses to be translated to the MMU. If the requested entry is not in the ATC, a table walk of the translation table will be initiated. The table walk sequence is described below.

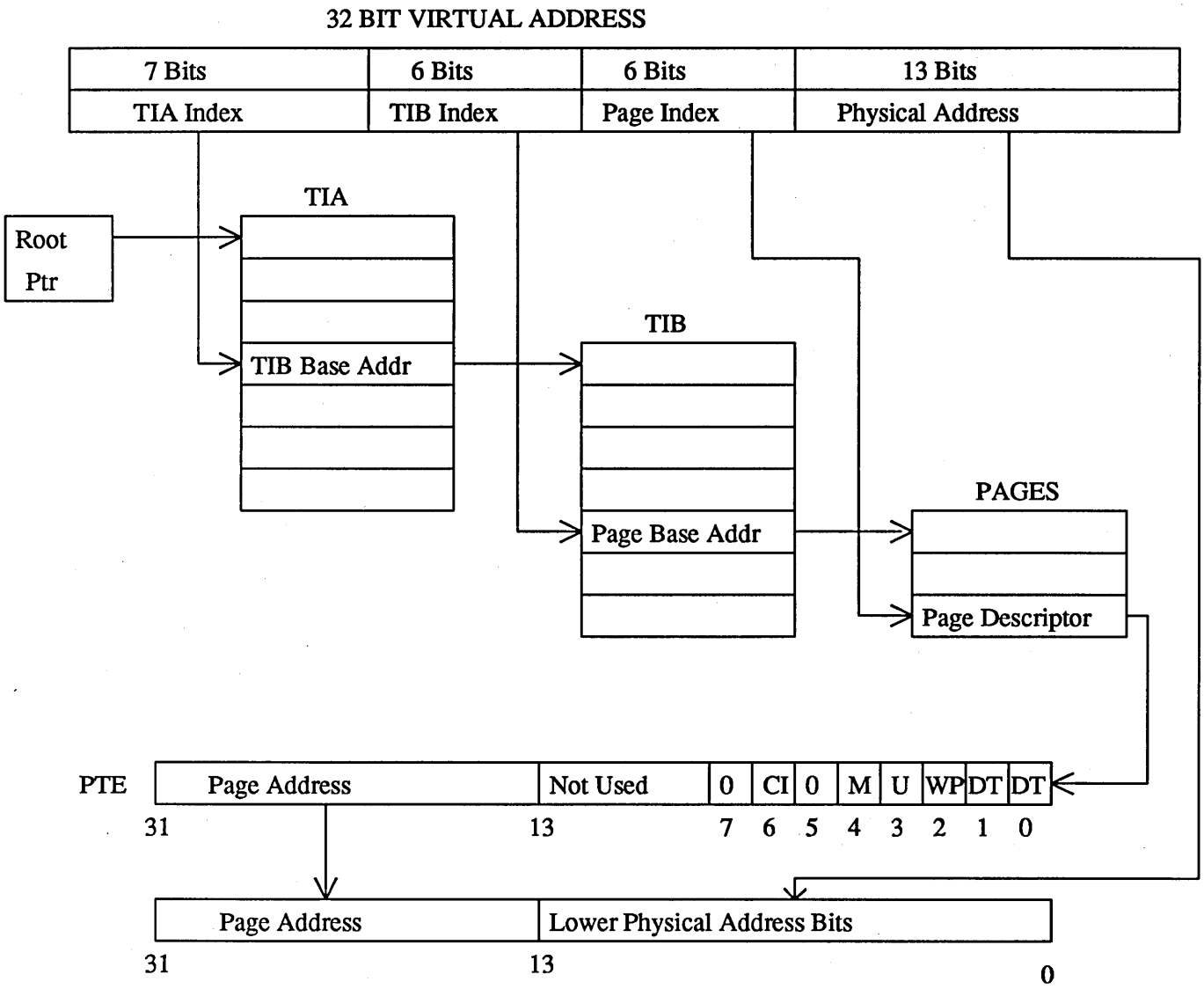
Step One: The CRP contains the base address of the TIA table in memory. The top seven bits of the Virtual Address are used to calculate the index into the TIA table. This index is added to the CRP to generate the specific TIA table entry. The TIA entry contains the base address of the TIB table for the next step.

Step Two: The next six bits of the virtual address are used as an index into the TIB table. When added to the base address from the TIA table the specific TIB table entry is generated. The TIB entry contains the base address of the PAGE Table.

Step Three: The next six bits of the virtual address are used as an index into the PAGE table. The base address from the TIB table plus the index result in the PAGE Table Entry (PTE). The PTE contains a 32 bit PAGE Descriptor of which 19 bits are the Page address, 5 are unused, and the remaining 8 are Status bits.

The Physical address is calculated by taking the top 19 bits from the PTE and the lower 13 bits from the Virtual address. These 13 bits are an offset into the physical memory page that is selected from the 19 bits.

The table walk is completed by passing the physical address back to the CPU. If an INVALID descriptor is ever encountered the table walk terminates.



A Few Example PTE Calculations

Example One: You wish to map a device which you have attached at physical `0x280008` onto bus type `vme24d16` which will be mapped into virtual memory at address `0xE000000`. What is the corresponding PTE?

Sun-3 Solution

Well, since we are mapping the device into `vme24d16`, we will use `0xF8000000` as the template. Then, following the Sun-3 rules, as given above, we add the physical address to `0xFF000000`. This yields `0xFF280008`. In binary, this is:

```
1111 1111 0010 1000 0000 0000 0000 1000
```

Shifting this right by 13 yields:

```
XXXX XXXX XXXX X111 1111 1001 0100 0000
```

Adding the template, 0xF8000000, we get values for the 13 bits that are undefined from the shift. Thus the PTE is:

```
1111 1000 0000 0111 1111 1001 0100 0000
```

Which is 0xF807F940.

A final note: we've now calculated the PTE that maps the virtual page beginning at 0xE000000 to the physical page containing 0x280008. To get the virtual address by which to access the device it's necessary to take the lower 13 bits of the physical installation address — the bits that are just passed through the MMU — and add them to virtual 0xE000000. The lower 13 bits of physical 0x280008 are 0008, and adding them to 0xE000000 yields 0xE000008, the virtual address by which the device can be accessed.

Sun-3x Solution

Our variables are:

```
physical address    280008
virtual address     E000000
bus type            vme24d16
```

The base address for vme24d16 for the Sun-3x, which is in Table 2-8 in Chapter 2, is 0x7e000000 So we add the physical memory address to the vme base pointer which gives us a specific physical address.

```
vme24d16    7E000000
physical      280008
            --------
physical    7E280008
```

Then we take off the top 19 bits to mask out just the vme page, which gives us the physical page of memory. We then need to logically 'or' in some status bits to allow us to write to this page. The value 1 enables the write status.

```
physical    7E280008
and mask    7E280000
            --------
page        7E280000
or flag            1
            --------
PTE         7E280001
```

To use the monitor to perform the mapping, use the 'p' command for displaying and changing the Page Table. The syntax is

```
p[virtual address]
```

where the virtual address is the original virtual memory given in the problem initially. The monitor returns the current PTE and asks you for a new value. The newly calculated PTE is input, which modifies the PTE to map to a new physical memory location

```
monitor cmd      >pE000000<cr>
return value     xxxxxxxx
new PTE          ?7E280001<cr>
exit monitor
```

Now every reference to the virtual memory location `E000008` will be mapped to the device. Note that since the original physical address was folded into the virtual address and then was masked, we still have the 8 offset at the end of the memory reference to index into the physical page of memory to access the device.

Example Two: You wish to map physical `0xEE48` on bus type `vme16d32` on a Sun-3. Using virtual address `0xE000000`, what is the PTE?

Sun-3 Solution

Since we are mapping the device into `vme16d32`, we will use `0xFC000000` as the template. Then, following the Sun-3 rules, as given above, we add the physical address to `0xFFFF0000`. This yields `0xFFFFEE48`. In binary, this is:

```
1111 1111 1111 1111 1110 1110 0100 1000
```

Shifting this right by 13 yields:

```
XXXX XXXX XXXX X111 1111 1111 1111 1111
```

Adding the template, `0xFC000000`, we get values for the 13 bits that are undefined from the shift. Thus the PTE is:

```
1111 1100 0000 0111 1111 1111 1111 1111
```

Which is `0xFC07FFFF`.

To get the virtual address by which to access the device at physical `0xEE48`, add its lower 13 bits, `0xE48`, to `0xE000000` — this yields `0xE000E48`.

Sun-3x Solution

Using the same steps above, this is how the solution looks:

```
physical     EE48
virtual      E000000
bus type     vme16d32

vme16d32     7D000000
physical         EE48
             --------
physical     7D00EE48


physical     7D00EE48
and mask     7D00E000    0111 1101 0000 0000 111
             --------
masked page  7D00E000
or flag             1
             --------
PTE          7D00E001
```

This is the new PTE value that can be used in the monitor as shown in the previous example.

Getting the Device Working and in a Known State

Before you even *think* about writing any code you should check out your device. You must get to know it, finding out early if it has any peculiarities that will affect its driver. It may, for example, have addressing and data-bandwidth limitations. Or, if it's a bus master, it may not implement the *release on request* bus-arbitration scheme the Sun supports. *Know the peculiarities of your device early, and then test it to verify that it's working before proceeding further with driver development.*

Make sure that the board is set up as specified in the vendor's manual. Device characteristics which, in general, have to be set properly before the device can successfully be used include:

- I/O register addresses for I/O mapped Multibus boards,
- Memory base addresses for Multibus boards that use Multibus memory space,
- Address and data widths,
- Interrupt levels,
- Interrupt vector numbers for VMEbus device,
- VMEbus address modifiers,
- The bus grant level for VMEbus devices should be set at 3.

Then, take down your system and power it off. Plug the device into the card cage and attempt to bring the system back up. If you can't boot the system, then there's a problem. Perhaps the board isn't really working, or perhaps it's responding to addresses used by other system devices. You must resolve this problem before proceeding further.

Take SunOS down again and attempt to contact the device using the PROM monitor. To do so, you will need to set up a PTE on the Sun-2, Sun-3, or Sun-4 which maps to the device's physical installation address. Use the procedures given above to calculate a PTE, then:

- Issue the monitor command that puts you into *supervisor data* state. This will be **s B** for Sun-4 machines and **s5** for all others. So, if you have a Sun-3, give the

  ```
  >s5
  ```

  command.
- Calculate, using the procedures given above, the PTE appropriate to the physical address you've chosen.
- Set the position in the kernel page map that corresponds to your physical address to contain the calculated PTE. This will map your chosen physical address, thus putting you in contact with your device. You may use the monitor's **p** command to perform this mapping. The **p** command takes a virtual address as its argument, displays the PTE that corresponds to that virtual address, and gives you the option of modifying the PTE. For example:

  ```
  >pF32000
  ```

  selects the page map entry that corresponds to the virtual address of `0xF32000` and displays it. It also displays a '?', which indicates that you may type in a new value to replace the one displayed. (See the appropriate *PROM Commands* chapter of the *PROM User's Manual* for more details). Note that all virtual addresses within a page select the same PTE.

Having contacted the device from the monitor, try some of the following:

- Try reading from the device status register(s), if there are any.
- Try writing to the device control and data registers(s), if there are any. Then try reading the data back to see if it got written properly (this assumes, of course, that the device allows the reading of these register(s).
- Try actually getting the device to do something by sending it data.
- If the device is a controller with separate slave devices, then switch a slave on and off and watch for changes in the controller status bits.

Your goal is to try to actually operate the device, for a moment, from the monitor. For example, if you have a line printer, try to print a line with a few characters. Be aware that bit and byte ordering issues are critical in this process. The reason you're doing this is to ensure that the device works and that you understand the way it works. When you understand the device's peculiarities, you can proceed to write a driver for it.

A Warning about Monitor Usage

When you use the monitor's **o**, **e** or **l** commands to open a location, the monitor *reads* the present contents of that location and displays them before giving you the option to rewrite them. In the best of all possible worlds, this would present no problems, *but many devices don't respond to reads and writes in as straightforward a fashion as does normal memory.*

For example, the Intel 8251A and the Signetics 2651 use the same externally addressable register to access *two* separate internal mode registers, and they have internal state logic that alternates accesses to the external register between the two internal registers. So suppose that you want to put something in mode register 1 of the 8251. You open the external register, the monitor displays its contents, and you then do your write. If, being cautious, you then read the external register to check that the data you wrote is there, you will find that it's not — because the read will sequence you on to the second register.

To deal correctly with such devices, it's necessary to use the monitor's "write without looking" facility and then read the locations back later to check them. You can write without looking with any of the monitor commands that "open" an area of memory; all that's necessary is that you enter a `value` after the `address` argument. For example:

```
>l [address] [value]
```

This will cause `value` to be written into `address` without first reading its current contents. For more information on hardware peculiarities and the problems that they can cause for the monitor, the *Hardware Peculiarities to Watch Out For* section of the *Hardware Context* chapter.

5.2. Installation Options for Memory-Mapped Devices

Memory-Mapped Device Drivers

Memory-mapped devices are the simplest types of devices to write drivers for. Frequently, however, their essential simplicity isn't obvious from a quick glance at their source code. This is because many memory-mapped devices are frame buffers, and frame-buffer drivers must set up and manage the low-level interface for the Sun window system as well as the standard device interface. Consequently, they tend to be littered with declarations and manipulations related to the "pixrect" (pixel rectangle) system. See the *Pixrect Reference Manual* for more details.

Memory-mapped devices are most frequently installed into Sun systems with simple drivers that map them into user address space (there are sometimes alternatives to such drivers, as you will see below). Such memory-mapped drivers don't really do much. Obviously, *xx*`probe()` and *xx*`mmap()` must exist, for the kernel must be able to check the device installation and perform the actual device mapping. And, in addition, *xx*`intr()` must be real if the device is interrupt driven. But *xx*`open()` and *xx*`close()` are usually stubs, and *xx*`read()` and *xx*`write()` can be calls to `nulldev`.

Keep in mind that the major purpose of a memory-mapped driver is to support the `mmap()` system call. This is very important because user processes which call window code must first map the frame buffer into their address space. They do so with the `mmap()` system call, which is translated by the kernel into a series of calls to the driver's `mmap` routine. Each of these calls returns page table entry information which the kernel needs to map a single page (the next page) of frame-buffer memory into a virtual address space. Here's some very simple driver *xx*`mmap()` code.

```
/*ARGSUSED*/
cgonemmap(dev,off,prot)
    dev_t dev;
    off_t off;
    int prot;
{
    return (fbmmap(dev,off,prot,NCGONE,cgoneinfo,CG1SIZE));
}

/*ARGSUSED*/
int fbmmap(dev, off, prot, numdevs, mb_devs, size)
    dev_t dev;
    off_t off;
    int prot, numdevs;
    struct mb_device **mb_devs;
    int size;
{
    int kpfnum;

    if ((u_int) off >= size)
        return -1;

    kpfnum =
      hat_getkpfnum(mb_devs[minor(dev)]->md_addr + off);
    return kpfnum;
}
```

dev is, of course, the device major and minor number, and *off* is the offset into the frame buffer (passed down from the user's `mmap()` system call). *prot* is also passed down from the user's call, but it is not currently used. As you can see, there's a bit of shuffling around and then a call to `hat_getkpfnum`, which returns a Page Frame Number which *xx*`mmap()` is expected to return.

Note that `mb_dev->md_addr` is the address of the frame buffer from the Main Bus device structure. This is the device installation address as given in the kernel config file. The offset is checked to be sure the user isn't mapping beyond the end of the frame buffer.

Mapping Devices Without Device Drivers

Under a restricted set of circumstances, it's possible to avoid writing a device driver altogether by using the `mmap()` system call to overlay the device's registers and memory onto user memory. Having done this, you can read and write the registers — as if they were normal user memory — from a user program.

What this really amounts to is piggybacking the new device onto an another, system standard, virtual memory device (and its driver). The `mmap()` routine of a system virtual memory device is then used to do the user-device mapping, and the "installation" is accomplished without the development of a driver specific to the user device. Instead, a user level program is written, one that calls the `mmap()` system call.

The restrictions on this shortcut are, however, fairly severe.

- The device must not require any special handling of the type that would go into *xx*`ioctl()`.
- The device (including all its control registers) must work with user function codes, since that's what it will get when mapped into and then accessed from user space.

NOTE *MC680X0 processors, SPARC processors and the Intel 80386 all run in either 'user' or 'supervisor' state. Many devices, in turn, restrict certain of their operations, and will only perform them when the processor is in supervisor state. The Sun CPU is in supervisor state only when executing kernel code. This means that device drivers, which are part of the kernel, can issue device commands which are not available from user processes. Also note that, when the CPU is in supervisor state, as it is when driver code is executing, the device will receive different VMEbus address modifier codes than when the CPU is in user state. For details about these codes see the VMEbus specification.*

- The device must not require any other sort of special handling — it cannot, for example, be multiplexed, interrupt driven, or do DMA.
- Finally, there are security problems associated with this sort of installation. Since the system virtual-memory devices are normally owned by and restricted to the superuser, your programs will either have to change their permissions to allow normal users to access them, or will have to run with superuser privileges. The former strategy is usually not acceptable in the long run, because it creates a gaping hole in the security of the system. And it's far from clear that the second alternative is desirable either.

The virtual-memory devices of interest here are those that support mapping over the entire range of a virtual address space. They are:

Table 5-3 *Virtual Memory Devices*

Machine Type	Memory Device Name
Multibus (Sun-2 only)	`mbmem`
Multibus (Sun-2 only)	`mbio`
VMEbus (All Sun's)	`vme16d16`
VMEbus (All Sun's)	`vme24d16`
VMEbus (Sun-3 and Sun-4)	`vme32d16`
VMEbus (Sun-3/Sun-3x/Sun-4)	`vme16d32`
VMEbus (Sun-3/Sun-3x/Sun-4)	`vme24d32`
VMEbus (Sun-3/Sun-3x/Sun-4)	`vme32d32`
ATbus (Sun386i only)	`atmem`

In addition, there are memory pseudo-devices that support access to the on-board devices that users are allowed to access. These are `/dev/fb`, `/dev/mem` and `/dev/kmem` (See the `mem(4)` manual page for details).

`/dev/fb` is a memory device which, on any given system, is set up to address the local frame-buffer device. It can be used as if it were a system memory device — on any given system, `/dev/fb` can be `mmap()`'ed into user memory and then written to, with the effect of writing the local frame buffer memory.

To use `mmap()` with one of the system memory devices, you must do three things:

- Open the device.
- Calculate the *offset* which you will need to call `mmap()`. This offset is merely the device address on the appropriate system memory device rounded to a page boundary. That is to say that you get the offset from the device manual and/or the switches on the device itself.
- Call `mmap()` to allocate virtual space and map in the physical bus address of your device, which you must know. (See the *Hardware Context* chapter for a discussion on how to pick a good physical address from the information in the system config file).

The following example program uses `/dev/fb` rather than one of the virtual memory devices. It makes a good example because it maps the system frame buffer into user memory so that it can then be written from a user program. It uses `mmap()` to set things up, but doesn't bother with calling `munmap()`, because unmapping occurs automatically when the memory device is closed. This close occurs implicitly when the program ceases execution. (The machine segment size is 128K for the Sun-2 and Sun-3; 256K for the Sun-4; and 4Mbytes for the Sun386i. Areas greater than the machine segment size should be mapped only with special care. The Sun-3x has no segment size so any input value will work. For details, see the discussion of `mmap()` in the *User Support Routines* appendix).

Once the device has been mapped into user space it can be treated as a piece of local user memory. (Remember that memory accesses performed by way of this mechanism will be reflected — at the device level — as non-privileged (user) accesses. This is because `mmap()` accesses inherit the privilege of the process that calls `mmap()`. Thus, if memory is mapped by a driver, subsequent accesses to it will have the standard supervisor data access privilege, but if it's called from a user process, as described here, subsequent accesses will be non-privileged. Attempts to access supervisor-only device registers without supervisor privilege might produce a bus error, i.e., they're inaccessible from a user program, and thus a kernel level driver must be written to manipulate them. The device will also receive different address modifier codes when accessed from a user process than when accessed via a device driver).

```
#include <stdio.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/types.h>

/* Width and Height of Frame Buffer in Bits */
#define WIDTH 1152
#define HEIGHT 900

main()
{
    int fd;
    unsigned len;
    char *addr;

    /* Open the frame-buffer device */
    if ((fd = open("/dev/fb",O_RDWR)) < 0)
        syserr("open");

    /* Compute total number of bytes */
    len = ((WIDTH * HEIGHT)/8);

    /*
     * offset must be page aligned.  /dev/fb
     * is already aligned with frame-buffer memory
     */
    offset = 0;

    /* Map device memory to user space */
    addr = mmap((caddr_t)0, len, PROT_READ|PROT_WRITE,
        MAP_SHARED, fd, 0);
    if (addr == (caddr_t)-1)
        syserr("mmap failed");

    writeFB(addr);
    exit(0);
}
```

```
writeFB(addr) /* Write to frame buffer */
    char *addr;
{
    char color;
    int i,j;

    color = 0xFF;
    for (i = 0; i < HEIGHT; i++) {
        color = ~color;
        for (j = 0; j < WIDTH/8; j++)
            *addr++ = color;
    }
}

syserr(msg) /* print system call error message and terminate */
    char *msg;
{
    extern int errno, sys_nerr;
    extern char *sys_errlist[];

    fprintf(stderr,"ERROR: %s (%d", msg, errno);
    if (errno > 0 && errno < sys_nerr)
        fprintf(stderr, "; %s)\n", sys_errlist[errno]);
    else
        fprintf(stderr,")\n");
    exit(1);
}
```

NOTE *This example uses the special memory device* `/dev/fb`, *since this device is always set up to address the frame buffer memory.*

So, despite the plethora of limitations on the sorts of devices that can be installed by way of mapping them into user space, it's quite an easy thing to do. If your device characteristics are such that this is an option, you may well wish to take it. And even if such an installation isn't an attractive long-term option (for example, because of unacceptable security problems) it may still be attractive as a short-term alternative to driver development. Even in environments where security considerations make it unacceptable in the long term, it can allow you to get your device up and running very quickly. Sometimes this counts for a lot.

Direct Opening of Memory Devices

It should be noted, for the purpose of completeness, that there's another approach to avoiding driver development, one that's even easier than the use of `mmap()` described here, and even more limited. That is, it's possible to simply open the virtual memory device that contains your board, to seek to the location of its registers, and then to read and write those registers as if they were regular memory.

This approach has most of the same problems as does the use of `mmap()`, *and is notable mainly because, with it, the device receives supervisor function codes. It does, however, introduce new problems.* It doesn't give you the same degree of control as does `mmap()`, and you often need that control when dealing with

devices. When you use mmap(), the device actually becomes part of your user memory space, and it's left to the compiler to generate exactly the I/O accesses which you implicitly specify in your structure and variable declarations. You can always access exactly what you want, and the accesses occur directly as *move byte* and *move word* operations. Thus they are very fast.

When, however, you simply open a system memory device as a file and then read and write to it, your communication with your board is mediated by the I/O system. The I/O systems will always try to do the "right thing" (if you request I/O at an odd address or for an odd number of bytes it will perform byte access as appropriate; otherwise it will use short integers), but it still doesn't give you the kind of control that can be had using mmap(). Furthermore, I/O operations involve lots of code, and take *hundreds* of times as long as direct references to `mmap()`'ed references, which proceed by way of the MMU and use low-level store and move instructions to directly access device registers and memory as physical memory.

So the bottom line is that, unless you need to access a device only a few times, or if you need to receive supervisor function codes (and the corresponding VMEbus address-modifier codes) and performance isn't critical, you can do your installation by opening a system memory device and then seeking to your device registers and memory space. Otherwise, use `mmap()` or write a driver. If you do decide to use the `open()`/`lseek()` method, do so with low-level I/O rather than with the standard I/O library. The standard I/O library implements a buffered I/O scheme which will add considerably to your problems.

The following user program is similar to the example above, in that it writes the same pattern to the memory of a frame buffer. This time, though, the write is done by way of the I/O system rather than by using mmap(), and the frame buffer is taken to be installed at *OFFSET* (whatever the device physical installation address is) in the `vme24d16` memory space.

NOTE *Since all Sun VMEbus machines have a built-in, on-board frame buffer, this example is only meaningful for color frame buffers. On Sun-2 Multibus machines, however, this code would work with* `/dev/obmem` *and an offset of* `BW2MB_FB`.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/buf.h>
#include <sys/file.h>

void syserr();
long lseek();

/* Width and Height of Frame Buffer in Bits */
#define WIDTH 1152
#define HEIGHT 900

main()
{
```

```
    int fd;

    /* Open the system memory device containing the frame buffer */
    if ((fd = open("/dev/vme24",O_RDWR)) < 0)
        syserr("open");

    /* Seek to the frame buffer memory */
    if (lseek(fd, (long)OFFSET, L_SET) == -1L)
        syserr("lseek");

    writeFB(addr);
    exit(0);
}

writeFB(fd) /* Write to frame buffer */
    int fd;
{
    char color;
    int i,j;

    color = 0xFF;
    for (i = 0; i < HEIGHT; i++) {
        color = ~color;
        for (j = 0; j < WIDTH/8; j++) {
            if (write(fd, &color, 1) == -1)
                syserr("write");
        }
    }
}
```

5.3. Debugging Techniques

As described above, it's a good idea to begin debugging by using the monitor to check that the device has been installed at the intended address, and that it works, before proceeding to debug your device driver. This allows you to avoid debugging the device simultaneously with the driver, and experience that you'd like to avoid for as long as possible. Alternatively, if you're confident in both your device and the correctness of your installation, you can simply make a new kernel, boot it and proceed with debugging. In this case you should put some `printf()` messages — see below — into the *xx*`probe()` routine. Then you can at least see the device get contacted and initialized.

Debugging drivers is significantly more difficult than debugging regular user programs, for a number of reasons:

- In the first place, device drivers are part of the system kernel. This means that the system is not protected from their errors. Addressing errors, for example, will frequently trip hardware traps and crash the system.
- As mentioned above, there's the possibility that the device hardware will be buggy. For this reason, you can't really trust your environment in the same way as you can when writing a user program on a mature computer system.

- Some devices behave in rather peculiar ways. (See *A Warning about Monitor Usage*, above).
- Finally, the debugging environment in the kernel is thinner than it is in user space. There is a kernel debugger, `kadb`, and this a a big step towards making life easier for driver developers. Still, life remains more difficult when debugging in kernel space.

  It's possible to prototype drivers in user address space by using techniques similar to those described in the Mapping Devices Without Device Drivers section of this chapter. The same constraints given there apply to prototyping. In particular, it's not possible to run an interrupt routine, or to probe for non-existent devices without generating bus errors from prototype drivers in user space. If the device generates no interrupts, and if it doesn't do DMA, the entire driver might be able to run in user space.

For all these reasons, you should give extra care to desk-checking your code, and check a reference manual when not absolutely sure of the meaning of a given construction. Don't take chances.

Also, make changes incrementally. Don't try to save time by making many changes at once. You will save time in the long run if you take the time to add and test a few parts at a time. Keep your feet on solid ground.

Use trace output from `printf()`, as described below. Drivers can act in surprising ways, and the best way to proceed is by making the flow of operations highly visible.

NOTE *On the Sun386i system, the loadable drivers feature makes driver development much easier because the code-compile-reboot-test cycle is reduced to code-compile-load-test.*

Debugging with `printf()`

With the availability of `kadb`, the kernel debugger, the importance of `printf()` in the debugging of device drivers has been significantly reduced. Still, even with `kadb` available, `printf()` statements remain useful as means of providing synchronous tracing of overall driver flow and structure. `kadb` can be made to provide a similar sort of tracing (by tying print commands to strategically chosen breakpoints) but this won't altogether eliminate the `printf()` statement. The `printf()` has long found application in driver debugging, and, as a matter of taste and experience, some programmers will continue to use it. For this reason, we will discuss its use in some detail.

The kernel `printf()` sends its message directly to the systems console, without going through the tty driver. As a consequence, the printing is uninterruptible—the characters aren't buffered. Furthermore, `printf()` runs at high priority, and no other kernel or user process activity takes place while its output is being produced. `printf()` thus radically limits overall system performance (though this is usually ok while device drivers are being debugged).

The window systems should not be up when you use `printf()` to debug a driver because its output will go to the console window. On

the Sun386i system, it is best to set the global variable `newlog` to 0.

There is a second kernel print statement, `uprintf()`. `uprintf()`, however, is of little use to driver developers. It attempts to print to the current user tty as identified in the `user` structure, and prints to the console only if there's no current user tty (at which point it becomes identical to `printf()`). `uprintf()` cannot be called from lower-half routines, which run in interrupt context and cannot make any assumptions about the `user` structure (where `uprintf()` looks to determine the current user tty). `uprintf()` is most useful for production drivers, like tape drivers that encounter media errors, which want to report errors not to a programmer but to the user.

There are occasions in which the use of `printf()` *(or* `uprintf()`*) statements will change the behavior of your driver.* `printf()` *statements, for example, can affect the timing of operations in the driver being tested as well as in other drivers. The output may be so slow relative to other device operations that interrupts are lost and system failures are introduced; thus, it is frequently impossible to synchronously trace a device interrupt routine. Driver code may begin to fail only when* `printf()`*s are introduced, or, even worse, only when* `printf()`*s are disabled. If you're debugging a tty driver, you may even face a situation where* `printf()`*-based tracing generates new calls to the driver being debugged. Thus, there are situations in which it cannot be used. In such situations, you should use* `kadb` *or the techniques suggested below in the section on Asynchronous Tracing.*

The best way to use `printf()` statements for tracing driver execution is by setting things up so that you can toggle printing by using the kernel debugger, `kadb` (see below) to set and reset print-control variables. Doing so is very simple. At the top of the driver source file, include statements like:

```
#ifdef XXDEBUG
int xxdebug = 0;
#define XXDPRINT if (xxdebug > 0) printf
#endif
```

(It's important that the variables like `xxdebug` be global, so that you can later access them freely from the debugger — remember that all drivers are part of one program, the kernel, and name your print-control variables so as to avoid naming conflicts).

Then, instead of calling `printf()` inside the driver routines, call `XXDPRINT`. Each call should be in the form:

```
#ifdef XXDEBUG
XXDPRINT("driver name...",...);
#endif
```

which will only call `printf()` if `XXDEBUG` is defined and `xxdebug` is set to a value greater than 0.

Make sure that each call to `XXDPRINT` identifies the driver, for it's possible that you, or some other programmer, will want to see debugging output from several drivers at once. And leave the debugging code in for a while after you're

finished — bugs may surface later.

Having set things up like this, you can turn the `printf()`'s on or off at any time by using `kadb` to set unset or change the print-control variable `xxdebug`. Or you can use `adb` if you wish, running it at user level in a separate window:

```
example adb -w /vmunix /dev/kmem
```

(`adb` won't allow you to set breakpoints in the kernel, but it will allow you to set and unset variables — you can change the value of `xxdebug`, or even reset a variable which has caused your driver to hang). *Remember that you're in the kernel and BE CAREFUL.*

Incidentally, `/dev/kmem` represents the kernel *virtual* address space, which is why it's used here. `adb -k /vmunix /dev/mem`, in contrast, generates a view of the *physical* address space, because `/dev/mem` represents the physical memory. This latter command is useful for examining core files.

Good places to put `printf()` statements include:

- driver routine entry points
- before critical subroutine calls
- upon reading status information from the device
- before writing of commands or data to the device
- at intermediate points in complex routines
- at routine exit points

Note again that you don't have to restrict yourself to a single `xxdebug` variable, or to binary tests that check to see if a variable is on or off. You can use as many variables, and as many values for each variable, as necessary to reflect the functional divisions most appropriate to your driver. It might even be useful to get truly esoteric, and send certain trace statements directly to the user tty (by calling `uprintf()`) while the rest use `printf()` and go to the console.

Event-Triggered Printing

In the above discussion, the `xxdebug` variable was initialized by the compiler, and toggled with a debugger. However, it's just as easy to have the driver routines themselves set a trigger variable under pre-chosen conditions.

For example, if you wanted to enable tracing after a given *condition* had occurred, you could declare `xxdebug`, just as was shown above, but define `XXDPRINT` somewhat differently:

```
#ifdef XXDEBUG
int xxdebug = 0;
#define XXDPRINT(v,msg,a1,a2) \
    if (xxdebug > (v)) printf(msg,a1,a2);
#endif
```

and then, in the code that checks for the condition:

```
#ifdef XXDEBUG
if (condition) xxdebug = 1;
#endif
```

Then to call XXDPRINT:

```
#ifdef XXDEBUG
XXDPRINT(0,"driver name...\n",a,b);
#endif
```

One major disadvantage of using the kernel `printf()` is that its output doesn't go through a device driver, and thus can't be paused with Control-S or redirected to a file. It's possible, then, that `printf()` will overwhelm you with output. There are a number of things that you can do if you run into this problem:

- If you haven't used multivalued print-control variables, then do so. This gives you more control than you have with simple on/off print control, and will allow you to reduce the amount to trace noise.
- You can use a debugger to set the global variable `noprintf`. This will keep `printf()`'s output from being sent to the console, but that output will still go to a buffer where kernel error messages are kept before being transferred to `/var/adm/messages`. You can examine the message buffer at your leisure, in one of two different ways:
  - From a user window, you can use `dmesg`.
  - From `kadb` (or `adb` on `/dev/kmem`) you can type `msgbuf+10/s`.
- It's also possible to reconfigure your system so that it uses a hardcopy terminal as its console over a RS-232 line. Then, you won't lose any of the `printf()` output.
- Best of all, you can get another machine and connect it to your machine over a RS-232 line. Having done so, use `tip` to open a window on the second machine *as the console of the test machine*. You can then use `tip`'s record feature (see the `tip` man page) to make a record of all the stuff that `printf()` is sending to the test machine's console.

Asynchronous Tracing

As mentioned above, there are occasions when timing problems forbid the use of the printf statement. In these cases, it's a good idea to give up any attachment that you might have to `printf()` statements and use `kadb`.

Or, if you prefer, it's possible to deal with timing problems by using `kadb` to patch the `noprintf` variable, and then to check the message buffer to see what's going on. Doing so:

- allows you to continue using the debugging code that you installed before encountering the timing problem, and

- presents you with a sequential list of the events in your driver, a list spelled out in English phrases and including interrupt-level events.

Or, you can simply use `kadb` for everything.

kadb — A Kernel Debugger

NOTE `kadb` *does not work with versions of the kernel earlier than 3.2.*

`kadb` is an interactive debugger similar in operation to `adb`. `kadb` differs in several key respects from `adb`. It runs as a standalone program under the PROM monitor, rather than as a user process in user address space. And it allows you to set breakpoints and single step in the kernel!

Thus, running a kernel under `kadb` is significantly different than running it under `adb -k`. The `k` option to `adb` merely makes it simulate the kernel memory mappings while `kadb` actually runs in the kernel address space. And unlike `adb`, which runs at user level and as a separate process from the process being debugged, `kadb` runs in system space as a *coprocess*. It shares not only the kernel address space but its CPU supervisor mode as well.

`kadb`, for all intents and purposes, is a version of `adb`. It has the same command syntax and almost the same command set. Thus, you can see the `kadb` and `adb` manual pages, as well as *Debugging Tools for the Sun Workstation*, for more details on its use. Note, however, the following points of special interest to driver developers:

- All interrupts are disabled while interacting with `kadb` (except non-maskable interrupts). Thus, when using `kadb` to examine memory, the kernel remains stable. However, while single stepped instructions are being executed, the actual standing priority of the kernel is temporarily restored, and interrupts can get dispatched, run and return. You won't notice unless you have a break point set in the interrupt routine, which works just fine.
- `kadb` is installed so that, when a program is being run under it, an abort sequence (L1-A) will transfer control not to the PROM monitor but to `kadb` itself. Once in `kadb`, you can abort again and be transferred to the monitor. The transfer is direct and immediate, so you can use the monitor to examine control spaces (e.g. page and segment maps) which are not accessible from `kadb`. The monitor `c` command will return you to `kadb`.
- `kadb` runs in the same virtual memory space as the kernel itself, and with the CPU in supervisor mode. This means that `kadb` uses the kernel memory maps when calculating virtual addresses, and that it can directly examine kernel structures. This is in contrast to the situation with `adb -k`, where software copies of the page table entries are used to map virtual addresses to physical pages.
- `kadb`'s memory view is almost the same as that resulting from `adb /vmunix /dev/kmem`. In other ways, however, `kadb` is much different. To give just one example: on Sun-3 and Sun-3x machines, where users and supervisors share the virtual address space, `kadb` allows the user to examine the user virtual address space (this is *not* true with `adb -k`).

- Finally, be aware that `kadb` — as a consequence of the way that `adb` works — always does 32-bit memory reads. Even if you tell `kadb` to read a byte it will read a long. This leads to a lack of control that can cause problems when reading device registers. (This problem does not exist on the Sun386i. On the Sun386i, when `kadb` is told to read a byte, it does. Within `kadb`, the `B` command is used to read a single byte and the `v` command to write one).

5.4. Device Driver Error Handling

There are various types of errors: "expected" errors like those generated by *xx*`probe()` routines, transient errors in operations that can reasonably be retried, fatal errors that require controlled shutdowns, and others. The kinds of errors that you will face depends upon the kinds of drivers that you write and the peculiarities of your devices; few generalizations can usefully be made.

To further complicate matters, the detection and treatment of errors varies greatly from device to device. You should begin by carefully reading your device specification manual to determine the error indications that can arise and the responses that should be made when they do. Most devices have at least an error bit in the control/status register, and usually more detailed error information is available. Ideally, you should understand all potential errors, avoid those that you can and recover from the rest. This ideal isn't always achievable, but try not to leave any obvious holes. *If you do nothing else, check for device errors and use the kernel* `printf()` *function to report them to the system console.*

There are various error reporting and management mechanisms available to the driver developer. Most of them have already been mentioned as they've become relevant; here they are collected and summarized:

Error Recovery

It's difficult to generalize about error-recovery mechanisms, for they are largely device specific. It's worth noting, however, that:

- Some errors are worth retrying and some aren't; the matter is entirely device specific.
- Error-recovery routines should be able to run at the interrupt level. This is because errors can occur either synchronously or asynchronously with respect to the dispatch of device commands, and, therefore, recovery routines must be callable from interrupt routines.
- If you do implement error recovery logic, you must do so consistently. The data structure that contains retry-status information must be global, and must be protected by critical sections. Error-recovery routines, like interrupt routines in general, must take special pains to protect data-structure integrity; indeed, they must *restore* such integrity upon encountering errors they can't recover from.

Error Returns

There are three mechanisms by which driver routines can report errors up to their calling routines. The first, of course, is by way of the values that the driver routines return to their callers. The second, and most important, is the error-reporting mechanism based upon the buffer-header. *This is the only mechanism that can be used when returning errors from* `xx`*`strategy()`*, `xx`*`start()`*, and* `xx`*`intr()`*. (See the discussion of* `xx`*`intr()`* error reporting in the *Summary of Device Driver Routines* chapter. Finally, it is possible to directly set the global error register, `u.u_error`, from routines in the top half of the driver.

Error Signals

It is sometimes desirable to have a driver send a software interrupt to the process or processes. It's possible, for example, that a device will fail in an unrecoverable fashion — in this case it's perhaps a good idea to signal the user processes, rather than merely returning an extraordinary error code. It's also possible (though rare) for a driver to encounter serious errors from which it can recover by restarting the device — user processes may also need to be notified in this case. The kernel `psignal()` and `gsignal()` routines can signal either a single process or all the processes in a given process group.

Error Logging

When you use the kernel `printf()` statement to report errors to the console, those errors are also placed into a system error-message buffer accessible to the `dmesg` daemon. `dmesg` can be, and typically is, run every 30 minutes by the `crontab` daemon, for the purpose of appending the messages in the buffer to `/var/adm/messages`. Note that the message buffer is small, and that if a lot of entries are being written into it, some of them will get lost before being transferred into `/var/adm/messages`.

Kernel Panics

The most unequivocal way of dealing with an error is to panic when you get it. The `panic()` routine is provided to help you do so in a somewhat controlled fashion — `panic()` is called only on unresolvable fatal errors. It prints "panic: mesg" on the console, and then reboots. (Or, if you're running under the debugger, it transfers control to `kadb`). `panic()` also keeps track of whether it's already been called, and avoids attempts to sync the disks (by flushing all pending write buffers) if it has, since this can lead to recursive panics.

The final production version of a driver should call `panic()` only when "impossible" situations are encountered; lesser errors should be recovered from. During debugging, though, `panic()` can be used to implement a passable assert mechanism.

```
#ifdef XXDEBUG
if (inconsistent condition)
    panic("Assertion Failed: ...");
#endif
```

(It's possible to write a fancier assert mechanism, for example by having an ASSERT macro which calls an `assert()` routine which prints error context information and then calls `panic()`, but this minimal hack will perhaps do).

Finally, note that in cases where it's *very* important to halt the system *immediately* after detecting an inconsistent condition, `kadb` can be used. The driver code can test for the inconsistent condition, and then set a debugging variable:

```
if (inconsistent condition)
    junk = 1;
```

`kadb` can then be used to set a breakpoint at the machine instruction generated from the assignment to `junk`.

5.5. System Upgrades

System upgrades generally have minimal effects on user-written device drivers. The changes that are necessary are rare and release specific.

Some changes must be made if user-written drivers are to work with new release software. In Release 2.0, for example, there was a minor change in one of the bus-interface structures. There wasn't much involved in adapting user-written drivers, but it had to be done.

In other cases, changes are optional. When VMEbus machines were introduced, for example, drivers had to be adapted to run on them; however, it was possible to upgrade Multibus machines without rewriting user-written drivers.

In any case, any release upgrades that imply changes — either optional or mandatory — to user-written device drivers will be documented in the *System Summary and Change Notes* for the release in question.

5.6. Loadable Drivers

The Sun386i supports loadable drivers. This feature allows you to add a device driver to a running system without rebooting the system or rebuilding the kernel. The loadable drivers feature reduces time spent on driver development, and makes it easier for users to install drivers from other vendors.

This section explains how to convert a non-loadable driver to be a loadable driver.

Conversion of a non-loadable driver to a loadable driver requires an initialization or "wrapper" module to be written. The module `zzinit.c` below is an example of a wrapper module that contains the same kind of information ordinarily provided by a config file and by the linker. Almost all wrappers are identical to the example below. Usually, only the actual structure initialization values are different.

The following module is a typical example of an initialization routine for a driver named `zz` that has one controller and one device on that controller.

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/buf.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sundev/mbvar.h>
#include <sun/autoconf.h>
```

```
#include <sun/vddrv.h>

extern zzopen(), nulldev(), zzstrategy(), zzdump();
extern zzsize(), zzread(), zzwrite(), zzioctl();
extern zzint(), nodev(), seltrue();

extern struct mb_driver zzcdriver;    /* defined in driver */

/*
 * Driver block device entry points (normally in <sun/conf.c>)
 */
struct bdevsw zzbdev = {
    zzopen, nulldev, zzstrategy, zzdump, zzsize, 0
};

/*
 * Driver character device entry points (normally in <sun/conf.c>)
 */
.
struct cdevsw zzcdev = {
    zzopen, nulldev, zzread, zzwrite, zzioctl, nodev,
    nulldev, seltrue, 0
};

/*
 * Controller structure (normally in ioconf.c) (see <sundev/mbvar.h>)
 */
struct mb_ctlr zzcctlr[] = {
    &zzcdriver, 0, 0, (caddr_t) 0x00001000, 2, 6,
    SP_ATMEM, 0
};

/*
 * Device structure (normally in ioconf.c) (see <sundev/mbvar.h>)
 */
struct mb_device zzcdevice[] = {
    &zzcdriver, 0, 0, 0, (caddr_t) 0x00000000, 0, 0, 0x0,
    0, 0x0
};

/*
 * The following structure is defined in <sun/vddrv.h>
 *
 * If the number of controllers is 0, then the address of the
 * controller structure array must be NULL. Similarly, if the number
 * of devices is 0, then the address of the device structure array
 * must be NULL. The bdevsw or cdevsw entries can be NULL if there
 * is no block or character device for the driver.
 */
struct vdldrv vd = {
    VDMAGIC_DRV,      /* Type of module. This one is a driver. */
    "zzdrv",          /* Name of the module. */
    zzcctlr,          /* Address of the mb_ctlr structure array */
```

```
    &zzcdriver,         /* Address of the mb_driver structure */
    zzcdevice,          /* Address of the mb_device structure array */
    1,                  /* Number of controllers */
    1,                  /* Number of devices */
    &zzbdev,            /* Address of the bdevsw entry */
    &zzcdev,            /* Address of the cdevsw entry */
    0,                  /* Block device number.  0 means let system choose. */
    0,                  /* Char. device number.  0 means let system choose.*/
};

/*
 * This is the driver entry point routine.  The name of the default entry point
 * is xxxinit.  It can be changed by using the "-entry" command to modload.
 *
 * inputs:  function code - VDLOAD, VDUNLOAD, or VDSTAT.
 *      pointer to kernel vddrv structure for this module.
 *      pointer to appropriate vdioctl structure for this function.
 *      pointer to vdstat structure (for VDSTAT only)
 *
 * return:  0 for success.  VDLOAD function must set vdp->vdd_vdtab.
 *      non-zero error code (from errno.h) if error.
 *
 */

xxxinit(function_code, vdp, vdi, vds)
    unsigned int function_code;
    struct vddrv *vdp;
    addr_t vdi;
    struct vdstat *vds;
{
    switch (function_code) {
        case VDLOAD:
            vdp->vdd_vdtab = (struct vdlinkage *)&vd;
            return (0);
        case VDUNLOAD:
            return (unload(vdp, vdi));
        case VDSTAT:
            return (0);
        default:
            return (EIO);
    }
}

static unload(vdp, vdi)
        struct vddrv *vdp;
        struct vdioctl_unload *vdi;
{
    extern struct buf zztab;

    struct buf *dp;

    dp = &zztab;
    if (dp->b_actf) {
```

```
        return(-1); /* The driver still has an active request. */
    }

    /* The driver can do any device shutdown stuff that it needs to do */

    return(0);
}
```

Your driver routines can be placed in the wrapper module if you like. If your driver is big, it is more appropriate to break it into several modules.

If you decide to place your driver in the wrapper module, then the driver can be compiled with the following command line:

```
example# cc -c -O -Dsun386 -Di386 -DTTYSOFTCAR -DWEITEK \
    -DVDDRV -DCRYPT -DVPIX -DIPCSHMEM -DIPCSEMAPHORE \
    -DIPCMESSAGE -DLOFS -DNFSSERVER -DNFSCLIENT -DUFS \
    -DINET -DSUN386 -DKERNEL -Umc68000 -Di386bug zzinit.c
```

However, if the driver consists of more than one module, then you must use the link editor, `ld`(1), with the `-r` option to preserve relocation information. For example you might type:

```
example# cc -c -O -Dsun386 -Di386 -DTTYSOFTCAR -DWEITEK \
    -DVDDRV -DCRYPT -DVPIX -DIPCSHMEM -DIPCSEMAPHORE \
    -DIPCMESSAGE -DLOFS -DNFSSERVER -DNFSCLIENT -DUFS \
    -DINET -DSUN386 -DKERNEL -Umc68000 -Di386bug zzinit.c

example# cc -c -O -Dsun386 -Di386 -DTTYSOFTCAR -DWEITEK \
    -DVDDRV -DCRYPT -DVPIX -DIPCSHMEM -DIPCSEMAPHORE \
    -DIPCMESSAGE -DLOFS -DNFSSERVER -DNFSCLIENT -DUFS \
    -DINET -DSUN386 -DKERNEL -Umc68000 -Di386bug zz1.c

example# cc -c -O -Dsun386 -Di386 -DTTYSOFTCAR -DWEITEK \
    -DVDDRV -DCRYPT -DVPIX -DIPCSHMEM -DIPCSEMAPHORE \
    -DIPCMESSAGE -DLOFS -DNFSSERVER -DNFSCLIENT -DUFS \
    -DINET -DSUN386 -DKERNEL -Umc68000 -Di386bug zz2.c

example# ld -r -o zz.o zzinit.o zz1.o zz2.o
```

Thus the object module can be created either by the `cc`(1) command, when the driver resides in one module, or by the `ld`(1) command, when the driver resides in several modules.

In either case the resulting object file (`zzinit.o` or `zz.o`) is a normal COFF file and can then be used with the `modload` command.[1] The driver is just like

[1] "COFF" = Common Object File Format, a UNIX object-file standard to which Sun386i assembler and link-editor output files (normally `a.out`) comply. See `coff`(5).

any other program, except its text segment starts somewhere in the range `0xFD000000` to `0xFE000000`.