sun microsystems

Writing Device Drivers *for the* Sun Workstation

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

Part No: 800-1195-01
Revision E of 15 May, 1985

Acknowledgements

Revision History

Rev	Date	Comments
A	15 July 1983	First release of this Manual as part of the *System Internals Manual for the Sun Workstation.*
B	15 August 1983	Minor corrections.
C	15 November 1983	Minor corrections.
D	19 November 1984	Minor corrections.
E	15 May 1985	Separated out of the *System Internals Manual for the Sun Workstation* to form a standalone manual. Added narrative to deal with VMEbus and support for vectored interrupts.

Contents

Chapter 1 Introduction **1-1**
1.1. General Overview 1-1

Chapter 2 General Hardware and Software Topics **2-1**
2.1. Device Names and Device Numbers — The /dev Directory 2-1
2.2. The Sun Hardware and the Multibus or VMEbus 2-4
2.2.1. Multibus Memory Address Space and I/O Address Space 2-5
2.2.2. Byte Ordering Issues 2-5
2.2.3. Things to Watch for in Multibus Boards 2-7
2.3. DMA Devices 2-8
2.3.1. Sun Main Bus DVMA 2-8
2.4. Allocation of Multibus Memory and I/O in the Sun System 2-9
2.5. Allocation of VMEbus Memory in the Sun System 2-12
2.6. Interrupt Vector Assignments 2-13
2.7. Main Bus Resource Management 2-14
2.8. Getting the Board Working and in a Known State 2-16
2.8.1. Using the Sun CPU PROM Monitor 2-17

Chapter 3 Device Drivers **3-1**
3.1. User Address Space versus Kernel Address Space 3-1
3.2. User Context and Interrupt Context 3-1
3.3. Device Interrupts 3-2
3.4. Interrupt Levels 3-3
3.5. Vectored Interrupts and Polling Interrupts 3-3
3.6. Some Common Service Functions 3-6
3.6.1. Timeout Mechanisms 3-6
3.6.2. Sleep and Wakeup Mechanism 3-6
3.6.3. Raising and Lowering Interrupt Priorities 3-7
3.6.4. Main Bus Resource Management Routines 3-8
3.7. Kernel printf Function 3-8
3.7.1. Macros to Manipulate Device Numbers 3-8
3.8. Overall Layout of a Device Driver 3-8
3.9. A Very Basic Skeleton Device Driver 3-9
3.10. General Declarations in Driver 3-11
3.11. Autoconfiguration Procedures 3-12
3.12. Probe Routine 3-12
3.13. Open and Close Routines 3-13

3.14. Read and Write Routines 3-15
3.14.1. Some Notes About the UIO Structure 3-16
3.15. Skeleton Strategy Routine 3-16
3.16. Skeleton Start Routine — Initiate Data Transfers 3-17
3.17. Interrupt Routines 3-18
3.18. Ioctl Routine 3-19
3.19. Devices That Do DMA 3-19
3.20. Multibus or VMEbus DVMA 3-19
3.21. Changes to the Driver 3-20
3.22. Errors 3-21
3.23. Memory Mapped Devices 3-22

Chapter 4 Configuring the System to Add Skeleton Driver **4-1**

Appendix A Summary of Functions **A-1**
A.1. Standard Error Numbers A-1
A.2. Device Driver Routines A-1
A.2.1. Autoconfiguration Routines A-1
A.2.1.1. `probe` — Determine if Hardware is There A-1
A.2.2. Open and Close Routines A-1
A.2.2.1. `open` — Open a Device for Data Transfers A-2
A.2.2.2. `close` — Close a Device A-2
A.2.3. Read, Write, and Strategy Routines A-2
A.2.3.1. `read` — Read Data from Device A-2
A.2.3.2. `write` — Write Data to Device A-3
A.2.3.3. `strategy` Routine A-3
A.2.3.4. `minphys` — Determine Maximum Block Size A-3
A.2.4. `ioctl` — Special Interface Function A-4
A.2.5. Low Level Routines A-4
A.2.5.1. `intr` — Handle Vectored Interrupts A-4
A.2.5.2. `poll` — Handle Polling Interrupts A-4
A.3. Common Service Routines A-4
A.3.1. `sleep` — Sleep on an Event A-5
A.3.2. `wakeup` — Wake Up a Process Sleeping on an Event A-5
A.3.3. `mbsetup` — Set Up to Use Main Bus Resources A-5
A.3.4. `mbrelse` — Free Main Bus Resources A-5
A.3.5. `physio` — Lock in User's Buffer Area A-6
A.3.6. `iowait` — Wait for I/O to Complete A-6
A.3.7. `iodone` — Indicate I/O Complete A-6
A.3.8. `pritospl` — Convert Priority Level A-6
A.3.9. `spl`*n*`()` — Set Specific Priority Level A-6
A.3.10. `splx` — Reset Priority Level A-7
A.3.11. `uiomove` — move data to or from the uio structure A-7
A.3.12. `ureadc` and `uwritec` — transfer bytes to or from a uio structure A-7
A.3.13. `peek`, `peekc` — Check Whether an Address Exists and Read A-8

A.3.14. `poke, pokec` — Check Whether an Address Exists and Write A-8
A.3.15. `geteblk` — Allocate Dynamic Buffer A-8
A.3.16. `brelse` — Free Dynamic Buffer A-8
A.3.17. `swab` — Swap Bytes A-8

Appendix B Sample Drivers **B-1**

Tables

Table 2-1 A sample listing of the /dev Directory 2-2
Table 2-2 Character Device Number Assignments 2-4
Table 2-3 Sun-2 Multibus Memory Map 2-10
Table 2-4 16-bit VMEbus Address Space Blocks 2-12
Table 2-5 24-bit VMEbus Address Space Blocks 2-12
Table 2-6 VMEbus Address Assignments for Individual Devices 2-13
Table 2-7 Vectored Interrupt Assignments 2-14

Chapter 1

Introduction

This document is a guide to adding software drivers for new devices to the kernel.

One of the UNIX† Operating System's major services to application software is a device-independent view of the hardware that stores and retrieves data and communicates with the outside world. The interface between UNIX application software and a given piece of raw hardware is provided by a ***device driver*** for that piece of hardware. A device driver provides an interface between the UNIX operating system's device-independent scheme of things and the special characteristics of a particular piece of hardware.

1.1. General Overview

The kernel supplied with the Sun system is a ***configurable kernel***, meaning that it is possible (within limits) to make changes to the kernel and to add new device driver modules. A detailed explanation of how to configure and build a kernel is in ***Building UNIX Systems with Config*** in the *System Manager's Manual*.

This document is aimed at the Sun user who has some expertise in writing UNIX device drivers, and who wishes to connect a new Multibus or VMEbus device to the Sun system. The UNIX system that runs on the Sun Workstation supports several different types of devices, and the scope of this document is limited to writing device drivers for the kinds of devices not already supplied by Sun. If you have no previous experience writing UNIX device drivers, you should expect to seek some advice from the Sun technical support organization or an outside consultant experienced in writing UNIX drivers. We can classify devices and their drivers into seven major categories:

1. Co-processors.
2. Disks and tapes.
3. Network interface drivers such as Ethernet or X.25.
4. Serial communications multiplexors.
5. General DMA devices such as driver boards for raster-oriented printers or plotters.
6. Programmed I/O devices.

† **UNIX is a trademark of Bell Laboratories.**

7. Frame buffers.

This manual ***only*** addresses devices and drivers in categories 5, 6, and 7. There is a wide range of devices which Sun does not support for which you might want to write a device driver. This document is primarily concerned with creating device drivers for devices such as parallel interfaces, analog to digital (A/D) converters, digital to analog (D/A) converters, interfaces to special outboard processors, frame buffers, memory-mapped graphics boards, and so on. Such devices can be cast into the model of ***unstructured*** or ***character*** I/O devices in the UNIX I/O system scheme, as opposed to block I/O devices that support a UNIX file system. Character I/O devices may support ***read*** and ***write*** operations, and may provide an ***ioctl*** interface for controlling the devices. Such devices may also provide for being mapped into the user's virual address space by supporting the ***mmap*** system call.

This document ***does not*** address devices and drivers in categories 1 thru 4. In particular, the considerations in writing device drivers for disks, tapes, serial communication devices, and local network interface drivers are quite involved — we do not discuss the construction of such drivers in this document. Most Sun customers should find that the extensive use of standards in the Sun product line should allow them to use hardware interfaces already provided by Sun to drive such peripherals.

To add a new hardware device-controller and its device driver to the system you must:

1. Get the device controller hardware into a state where you know it works as advertised — it is ***extremely*** difficult to debug your device driver software (step 4 below) if the hardware is not known to be working,
2. Write the device driver itself,
3. Add it to the system configurator's data base, describe a system containing the driver, and compile this system containing the new device driver,
4. Debug the driver.

Chapter 2 is a general overview of the hardware and software environment provided by the Sun Workstation.

Chapter 3 is a description of the I/O system and device drivers. Chapter 3 provides a model of a very simple device driver and describes the issues involved in programming device drivers on the Sun system.

Chapter 4 is a description of how to add a new device driver to the kernel.

Finally, samples of actual drivers are included with this document so that the reader can see how the actual code is used. The drivers we have included as samples are:

cgone	A simple memory-mapped driver for the black and white framebuffer.
sky	A simple programmed I/O driver for the SKY floating-point board.
vp	A DMA device driver for the Versatec printer/plotter.

Hint: Spend as much time as you need in the Sun Workstation PROM monitor poking, prodding and cajoling your device until you are thoroughly familiar with its behavior. This will save you a lot of grief later. There is a discussion a little later on the kinds of things you can do with the PROM monitor.

Chapter 2

General Hardware and Software Topics

2.1. Device Names and Device Numbers — The /dev Directory

All devices and special files are defined externally in the */dev* directory. Devices are characterized by a major device number, a minor device number, and a class (block or character). When a file of any type is opened, the device driver to call is obtained from the entry in the */dev* directory. Entries in the */dev* directory are created via the **mknod**(8) (make a node) administration command. Here is a fragment of what the */dev* directory looks like from an **ls −l** command:

Table 2-1: A sample listing of the /dev Directory

Type	*permissions*	*size*	*owner*	*major #*	*minor #*	*date*	*name*
c	rw--w--w-	1	henry	0,	0	Feb 21 09:45	console
c	rw-r--r--	1	root	3,	1	Dec 28 16:18	kmem
c	rw-------	1	root	3,	4	Jan 13 23:07	mbio
c	rw-------	1	root	3,	3	Jan 13 23:07	mbmem
c	rw-r--r--	1	root	3,	0	Dec 28 16:18	mem
c	rw-rw-rw-	1	root	13,	0	Dec 28 16:18	mouse
c	rw-rw-rw-	1	root	3,	2	Feb 22 16:40	null
c	rw-------	1	root	9,	0	Dec 28 16:19	rxy0a
c	rw-------	1	root	9,	1	Dec 28 16:19	rxy0b
							⋮
c	rw-------	1	root	9,	6	Feb 25 1984	rxy0g
c	rw-------	1	root	9,	7	Dec 28 16:19	rxy0h
b	rw-------	1	root	3,	0	Feb 25 1984	xy0a
b	rw-------	1	root	3,	1	Jan 17 20:12	xy0b
							⋮
b	rw-------	1	root	3,	6	Dec 28 16:19	xy0g
b	rw-------	1	root	3,	7	Dec 28 16:19	xy0h

The connection between the specific device name in the */dev* directory is made through two C structures named ***bdevsw*** (block device switch table) and ***cdevsw*** (character device switch table) in the file called ***conf.c***. When you add a new device driver you must add entries to the corresponding structure. Since we are discussing only character-oriented devices in this manual, you can ignore the ***bdevsw*** structure and concentrate on the ***cdevsw*** structure.

Application programs make calls upon the operating system to perform services such as opening a file, closing a file, reading data from a file, writing data to a file, and other operations that are done in terms of the file interface. The operating system code turns these requests into specific requests on the device driver involved with that particular file. The glue between the specific file operation involved and the device driver entry-point is through the ***bdevsw*** and ***cdevsw*** tables.

Entries in ***bdevsw*** or ***cdevsw*** contain an array of entry points into the device drivers. The position in the structure corresponds to the major device number assigned to the device. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The ***cdevsw*** table specifies the interface routines present for character devices. Each character device may provide seven functions: ***open***, ***close***, ***read***, ***write***, ***ioctl***, ***select***, and ***mmap***. If a call on the routine should be ignored, (for example *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev*; if it should be considered an error, (for example *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the array of *tty* structures associated with the driver.

Here is what the declaration of the character device switch looks like. Each entry (row) is the only link between the main unix code and the driver. The initialization of the device switches is in the file *conf.c*.

```
struct cdevsw
{
int       (*d_open) ();       /*  routine to call to open the device   */
int       (*d_close) ();      /*  routine to call to close the device  */
int       (*d_read) ();       /*  routine to call to read from the device  */
int       (*d_write) ();      /*  routine to call to write to the device  */
int       (*d_ioctl) ();      /*  special interface routine  */
int       (*d_stop) ();
int       (*d_reset) ();
struct tty *d_ttys;           /*  tty structure  */
int       (*d_select) ();     /*  routine to call to select the device  */
int       (*d_mmap) ();       /*  routine to call to mmap the device  */
};
```

Only teletype-like devices (such as the the console driver, the *mti* driver, and the *zs* driver) use the *tty* structure. All other devices set this field to zero.

And here is a typical line from the *conf.c* file which fills in the requisite pointers in the *cdevsw* structure:

```
                •
                •
                •
                All the other cdevsw entries between 0 and 13 appear first
    {
cgoneopen,      cgoneclose,     nodev,  nodev,  /*14*/
cgoneioctl,     nodev,  nodev,  0,
seltrue,        cgonemmap,
    },

                Then all the other cdevsw entries from 15 upwards
                •
                •
                •
```

In the Sun system, a number of devices in *cdevsw* are preassigned. The table below shows the assignments to date. Those major device numbers shown as 'For Local Use' are available for user-written device drivers.

Table 2-2: Character Device Number Assignments

Character-Device Number Assignments		
Major Device Number	*Device Abbreviation*	*Device Description*
0	cn	Sun Console
1	unused	no device
2	sy	Indirect TTY
3	Memory special files	
4	ip	Raw Interphase Disk Device
5	tm	Raw Tapemaster Tape Device
6	vp	Ikon Versatec Parallel Controller
7	Not available	
8	ar	Archive Tape Controller
9	xy	Raw Xylogics Disk Device
10	mti	Systech MTI
11	des	DES Chip
12	zs	On board UARTS
13	ms	Mouse
14	cg	Color Graphics Board
15	win	Window Pseudo Device
16	ii	INGRES lock device
17	sd	Raw SCSI disk
18	st	Raw SCSI tape
19	nd	Raw Network Disk Device
20	pts	Pseudo TTY
21	ptc	Pseudo TTY
22	fb	Monochrome Video board
23	ropc	RasterOp Chip
24	sky	SKY Floating Point Board
25	pi	Parallel input device
26	bwone	Sun-1 Monochrome frame buffer
27	bwtwo	Sun-2 Monochrome frame buffer
28	vpc	Parallel driver for Versatec printer
29	kbd	Sun keyboard driver
30	xt	Xylogics 472 Tape Controller
31	cgtwo	Sun-2 Color Frame Buffer
40–??	**For Local Use**	

2.2. The Sun Hardware and the Multibus or VMEbus

The Sun system hardware is built around the IEEE-P796 Multibus or the VMEbus in some models. This section discusses several issues relevant to the Multibus or VMEbus and devices that can be obtained for it.

There are data structures in the kernel's interface to device drivers that are named ***mb something or other***. These structures are for setting up linked lists of controllers and devices and the information associated with them. The **mb** used to stand for Multibus, but with the advent of Sun systems supporting the VMEbus, the Multibus, and possibly VMEbus based systems with VMEbus to Multibus adaptors, the **mb** has been used to stand for 'Main bus'.

2.2.1. Multibus Memory Address Space and I/O Address Space

Although Sun uses Motorola MC68000 family processors for its products, the systems are actually built around the IEEE-P796 Multibus. The MC68000 processors do what is known as 'memory-mapped' input-output in that you just store data somewhere or fetch data from somewhere to transfer data to or from a peripheral device or memory — there is no distinction between the memory and peripherals. The Multibus, on the other hand, was originally designed for processors that have one kind of instruction for storing data in memory or fetching data from memory (instructions such as MOV), and a different kind of instruction (such as IN and OUT) for transferring data to or from peripheral devices. Thus the Multibus has the notion of two separate address spaces:

Multibus memory space
is simply used for memory or devices that look like memory, in that you talk to such devices simply by writing data to memory locations or reading from memory locations. The Sun color controller board is a good example of a device that is addressed as memory in the Multibus memory address space. Devices that look like memory are called 'memory mapped' devices.

Multibus I/O address space
is another 'space' that is typically used for device control registers. Devices using the I/O address space are said to be 'I/O mapped' devices.

This concept of two different address spaces derives from the Intel 8080 family of processors. The MC68000 family doesn't have this separation of memory and I/O, but treats the entire universe as one address space. The Sun memory management hardware can map any portion of the system's address space to the Multibus memory space or the Multibus I/O address space. Ultimately, the different kinds of address-space end up just waggling different control lines on the Multibus.

Be aware though, that the memory space of the Multibus is designed for a 20-bit r or a 24-bit addressing scheme (Sun uses 20-bit addresses), whereas the I/O space of the Multibus is only an 8-bit or a 16-bit addressing scheme (Sun uses 16-bit addresses), and some older Multibus boards only accept 8-bit I/O addresses.

2.2.2. Byte Ordering Issues

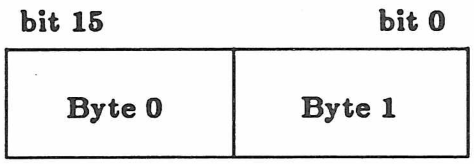
The Sun processor is a Motorola MC68000 processor, built on an IEEE-P796 Multibus board. IEEE-P796 and Motorola do not agree on the addressing of bytes in a word. IEEE-P796 and Motorola both agree that there are 16 bits in a word and that is about all they agree on. The disagreement about which end of the word contains byte 0 leads us into two separate problems, with two separate fixes you must apply:

1. You are moving a single ***byte*** across the interface between the MC68000 and the P796 Bus. Because of the disagreement about which end of the word the byte actually appears in, you have to toggle the least significant bit of the ***byte*** address.

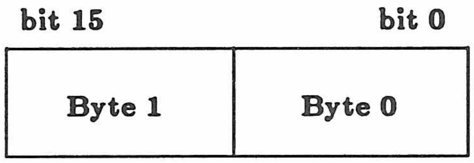
2. You are moving a whole 16-bit ***word*** across the interface between the MC68000 and the P796 Bus. This word actually contains a byte structure destined for the device on the other side of the bus. The device will interpret the byte-order different from what you thought, and so in this case you must physically swap the bytes in the word before you ship the word across the bus interface.

Here are a few pictures describing the problem in detail:

Motorola Byte Ordering



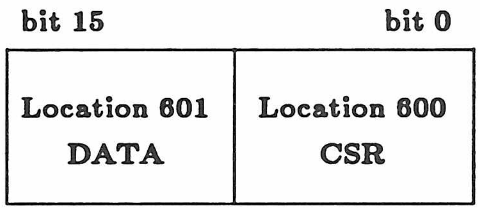
IEEE-P796 Byte Ordering



That is, Motorola places byte 0 in bits 8 thru 15 of the word, whereas IEEE-P796 places byte 1 in those bits. The only place where this causes trouble is when you are moving a single ***byte*** across the interface between the MC68000 and the Multibus. If you did everything with the 68000, or everything on the Multibus, there would never be any conflict, since things would be consistent. However, as soon as you cross the boundary between them, the byte order is reversed. What this means in practice is that you have to toggle the least significant bit of the address of any ***byte*** destined for the Multibus.

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 68000 processor views that device as looking like this:

Hypothetical Board Registers

bit 15	**bit 0**
Location 600 **CSR**	**Location 601** **DATA**

so that if you were to read location 600 from the point of view of the 68000 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 like this:

```
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 MC68000 and the Multibus.

Take special care when a Multibus device structure contains mixed bytes and words. Many of the Multibus device controllers on the market are geared up 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 is really a word quantity (such as a 16-bit address which must be two-byte aligned in the MC68000) starting on an odd byte boundary. Some of the device drivers 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 up for the 8086 family with its segmented adress 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, look at the code for ***vp.c,*** (attached as an appendix to this document).

2.2.3. Things to Watch for in Multibus Boards

Although there are a myriad of vendors offering Multibus products, be aware 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 bits wide 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 only 8 bits of I/O space. In particular, watch for the following things:

- 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 at 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 in our system, the addresses that such a board responds to would be replicated sixteen times through the one-

Megabyte address space on the Multibus.

- A memory-mapped Multibus board that uses 24-bit addressing (thereby using the P2 bus on the backplane) must use a P2 bus that is physically isolated from the P2 bus that any Sun boards use. See the ***Sun Configuration Guide*** for information on configuring boards in the backplane.
- 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 can't cope and only use eight-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.
- Watch out for boards containing PROM code that expects to find a CPU busmaster with an Intel 8080, 8085, or 8086 on it. Such boards are of course useless in the Sun System.
- Take special care to determine how the board generates interrupts. A board should put up an interrupt when the device it is controlling is ready for more data ***and*** the board is ready for more data — we have experienced designs where the interrupt indicated that the board was ready, or the device was ready, but not both at once. A board should ideally come up in its power up state with interrupts disabled and only start interrupting when told to. There should also be a way to determine that a board has actually generated an interrupt. Finally, an interrupting board should shut off its interrupt when it is told to.

2.3. DMA Devices

Many device controller boards are capable of what is known as Direct Memory Access or DMA. This means that the processor tells the device controller the address in memory where a data transfer is to take place, plus the length of the data transfer, and then tells the device controller to start the transfer. The data transfer then takes place without further intervention on the part of the processor. When the transfer is complete, the device controller interrupts to say that the transfer is finished.

2.3.1. Sun Main Bus DVMA

Direct Virtual Memory Access (DVMA) is a mechanism provided by the Sun memory management unit that allows DMA from devices on the Main Bus to Sun processor memory, or from Main Bus master devices directly to Main Bus slaves without going through processor memory. DVMA uses the first 256K bytes of the Main Bus address memory address space to map addresses between Sun processor memory and the Main Bus memory address space.

On the Sun-2, the memory management unit is always listening to the Main Bus for memory references. When a request to read or write Multibus memory between addresses 0 and 256K comes up, the DVMA hardware takes the address, adds `0xF00000` to it,[1] and goes through the kernel memory map to find the location in processor memory that will be used. On VMEbus systems, DVMA responds to the least significant 1 Megabyte of the VMEbus physical address space (`0x000000` to `0x100000`) and maps it into the most significant 1 Megabyte of system context virtual address space (`0xf00000`). Thus if you wish to do DMA over the Main Bus, you must

[1] The system places the Main Bus memory address space at location 0xF00000 in the virtual address space.

make the appropriate entries in the kernel memory map. As you might expect, there are functions to help with this chore.

2.4. Allocation of Multibus Memory and I/O in the Sun System

Here are some simple rules for the way that Multibus memory resources are doled out in the Sun system.

No devices may be assigned addresses below 256K in Multibus memory space or below 1 Megabyte in VMEbus space — the CPU uses these addresses for DVMA.

Devices that interface to the Sun system do so either through I/O registers in Multibus address space, or through the Multibus memory space. In some cases, a device may have both I/O registers and memory on the Multibus. The Sun system makes the assumption that any address lower than 64K is a Multibus I/O address. This is a reasonable assumption given that user-installed Multibus memory cannot appear in this region of the address space anyway. This assumption is carried through into the autoconfiguration routines in that addresses less than 64K are automatically mapped to the Multibus I/O address space.

To configure such a device,

1. the ***probe*** function for the device driver must return the amount of Multibus memory space that the device uses,

2. Multibus I/O address space is at 'mbio' and may be addressed as such. Alternatively, use virtual address `0xeb0000`.

3. the autoconfiguration utility (config) can not deal with I/O address space at the same time as memory address space for the same device.

The table on the next page shows a map of how Multibus memory is laid out in the Sun system.

Table 2-3: Sun-2 Multibus Memory Map

Address	Device
0x00000	DVMA Space
	.
	. (256 Kbytes)
	.
0x3f800	DVMA Space
0x40000	Sun Ethernet Memory (#1)
	.
	. (256 Kbytes)
	.
0x7f800	Sun Ethernet Memory (#1)
0x80000	SCSI (#1)
	.
	. (16 Kbytes)
	.
0x83800	SCSI (#1)
0x84000	SCSI (#2)
	.
	. (16 Kbytes)
	.
0x87800	SCSI (#2)
0x88000	Sun Ethernet Control Info (#1)
	.
	. (16 Kbytes)
	.
0x8b800	Sun Ethernet Control Info (#1)
0x8c000	Sun Ethernet Control Info (#2)
	.
	. (16 Kbytes)
	.
0x8f800	Sun Ethernet Control Info (#2)
0x90000	*** **FREE** ***
	.
	. (64 Kbytes
	.
0x9f800	*** **FREE** ***

Address	Device
0xa0000	Sun Ethernet Memory (#2)
	.
	. (64 Kbytes)
	.
0xaf800	Sun Ethernet Memory (#2)
0xb0000	*** **FREE** ***
	.
	. (64 Kbytes)
	.
0xbf800	*** **FREE** ***
0xc0000	Sun Model 100 or Model 150 Frame Buffer
	.
	. (128 Kbytes)
	.
0xdf800	Sun Model 100 or Model 150 Frame Buffer
0xe0000	3COM Ethernet (#1)
0xe0800	3COM Ethernet (#1)
0xe1000	3COM Ethernet (#1)
0xe1800	3COM Ethernet (#1)
0xe2000	3COM Ethernet (#2)
0xe2800	3COM Ethernet (#2)
0xe3000	3COM Ethernet (#2)
0xe3800	3COM Ethernet (#2)
0xe4000	*** **FREE** ***
	.
	. (16 Kbytes)
	.
0xe7c00	*** **FREE** ***
0xe8000	Sun Color
	.
	. (64 Kbytes)
	.
0xf7800	Sun Color
0xf8000	*** **FREE** ***
	.
	. (16 Kbytes)
	.
0xff800	*** **FREE** ***

2.5. Allocation of VMEbus Memory in the Sun System

This section defines blocks of address space which will be used for specific devices. Assignments for individual devices appear at the bottom.

Table 2-4: 16-bit VMEbus Address Space Blocks

16-bit VMEbus address space blocks			
Address Range	*Number of Pages Used*	*Allocated From*	*Description of Use*
0000-8000	16	Low	Reserved for OEM/user devices
8000-FFFF	16	High	Reserved for Sun devices

Note: The Multibus/VME Adapter will put cards into the same place in 16-bit VMEbus space as they were in Multibus I/O space. This may place the standard Multibus addresses for some cards into the OEM/user area on the VMEbus.

Table 2-5: 24-bit VMEbus Address Space Blocks

24-bit VMEbus address space blocks			
Address Range	*Number of Pages Used*	*Allocated From*	*Description of Use*
000000-100000	512		CPU board DVMA space
100000-200000	512		Reserved for the Future.
200000-300000	512	Low	Reserved for small Sun devices
300000-400000	512	High	Reserved for large Sun devices
400000-800000	2048	(Taken)	Reserved for huge Sun devices
800000-C00000	2048	High	Reserved for huge OEM/user devices
C00000-D00000	512	Low	Reserved for large OEM/user devices
D00000-E00000	512	High	Reserved for small OEM/user devices
E00000-F00000	512		Multibus-to-VMEbus memory space
E00000-FF0000	480		Reserved for the Future
FF0000-FFFFFF	32		Not addressable (CPU references 16-bit space)

32-bit-address VMEbus space can be dealt with later, when we have a CPU which can generate 32-bit-address accesses.

These same assignments apply to both 16-bit-data and 32-bit-data VMEbus accesses. Currently no Sun devices generate or respond to 32-bit-data references.

The 'Alloc from' field shows whether we allocate individual devices from the high end of the range or the low end. The idea is to keep the maximum size 'hole' in the middle in case we need to shift the boundary later.

Table 2-6: VMEbus Address Assignments for Individual Devices

Device	*Addressing*	*Address Space*
VMEbus SKY Board	16-bit	8000 - 8FFF
VMEbus SCSI Board	24-bit	200000 - 200800
VMEbus TOD Chip	24-bit	200800 - 2008FF
Sun-2 Color Board	24-bit	400000 - 4FF800

The VME Sky board occupies addresses 8000-9000 in 16 bit address space. The Sky board requires that the high nibble of the address be '8'.

2.6. Interrupt Vector Assignments

The table below shows the assignments of interrupts vectors for those devices which can supply interrupts through the VMEbus vectored interrupt interface.

Table 2-7: Vectored Interrupt Assignments

Vector Numbers			*Description*
64	*thru*	71	sc0, sc? — Serial Communications Controllers
72	*thru*	79	xyc0, xyc1, xyc? — Xylogics Disk Controllers
80	*thru*	95	future disk controllers
96	*thru*	99	tm0, tm1, tm? — TapeMaster Tape Controllers
100	*thru*	103	xtc0, xtc1, xtc? — Xylogics Tape Controllers
104	*thru*	111	future tape controllers
112	*thru*	115	ec? — 3COM Ethernet Controller
116	*thru*	119	ie? — Sun Ethernet Controller
120	*thru*	127	future ethernet devices
128	*thru*	131	vpc? — Systech VPC-2200
132	*thru*	135	vp? — Ikon Versatec Parallel Interface
136	*thru*	139	mti0, mti? — Systech Serial Multiplexors
140	*thru*	163	future serial devices
164	*thru*	175	future frame buffer devices
176	*thru*	179	sky0, ? — SKY Floating Point Board
180	*thru*	199	Reserved for Sun
200	*thru*	255	Reserved for Customer Use

2.7. Main Bus Resource Management

The following data structures in fact reflect the layout of information in the configuration file which we describe in a later part of this paper. Controllers and devices can be thought of as being attached to the Main Bus Certain kinds of devices (disks and tapes) are then thought of as being slaves to their controllers. This layout gives rise to three data structures whose descriptions exist in the header file */usr/include/sundev/mbvar.h.*

Main Bus The first data structure is the Main Bus header data structure. The fact that it is called 'Main Bus' is a complete red herring — it is simply a hook to hang all the other data structures on. The Main Bus data structure contains a list of controllers using this resource.

Controller Contains a list of structures that describe controllers. There is sometimes considerable confusion as to exactly what is a controller and what is a device. Essentially a *controller* is a piece of hardware that can control more than one device, but *only*

one data transfer can be active at a time. Each device controller on the Main Bus has a structure associated with it. The structure is called `mb_ctlr` and can be found in ***/usr/include/sundev/mbvar.h***.

Device Contains a list of devices. Each device driver has a data structure describing how the Main Bus resource-management routines view the driver. The per-driver data structure is called `mb_driver` and can be found in ***/usr/include/sundev/mbvar.h***. The device data structures are either hooked directly onto the Main Bus header structure, or they are hooked to controller structures in which case the devices are said to be ***slaves*** to their controllers. The device structure, `mb_driver`, is the really important data structure that you need to be concerned with when writing a driver. Here is the layout of the ***mb_driver*** structure:

```
struct mb_driver {
    int         (*mdr_probe)();          /* see if a driver is really there */
    int         (*mdr_slave)();          /* see if a slave is there */
    int         (*mdr_attach)();         /* setup driver for a slave */
    int         (*mdr_go)();             /* routine to start transfer */
    int         (*mdr_done)();           /* routine to finish transfer */
    int         (*mdr_intr)();           /* polling interrupt routine */
    int         mdr_size;                /* amount of memory space needed */
    char       *mdr_dname;               /* name of a device */
    struct      mb_device **mdr_dinfo;   /* backpointers to mbdinit structs */
    char       *mdr_cname;               /* name of a controller */
    struct      mb_ctlr **mdr_cinfo;     /* backpointers to mbcinit structs */
    short       mdr_flags;               /* want exclusive use of Main Bus */
    struct      mb_driver  *mdr_link;    /* interrupt routine linked list */
};
```

Here is a brief discussion of the fields in the `mb_driver` structure and what parts of it you need to fill in when declaring `mb_driver`:

`mdr_probe`
is a pointer to a ***probe*** function within your driver. ***Probe*** determines if the device for which this driver is written is really there in the system. Fill in this field only if your driver has a ***probe*** routine (it generally will).

`mdr_slave`
is a pointer to a ***slave*** function within your driver. Fill in this field for controllers that have more than one device. The ***slave*** function always returns a 1.

`mdr_attach`
is a pointer to an ***attach*** function within your driver. The ***attach*** function does preliminary setup work for a slave device. Typical applications include reading the label from a disk. Fill in this field only if there is an ***attach*** routine in your driver. In general, the drivers we are considering in this paper don't have ***attach*** routines, and so you fill in a zero (0) in this field.

`mdr_go`
`mdr_done`
are pointers to ***go*** and ***done*** functions within your driver. These fields are usually zero for the types of drivers we talk about in this paper. They are normally for disk drivers who can't afford to wait for `mbsetup`.

`mdr_intr`
is a pointer to a polling interrupt routine (function) within your driver. Fill in this field if

your driver actually has a polling interrupt routine (in general it will). If your driver doesn't have a polling interrupt routine, fill in a zero (0) in this field.

`mdr_size`
: is the size in bytes of the amount of memory that a memory-mapped device requires. This field ***must*** be filled in if ***mdr_maddr*** is used for a memory-mapped device.

`mdr_dname`
: is the name of the device for which this driver is written. This field takes the form of a regular null-terminated C string.

`mdr_dinfo`
: an array of pointers to ***mb_device*** structures. Auto configuration fills in the pointers, then the driver can access ***mb_device*** structures if it wants to.

`mdr_cname`
: is the name of the controller for which this driver is written. This field takes the form of a regular null-terminated C string. Fill in this field if you actually have a controller.

`mdr_cinfo`
: an array of pointers to ***mb_controller*** structures. Auto configuration fills in the pointers, then the driver can access ***mb_controller*** structures if it wants to.

`mdr_flags`
: consists of some flags, as follows:

  MDR_XCLU
  : needs exclusive use of bus

  MDR_DMA
  : device does Main Bus DMA

  MDR_SWAB
  : Main Bus buffer must be swabbed

  MDR_OBIO
  : device in on-board I/O space

  These flags must be OR'ed together if you wish to place any of that information there. Place a zero (0) in this field if none of the flags apply to this driver.

`mdr_link`
: This field is used by the autoconfiguration routines and is not for the driver's use.

2.8. Getting the Board Working and in a Known State

This section discusses getting the hardware device controller operational and in a known state,

Before you even ***think*** about writing any code you should check out the Multibus or VMEbus board by performing various tests.

First, make sure that the board is properly set up as defined in the vendor's manual. Things you have to select in general are:

- I/O register addresses for those boards that use I/O ports on the Multibus,
- Memory base address for those boards that use memory space on the Multibus,
- Interrupt level selection.

- Interrupt vector number for VMEbus devices.

Then, take your system down and power it off. Plug your Multibus or VMEbus board into the card cage and attempt to bring the system back up. If you cannot boot the system, then there is a problem such as the board not really working or the board responding at an address used by other boards in the system. You must resolve this problem before proceeding further.

Next take your system down again and see if the device responds. from the monitor, try some of the following things:

- Try reading from the board status register(s) if there are any.
- Try writing to the board control or data register(s) if there are any. Then try reading the data back to see if it got written properly (assuming that the board can read back what you wrote).
- Try sending data to the actual device itself through the board if this is possible.
- Switch the actual device offline and online and watch the status bits go on and off (if this is possible).

For example, if you have a line printer, try to print a line with a few characters. The section just below on *Using the Sun CPU PROM Monitor* has some hints on reading and writing device registers. Be aware that bit and byte ordering issues are critical in this process; the main reason for doing this step is to discover what the board really does. When you have developed confidence in how the board works you can proceed to write a driver for it.

2.8.1. Using the Sun CPU PROM Monitor

To do some of the poking around as described in the previous paragraphs, you can use the CPU PROM monitor whose commands are described in detail in the ***System Internals Manual***. The PROM monitor has commands for looking at memory locations. So if you have located your new Multibus or VMEbus board at a specific place in the address space, you could use the monitor to look at that place to see if there's anything there. For example, if you think your board has an I/O control register at location 0x600, you could use the monitor's 'open a byte location' command to look at that place in memory:

```
        > o eb0600
```

and so on. If you get a bus error timeout, the board isn't there and you have to go back to the manual to see if you've set the address jumpers correctly. Note that locations starting at eb0000 is where the PROM monitor maps the Multibus I/O space, hence the eb0600 in the example above to get at location 0600.

Here are a couple of notes about using the monitor to look at devices. When you use the Monitor's 'o' command to open a location, the Monitor ***reads*** the contents of that location and displays them before asking you what you want to put there (if anything). Now some devices (the Intel 8251A and the Signetics 2651 immediately spring to mind) use the same location (register) to address ***two*** separate internal mode registers, and the chip has internal state-logic that sequences around them in 1-2-1-2... order. So suppose you want to put something in mode register 1 of the 8251? You open that location, the Monitor displays the contents, and you then write the byte. Being cautious, you then open that location again and bingo! the data you wrote isn't there — it's in the second register because the action of ***reading*** that location sequenced you on to the second register. To do this thing right you have to use the Monitor's 'write without looking' facility and then read the locations back later to check.

Another chip that has internal sequencing logic of this type is the NEC PD7201 PCC. This chip has a a bunch of internal data registers. You load data-register 0 with the number of the data register into which the next byte of data will go, then you send the byte of data and it goes into that specific data register, and then you are back to data-register 0 again, all done with internal sequencing logic.

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***!

Chapter 3

Device Drivers

This section discusses the major issues in creating a device driver for the system.

A first step in writing a device driver is deciding what sort of interface the device should provide to the system. The way in which `read` and `write` operations should occur, the kinds of control operations provided via `ioctl`, and whether the device can be mapped into the user's address space using the `mmap` system call, should be decided early in the process of designing the driver.

Device drivers have access to the memory management and interrupt handling facilities of the UNIX system. The device driver is called each time the user program issues an `open`, `close`, `read`, `write`, `mmap`, or `ioctl` system call. The device driver can arrange for I/O to happen synchronously, or it can allocate buffers so that output can proceed while the user process runs, or gather input while the user process is not waiting.

3.1. User Address Space versus Kernel Address Space

A device driver is a part of the kernel. The kernel uses a completely different virtual address space from the virtual address space that a user process uses. When a device driver function is invoked through a system call, the driver must often map data from the user virtual address space to the kernel's virtual address space (most often in the case of some DMA devices). Functions and macros are provdied to allow this 'dual' mapping of data. Normally the kernel can only access data that is addressable in its own address space.

3.2. User Context and Interrupt Context

A device driver has a ***top half*** and a ***bottom half***. The top half is the part of the driver that runs only in the context of a user process making requests on the driver. The top half of a driver can start tasks which can cause long delays during which the system would want to switch to another process and continue doing useful work. When this happens the driver uses the `sleep` primitive to wait for a particular event to occur. Thus if a user program issues a `read` on (say) an A/D converter, the process would normally `sleep` until some input arrived. The driver could also use the ***iowait*** call for transfers that have already started.

The ***bottom half*** of a device driver is the part that runs at interrupt level. Thus in an A/D converter driver, the converter might interrupt when a sample was available. The bottom half of the driver could then store the data in a buffer and ***wakeup*** any user process sleeping in the top half so that that process could retrieve the data. If there was no user process sleeping in the top half, the ***wakeup*** would do nothing, but the next process to `read` the A/D driver would find the data already there and would not have to `sleep`.

3.3. Device Interrupts

Each hardware device interrupts (that is, the device ***should*** interrupt) at some ***priority level***, trapping from wherever the system is currently executing, into the bottom half of the device driver at that priority level. This means that the ***top half*** of the device driver can be interrupted at any time by the bottom half of the driver. The top half and the bottom half share data structures which they wish to keep consistent. An example of such a data structure might be a pointer to a current buffer and a character counter. The top half of the driver must protect itself so that data structures can be updated as atomic actions, that is, the bottom half must not be allowed to interrupt during the time that the top half is updating some shared data structure. The way this protection is done is to bracket the critical sections of code (that updates or examines shared data structures) with a subroutine call that raises the processor priority to a level where the bottom half cannot interrupt. Such a piece of code looks like:

```
s = splN();
        critical section of code which cannot be interrupted
(void)splx(s);
```

Note here that we raised the processor priority level and then restored the processor priority level after the protected section of code. (Determining the correct ***hardware_priority*** will be discussed later.) One section of code that almost always needs to be protected is the section where the top half checks to see if there is any data ready for it to read, or whether it can write data or start the device. Since the device can interrupt at any time, the section of code that checks for input in this fashion is wrong:

```
if (no input ready)
        sleep (awaiting input, software_priority)
```

because the device might well interrupt while the `if` condition is being tested, or while the preamble code for the ***sleep*** function is being executed.

The above section of code must be rewritten to look like this:

```
s = splN();
while (no input ready)
        sleep (awaiting input, software_priority)
(void)splx(s);
```

If the top half executes the `sleep` system call, the bottom half will be allowed to interrupt, because the hardware priority level is reset to 0 as soon as the `sleep` context switches away from this process.

3.4. Interrupt Levels

In many cases it is possible to set the interrupt level a device will interrupt at by setting switches on the board. If so, you must decide what level this device is going to interrupt at. At first it may seem that your device is very high priority, but you must consider the consequences of locking out other devices:

- If you lock out the clock (level 5) time will not be accurate, and the UNIX scheduler will be suspended.
- If you lock out the on-board UARTS (level 6) characters may be lost.
- If you lock out the Ethernet (level 3), packets may be lost and retransmissions needed.
- If you lock out the disks (level 2), disk rotations may be missed.
- Level 1 is used for software interrupts and cannot be used for real devices.

In general, it is best to use level 2 to avoid the consequences of locking out other important system activities.

3.5. Vectored Interrupts and Polling Interrupts

This section contains a general discussion of device interrupts. When using Multibus devices, the Sun UNIX kernel was set up to use auto-vectored interrupts. With auto-vectoring, the interrupt vector used when a device interrupts is based only on the priority level of the device. A given configuration of boards in the card cage may result in more devices than there are interrupt levels available. This means that several devices may have to share the same interrupt level, and there must be a way to determine which device driver should handle the spcific interrupt. In systems that only have auto-vectored interrupts, the kernel 'polls' the polling interrupt routine of each device driver for the appropriate level until some driver eventually indicates that the interrupt was from a device that it is set up to handle.

Typically, a driver's polling interrupt routine is called *xx*`poll`, where *xx* is the two-letter name of the driver.

With the VMEbus we can take advantage of vectored interrupts. A vectored interrupt passes control directly to a short stub of code which then calls the appropriate interrupt routine, passing an argument to identify which device interrupted.

Typically, a driver's vectored interrupt routine is called *xx*`intr`, where *xx* is the two-letter name of the driver.

It is possible that a given device driver can support both vectored interrupts ***and*** polling interrupts. In such a case, the polling interrupt routine just calls the vectored interrupt routine to actually service the interrupt.

For devices which interrupt in a VMEbus based system, the vector ***number*** (as opposed to the vector address) is in the range 64 to 255. Note that vector numbers 200 and above are reserved for customer use only.

By default, when *xx*`intr` is called it is passed the controller or unit number of the device which interrupted. In addition it is possible for the driver to modify the variable passed to the controller.

There are cases where no separate *xx*`poll` routine is needed. The first case is where a driver ***knows*** that it supports a maximum of one device per system. In this case only a *xx*`intr` routine need exist and can provide the functionality of the *xx*`poll` routine. In this case *xx*`intr` is specified in the `mb_device` structure for the auto-vectored case and in the ***config*** input file for the vectored interrupt case. The *xx*`intr` routine should return a value for supporting the polling case.

The other case where the *xx*`poll` routine is not needed is when a driver will ***never*** support polling. In this (unlikely) case the interrupt routine specified in the `mb_device` should be 0 while *xx*`intr` is specified in the ***config*** input file.

Note that the first case points out the fact that in the easiest case nothing need be done to a driver to make it work with vectored interrupts, although some efficiency will be lost for the vectored interrupts. The *xx*`intr` routine which still implements polling can be called for the polling case and vectored interrupt case. However when the interrupt routine is called in the vectored case and the routine goes through polling all of its devices to find out which one interrupted unnecessary cycles are wasted since the interrupting unit number is passed to the routine. Thus it is suggested that all appropriate drivers be have separate *xx*`intr` and *xx*`poll` routines.

Another issue that may require changes to the driver is that of setting up the interrupting vector number. When using the VMEbus-Multibus adapter or certain VMEbus devices, the vector number is set by jumpers on the circuit board. But some devices require that software set up the device to tell which vector number to use on interrupts. Presently, the only place where this can be done is at "attach" time. The `mb_device` and `mb_ctlr` structures have `md_intr` and `mc_intr` fields respectively, which point to an array of `vec` structures (a value of NULL indicates that no vectors where specified in the config file). This array is terminated by an entry of all 0s for the last structure and is always guaranteed to have at least one non-zero entry.

The *xx*`attach` routine can look at the `mb_device` or `mb_ctlr` structure to find out the vector number (if any) specified in the config file.

xx`attach` must also examine the global "cpu" variable to determine if the CPU is capable of vectored interrupts (`cpu == SUN2_VME`). Based on this information the driver can then set the card to interrupt using the auto-vectors `if (cpu != SUN2_VME)` or if no vectored interrupts were specified (the vector for auto-vectoring is 24 + m[cd]_intpri). Otherwise it can set the card to use the vector number specified in the config file by using `m[cd]_intr->v_vec`.

Also, if the driver chooses to pass a value other than the unit number to the vectored interrupt routine, this is set up at ***xxattach*** time.

If the driver is capable of supporting multiple vectored interrupts, then it should perform any checking needed and install the vectors and/or the values to be passed as required. Remember that the order of the `vec` structures corresponds to the order specified in the ***config*** file. For any device which supports multiple vectored interrupts, the order of vector specifications in the ***config*** file is very important.

Thus a skeleton for a "typical" driver supporting both vectored and polling interrupts which uses software to set interrupt vectors might look like:

```
struct  mb_driver xxdriver = {
        xxprobe, 0, xxattach, 0, 0, xxpoll,
        sizeof (struct xx_device), "xx", xxinfo, 0, 0, 0,
};

/*
 * Attach routine - device xx must set vector number in software.
 * We know that there is a maximum one vectored interrupt to look at.
 */
xxattach(md)
        struct mb_device *md;
{
        register struct xxctlr *c = &xxctlrs[md->md_unit];

        if ((cpu == SUN2_VME) && md->md_intr) {
                /*
                 * use vectored interrupts, set up the interrupt vector
                 * plus set up to pass xxctlr structure pointer.
                 */
                c->c_addr->intvec = md->md_intr->v_vec;
                *(md->md_intr->v_vptr) = (int)c;
        } else {
                /* set up for using auto-vectoring */
                c->c_addr->intvec = 24 + md->md_intpri;
        }
        /* any other attach code */
}

/*
 * Handle xx interrupt - called from xxpoll and for vectored interrupts
 * Note that we expect to be called with a pointer to the controller.
 */
xxintr(c)
        struct xxctlr *c;
{
        /* handle the interrupt here */
}

/*
 * Handle xx polling auto-vectored interrupt
 */
xxpoll()
{
        struct xxctlr *c;
        int serviced = 0;

        for (c = xxctlrs; c < &xxctlrs[NXX]; c++) {
                if (!c->c_present || (c->c_iobp->status & XX_INTR) == 0)
                        continue;
                serviced = 1;
                xxintr(c);
        }
        return (serviced);
}
```

3.6. Some Common Service Functions

The kernel provides clusters of common service functions which device drivers can take advantage of. The common service functions fall into these major catagories:

Timeout Facilities
: are available when a device driver needs to know about real-time intervals.

Sleep and Wakeup Facilities
: suspend and resume execution of a process.

Raising and Lowering Interrupt Priorities
: Lock out devices by raising processor priority leve to stop the devices interrupting during critical operations (such as accessing shared data structures).

Main Bus Resource Management
: includes the routines `mbsetup` and `mbrelse` for scheduling the Main Bus resources.

Buffer Header Management
: Manages the in-memory disk buffer cache. We aren't dealing with disk drivers here so this needn't concern us.

There is also a kernel-specific version of the `printf` routine. The kernel `printf` is described later in this section.

3.6.1. Timeout Mechanisms

If a device needs to know about real-time intervals,

```
timeout(func, arg, interval)
```

is useful. `timeout` arranges that after *interval* clock-ticks (fiftieths of a second), the *func* is called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read a device if there is no response within a specified number of seconds (that is, there was a lost interrupt). Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general (you can't call `sleep` from within *func* for instance).

3.6.2. Sleep and Wakeup Mechanism

The other major help available to device handlers is the sleep-wakeup mechanism. The call

```
sleep(event, software_priority)
```

makes the process wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call returns when there is no process with higher *software_priority*.

The call

```
wakeup(event)
```

indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver (for a specific device if there is more than one minor device), which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened when they are awakened; they should check that the conditions which caused them to sleep no longer hold.

Software priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter `PZERO` and those at numerically larger priorities. The former cannot be interrupted by signals. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to **`sleep`** with larger priority may never return if the process is terminated by some signal in the meantime. In general, sleeps at less than PZERO should only be waiting for fast events like disk and tape I/O completion. Waiting for human activities like typing characters should be done at priorities greater than PZERO. Incidentally, it is a gross error to call **`sleep`** in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area 'u.' should be touched, let alone changed, by an interrupt routine.

3.6.3. Raising and Lowering Interrupt Priorities

At certain places in a device driver it is necessary to raise the hardware interrupt priority so that a section of critical code cannot be interrupted, for example, while adding or removing entries from a queue, or modifying a data structure common to both halves of a driver.

The `splx` function changes the interrupt priority to a specified level, and returns a value which is what the level was before it changed.

For configuration reasons, the routine:

```
pritospl(md->md_intpri)
          or
pritospl(mc->mc_intpri)
```

must be used to convert from the Main Bus hardware interrupt level to the CPU hardware priority level. Here is how you normally use the `pritospl` and `splx` functions in a hypothetical ***strategy*** routine:

```
hypo_strategy(bp)
        register struct buf *bp;
{
        register struct mb_ctlr *mc = hypoinfo[minor(bp->b_dev)];
        int s;

        s = splx(pritospl(md->md_intpri));
        while (bp->b_flags & B_BUSY)
                sleep((caddr_t)bp, PRIBIO);
        . . .
            here is some critical code section
        . . .
        (void)splx(s);       /* Set priority to what it was previously */
        . . .
}
```

3.6.4. Main Bus Resource Management Routines

The routine `mbsetup` is called when the device driver wants to start up a transfer to the device using Main Bus resource management.

At some later time, when the transfer is complete, the device driver calls the `mbrelse` routine to inform the Main Bus resource manager that the transfer is complete and the resources are no longer required.

3.7. Kernel printf Function

The kernel provides a `printf` function analogous to the `printf` function supplied with the standard I/O package for user programs. The kernel `printf` writes directly to the console however. The kernel `printf` function can be used to debug a driver.

There are three items of interest about the kernel `printf` function that you should be aware of:

1. When using the kernel `printf`, you should not use any floating-point conversions.
2 The kernel `printf` function raises the priority level and therefore may lock out interrupts while it is sending data to the console).
3. The kernel `printf` displays its messages directly on the console, that is, the boot device, unless specifically redirected by the TIOCCONS ioctl.

3.7.1. Macros to Manipulate Device Numbers

A device number (in this system) is a 16-bit number divided into two parts called the ***major*** device number and the ***minor*** device number. There are macros provided for the purpose of isolating the major and minor numbers from the whole device number. The macro

```
major(dev)
```

returns the major portion of the device number *dev*, and the macro

```
minor(dev)
```

returns the minor portion of the device number. Finally, given a major and a minor number *x* and *y*, the macro

```
makedev(x,y)
```

creates a device number from the two portions.

3.8. Overall Layout of a Device Driver

Here is a summary of the kit of parts that comprises a typical device driver. In any given driver, some routines may be missing. In a complex driver, all of these routines may well be present. A typical device driver consists of a number of major sections, containing the routines described below.

Auto Configuration
called by the kernel at system startup time to determine if the devices actually exist. This section contains the ***probe*** routine.

Opening and Closing the Device
The `open` routine is called for each instance of an `open` or ***create*** request against that file. The `close` routine is called when a `close` request is made against that file for the last time.

Reading and Writing from or to the Device
The `read` and `write` routines are called to get data from the device, or to send data to the device. The `read` and `write` routines may use the ***tty*** interfaces for devices such as terminals, or they might use a ***strategy*** routine to handle devices that transfer data in chunks. ***Strategy*** is most often used for DMA (Direct Memory Access) transfers, where the actual data buffer must be mapped in for the duration of the transfer.

Start Routine
The ***start*** routine is called to actually initiate the I/O operation. ***Start*** is needed in drivers that queue requests; it is called from the `read`, `write` or ***strategy*** routine to start the queue and is also called from the interrupt routine to start the next element on the queue.

Mmap Routine
The `mmap` routine is present in cases where it is required to map the device into user memory — a frame buffer for instance.

Polling Interrupt Routine
The polling interrupt routine of a device driver is called to service interrupts, possibly from the device for which this driver exists. However, there can be more than one device sharing the same interrupt level, and it is then also the task of the polling interrupt routine to determine if the interrupt is actually destined for this driver.

Ioctl Routine
The `ioctl` routine is called when the user process does an `ioctl` system call. A typical use is to change the baud-rate for a serial interface.

3.9. A Very Basic Skeleton Device Driver

At this stage, we quit discussing the I/O system and start writing a very simple device driver. This model will be one of the simplest drivers we can produce. There is a complete version of this driver in the attachments to this manual — the parts are presented piecemeal here with some discussion on their functions.

What we do here is to invent an interface board called a Skeleton controller. The Skeleton board is a very simple I/O mapped board, that is, it uses I/O ports in the Multibus I/O address space. The Skeleton board has a single-byte command/status register, and a single-byte data register. You can only write data to the outside world from the Skeleton board. This board is not a slow teletype style interface — you can provide vast blocks of data and the board sends it all out very fast. The Skeleton board interrupts when it is ready for a data transfer. The board comes up in the power on state with interrupts disabled and everything else in a 'normal' state.

The status register of the Skeleton interface is located at 0x600 in Multibus I/O space, and the data register at 0x601. The status register is both a read and a write register. The bit assignments are as shown in the tables below.

BIT	8	7	6	5	4	3	2	1
Read	Inter-rupt				Device Ready	Interface Ready		Interrupt Enabled

BIT	8	7	6	5	4	3	2	1
Write						Reset		Enable Interrupt

Here is a brief description of what the bits mean:

When ***reading*** from the status register

bit 8 is a 1 when the board is interrupting, 0 otherwise.

bit 4 is a 1 when the device that the board controls is ready for data transfers.

bit 3 is a 1 when the Skeleton board itself is ready for data transfers.

bit 1 is a 1 when interrupts are enabled, 0 when interrupts are disabled.

When ***writing*** to the status register

bit 3 resets the Skeleton board to its startup state — interrupts are disabled and the board should indicate that it is ready for data transfers.

bit 1 enables interrupts by writing a 1 to this bit, disables interrupts by writing a 0.

The header file for this interface is in ***skreg.h***. By convention, we put the register and control information for a given device (say ***xy***) in a file called ***xyreg.h***. The actual C code for the ***xy*** driver would by convention be placed in a file called ***xy.c***. The header file for the Skeleton board looks like this:

```
/*
 * Registers for Skeleton Multibus I/O Interface
 */
struct sk_reg {
        char    sk_data;        /* 01: Data Register */
        char    sk_csr;         /* 00: command(w) and status(r) */
};

/* sk_csr bits (read) */
#define SK_INTR 0x80            /* 1 if device is interrupting */
#define SK_DEVREADY     0x08    /* Device is Ready */
#define SK_INTREADY     0x04    /* Interface is Ready */
#define SK_INTENAB      0x01    /* Interrupts are enabled */

/* sk_csr bits (write) */
#define SK_RESET        0x04    /* reset the device and interface */
#define SK_ENABLE       0x01    /* Enable interrupts */
```

The complete device driver for the Skeleton board consists of the following parts:

`skprobe`
is the autoconfiguration routine called at system startup time to determine if the ***sk*** board is

actually in the system.

skopen* and *skclose
: routines for opening the device for each time the file corresponding to that device is opened, and for closing down after the last file has been closed.

skwrite
: routine which is called to send data to the device.

skstrategy
: routine which is called from the `write` routine via `physio` to initiate transfers of data.

`skstart`
: routine which is called for every byte to be transferred.

`skpoll`
: the polling interrupt routine which services interrupts and arranges to transfer the next byte of data to the device.

The subsections to follow describe these routines in more detail.

3.10. General Declarations in Driver

In addition to including a bunch of system header files, there are some data structures which the driver must define.

```
#include "sk.h"             /* header file generated by config (defines NSK) */

#define SKPRI    (PZERO-1)          /* software sleep priority for sk */

#define SKUNIT(dev)      (minor(dev))

struct  buf rskbuf[NSK];

int skprobe(), skpoll();

struct  mb_device *skdinfo[NSK];
struct mb_driver skdriver = { skprobe, 0, 0, 0, 0, skpoll,
                        sizeof(struct sk_reg), "sk", skdinfo, 0, 0, 0,
};

struct sk_device {
        struct buf *sk_bp;          /* current buf */
        int     sk_count;           /* number of bytes to send */
        char    *sk_cp; /* next byte to send */
        char    sb_busy;            /* true if device is busy */
} skdevice[NSK];
```

Here's a brief discussion on the declarations in the above example.

sk.h
: file is generated by the *config* program (discussed later). It contains the definition of NSK, the number of sk devices configured into the system.

SKPRI declaration declares the software priority level at which this device driver will sleep.

SKUNIT macro is a common way of obtaining the minor device number in a driver. Study just about any device driver and you will find a declaration like this — it is a stylized way of referring to the minor device number. One reason for this is that sometimes a driver will encode the bits of the minor device number to mean things other than just the device number, so using the **SKUNIT** convention is an easy way to make sure that is things change, the code will not be affected.

rskbuf array is necessary so that there will be *buf* structures to pass to the `physio` routine. `physio` will fill in certain fields before calling our *strategy* routine with the *buf* structure as the argument.

- Then there is a definition of the system dependent entry points into the device driver. In this driver, the only entry points we use are `skprobe` (probes the Multibus during system configuration time) and `skpoll` (interrupt routine).

skdinfo is the *device* structure for this driver. The system autoconfiguration routines fill in the apporpriate fields in this structure at startup time.

skdriver is a definition of the *driver* structure for this driver. An explanation of the fields in this structure and when they should be filled in appears earlier in this chapter.

Note that this data structure is the major linkage to the kernel. The structure *must* be called *driver-name*`driver` where *driver-name* is the name of your device driver — the *config*(8) program (described later) that builds your kernel from a description file assumes that all device driver structures have names of the form *driver-name*`driver`.

`sk_device`
is a definition of a structure that holds state information for each unit. This is information specific to this driver that needs to be remembered between subroutine calls.

3.11. Autoconfiguration Procedures

Part of a device driver's work is handling the automatic determination of the system configuration. When the Sun UNIX system boots up, it determines the peripheral configuration details by probing the memory space and I/O space located on the various buses of the machine.

Note that the `config`(8) program does the mapping of where things are in the system. You use the `[mach space]` clause to specify exactly where devices reside in the address space. If you don't specify where devices are, `config` uses these assumptions for backwards compatibility:

- Any address less than 64K is assumed to be Multibus I/O address space.
- Addresses less than 256K are assumed to be Multibus memory addresses.

3.12. Probe Routine

There should be a *probe* function in every driver. *Probe* is called at system initialization time with an address to be probed. *Probe* has two functions:

1. To determine if the device that this driver is written for exists at the specified address, and:

2. To make the kernel aware of how much of the system's resources to reserve for that device.

Under normal circumstances, addressing non-existent memory or I/O space on the Multibus or the VMEbus generates a bus error in the CPU. The kernel provides some functions to probe the address space, recover from possible bus errors, and return an indication as to whether the attempt to address a specific location generated a bus error.

Determining whether a device actually exists or not is assisted by the functions ***peek***, ***peekc***, ***poke***, and ***pokec***. These functions provide for accessing possibly non-existent addresses on the bus without generating bus errors that would terminate the process trying to access those addresses. ***Peek*** and ***poke*** read and write, respectively, 16-bit words (short's in the Sun system). ***Peekc*** and ***pokec*** read and write 8-bit characters. In general, you will use the character routines for probing single-byte I/O registers. See the section ***Summary of Functions*** for details on these routines.

Having determined whether the device exists in the system, the ***probe*** function returns either:

- the size (in bytes) of the device structure if it does exist. The kernel uses the value returned from ***probe*** to reserve memory resources for that device. For I/O mapped devices, ***probe*** returns the amount of I/O space that the device registers consume. For memory-mapped devices, ***probe*** returns the amount of memory that the device consumes.
- a value of 0 (zero) if the device does not exist.

Now we can write `skprobe`:

```
/*  ARGSUSED  */
skprobe(reg, unit)
        caddr_t reg;
        int unit;
{
        register struct sk_reg *sk_reg;
        register int c;

        sk_reg = (struct sk_reg *)reg;
        c = peekc((char *)&sk_reg->sk_csr);
        if (c == -1)
                return (0);

        return (sizeof (struct sk_reg));
}
```

The ***reg*** argument is the purported address of the device. The ***unit*** argument is usually not needed.

If the ***probe*** routine determines that the device actually exists and it returns the amount of resources that the deevice uses, the system startup routines set the ***md_alive*** field in the device structure to non-zero. The ***md_alive*** field is then used subsequently by other driver functions to check that the device was probed successfully at startup time.

3.13. Open and Close Routines

During the processing of an `open` or ***creat*** call for a special file, the system always calls the device's `open` routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the `close` routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device driver to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

The *Open* routine for the **sk** driver is simple. ***Skopen*** is called with two arguments, namely, the device which must be opened, and a flag indicating whether the device should be opened for reading, writing, or both. The first task is to check whether the device number to be opened actually exists — ***skopen*** returns an error indication if not. The second check is whether the open is for writing. Since **sk** is a 'write only' device, it is an error to open it for reading only. If all the checks succeed, ***skopen*** enables interrupts from the device, and then returns a zero (0) as an indication of success. Here is the code for the ***skopen*** routine:

```
skopen(dev, flags)
        dev_t dev;
        int flags;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;

        if (SKUNIT(dev) >= NSK ||
            (md = skdinfo[SKUNIT(dev)]) == 0 || md->md_alive == 0)
                return (ENXIO);

        if (flags & FREAD)
                return (ENODEV);

                                        /*  enable interrupts  */
        sk_reg = (struct sk_reg *)md->md_addr;
        sk_reg->sk_csr = SK_ENABLE;

        return (0);
}
```

The first if statement checks if the device actually exists. Note the use of the SKUNIT macro to obtain the minor device number — we discussed this earlier on.

The `close` routine for the **sk** driver is very simple — all it does is disable interrupts:

```
/*ARGSUSED*/
skclose(dev, flags)
        dev_t dev;
        int flags;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        md = skdinfo[SKUNIT(dev)];

                                        /*  disable interrupts  */
        sk_reg = (struct sk_reg *)md->md_addr;
        sk_reg->sk_csr |= ~SK_ENABLE;
}
```

`skclose` could in fact be more complicated than this. Some of the actions that could take place in a `close` routine might be to deallocate any resources that were allocated for this device driver, and possibly to `sleep` on completion of I/O transfers for that device.

3.14. Read and Write Routines

When a `read` or `write` takes place, the user's arguments and the ***file*** table entry are used to set up the variables `iovec.iov_base`, `iovec.iov_len`, and `uio.uio_offset` which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate `read` or `write` routine is called — this `read` or `write` routine is responsible for transferring data and updating the count and current location appropriately as discussed below.

The `write` routine for the skeleton driver is very simple. ***write*** simply calls the ***strategy*** routine through the `physio` system routine. `physio` ensures that the user's memory space is available to the driver for the duration of the data transfer. `physio` also takes care of updating the count and current location as appropriate. The `write` routine looks like this:

```
skwrite(dev, uio)
        dev_t dev;
        struct uio *uio;            see below for some notes on this
{

        if (SKUNIT(dev) >= NSK)
                return (ENXIO);
        return (physio(skstrategy, &rskbuf[SKUNIT(dev)], dev, B_WRITE,
                      skminphys, uio));
}
```

The ***skminphys*** routine is called by `physio` to determine the largest reasonable blocksize to transfer at once. If the user has requested more bytes than this, `physio` will call ***skstrategy*** repeatedly, requesting no more than this blocksize each time. The case where this is important is when DVMA transfers are done. (DVMA is covered in more detail below.) The reasoning is that only a finite amount of address space is available for DMVA transfers and it is not reasonable for any device to tie up too much of it. A disk or a tape might reasonably ask for as much as 64 Kbytes; slow devices like printers should only ask for one to four Kbytes since they will tie up the resource for a relatively long time.

Here is the ***skminphys*** routine.

```
skminphys(bp)
        struct buf *bp;
{

        if (bp->b_bcount > MAX_SK_BSIZE)
                bp->b_count = MAX_SK_BSIZE;
}
```

Note that if you don't supply you own ***minphys*** routine, you place the name of the system supplied ***minphys*** routine, whose name is ***minphys***, as the argument to the ***strategy*** routine at that place, and the system supplied ***minphys*** routine gets used instead.

3.14.1. Some Notes About the UIO Structure

When the system is reading and writing data from or to a device, the uio structure is used extensively. The uio structure is a general structure to allow for what is called gather-write and scatter-read. That is, when writing to a device, the blocks of data to be written don't have to contiguous in the user's memory but can be in physically discontiguous areas. Similarly, when reading from a device into memory, the data comes off the device in a continuous stream but can go into physically discontiguous reas of the user's memory. Each discontiguous area of memory is described by a structure called an iovec (I/O vector). Each iovec contains a pointer to the data area to be transferred, and a count of the number of bytes in that area. The uio structure describes the complete data transfer. uio contains a pointer to an array of these iovec structures. Thus when you want to write a number of physically discontiguous blocks of memory to a device, you can set up an array of iovec structures, and place a pointer to the start of the array in the uio structure. In the trivial case, there is generally just one block of data to be transferred, and so the uio structure is fairly simple.

3.15. Skeleton Strategy Routine

The *strategy* routine is called by physio after the user buffer has been locked into memory. The *strategy* routine must check that the device is ready and initiate the data transfer. *Strategy* then waits for the the completion of the data transfer, which will be signaled by the interrupt routine.

```
skstrategy(bp)
        register struct buf *bp;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;
        int s;

        md = skdinfo[SKUNIT(bp->b_dev)]
        sk_reg = (struct sk_reg *)md->md_addr;
        sk = &sk_device[SKUNIT(dev)];
        s = splx(pritospl(md->md_intpri));
        while (sk->sk_busy)
                sleep((caddr_t) sk, SKPRI);
        sk->sk_busy = 1;
        sk->sk_bp = bp;
        sk->sk_cp = bp->b_un.b_addr;
        sk->sk_count = bp->b_bcount;
        skstart(sk, (struct sk_reg *)md->md_addr,);
        sk->sk_busy = 0;
        wakeup((caddr_t) sk);
        (void)splx(s);
}
```

3.16. Skeleton Start Routine — Initiate Data Transfers

The *start* routine is responsible for getting the actual data bytes out to the device itself. *Start* is called once by *strategy* to get the very first byte out to the interface. After that, it is assumed that the device will interrupt every time it is ready for a new data byte, and so *start* is thereafter called from the interrupt routine. Here is the *start* routine:

```
skstart(sk, sk_reg)
        struct sk_device *sk;
        struct sk_reg *sk_reg;
{

        sk_reg->sk_data = *sk->sk_cp++;
        sk->sk_count--;
        sk_reg->sk_csr = SK_ENABLE;
}
```

This routine will work, but there is a lot of overhead in taking an interrupt from the device on every character. Since we know that the device can take characters very quickly. it would be more efficient to try to give characters quickly. What we will do is to check after each character and give another one if the device is ready. Here is the new, more efficient `skstart` routine.

```
skstart(sk, sk_reg)
        struct sk_device *sk;
        struct sk_reg *sk_reg;
{

        do {
                sk_reg->sk_data = *sk->sk_cp++;
                sk->sk_count--;
        } while (sk->sk_count && sk_reg->sk_csr & SK_DEVREADY);
        if (sk->sk_count)       /* more characters to go */
                sk_reg->sk_csr = SK_ENABLE;
        else {
                sk_reg->sk_csr = 0;     /* disable interrupts */
                iodone(sk->sk_bp);
        }
}
```

We give characters to the device as long as there are more characters and the device is ready to receive them. If we run out of characters, we disable interrupts to keep the device from bothering us and call `iodone` to mark the buffer as done.

It may be that the device is not quite quick enough to take a character and raise the SK_DEVREADY bit in the time we can decrement and test the counter. If so, it would be very worthwhile to busy wait for a short time. The reasoning is that while busy waiting is a waste, servicing an interrupt costs lots more CPU time, and if busy waiting works fairly often it is a big win. There is a macro DELAY which takes an integer argument which is approximately the number of microseconds to delay, so we could add

```
DELAY(10);
```

just before the `while`. Clearly this is an area where experimentation with the real device is called for.

3.17. Interrupt Routines

Each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. After the interrupt has been processed, a return from the interrupt handler returns from the interrupt itself.

The address of the polling interrupt routine for a particular device driver is contained in the per-driver (that is, `mb_driver`) data structure for that device driver. The address of the polling interrupt routine is filled in statically at the time the data structure is declared and initialized.

Since there may be many devices sharing a common interrupt level, it is the specific driver's responsibility to determine if the interrupt is intended for it or not. If the interrupt ***is*** for this driver, the driver must service the interrupt and return a non-zero value to indicate that the interrupt has been serviced. If the interrupt is ***not*** for this device driver, the polling interrupt routine must return a zero value.

It is expected that the device actually indicates when it is interrupting. If there are any more bytes to transfer, the interrupt routine calls the ***start*** routine to transfer the next byte. If there are no more bytes to transfer, the interrupt routine disables the interrupt (so that the device won't keep interrupting when there is nothing to do), and finishes up by calling **`iodone`**. Here are the interrupt routines for this device:

```
skpoll()
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;
        int serviced;

        serviced = 0;
        for (i = 0; i < NSK; i++) {
                md = &skdinfo[i];
                sk_reg = (struct sk_reg *)md->md_addr;
                if (sk_reg->sk_csr & SK_INTR) {
                        serviced = 1;
                        skintr(i);
                }
        }
        return (serviced);
}
```

```
skintr(i)
        int     i;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;
        int serviced;

        md = &skdinfo[i];
        sk = &sk_device[i];
        sk_reg = (struct sk_reg *)md->md_addr;
        if (sk->sk_count == 0) {
                sk_reg->sk_csr = 0;     /* Clear interrupt  */
                        iodone(sk->sk_bp);
                } else
                        skstart(sk, sk_reg);
}
```

3.18. Ioctl Routine

The `ioctl` routine is used to perform any tasks that can't be done by the regular `open`, `close`, `read`, or `write` routines. Typical applications are: 'what is the status of this device', or 'tell me the partitions on disk **xy1**'. This device does not need any special functions so we don't have an `ioctl` routine.

3.19. Devices That Do DMA

Devices that are capable of doing DMA are treated a little differently than the skeleton device we have been working with so far. Let us assume that we have a new version of the skeleton board; call it the Skeleton II. It can do DMA transfers and we want to use this feature since it is much more efficient. First we must describe DMA on the Sun-2.

3.20. Multibus or VMEbus DVMA

On the Sun-2, the processor board is always listening to the Multibus or VMEbus for memory references. When a request to read or write Multibus or VMEbus memory between addresses 0 and 256K comes up, the DVMA hardware takes the address, adds 0xF00000 to it, and goes through the kernel memory map to find the location in processor memory that will be used. Thus if you wish to do DMA over the Multibus or VMEbus, you must make the appropriate entries in the kernel memory map. As you might expect, there are subroutines to help with this chore. `mbsetup` sets up the map and `mbrelse` releases the map.

3.21. Changes to the Driver

The changes to the driver are surprisingly simple. FIrst we must extend the `sk_reg` structure which defines the device registers. We assume that the Skeleton II supports the following structure.

```
struct sk_reg {
        char    sk_data;        /* 01: Data Register */
        char    sk_csr;         /* 00: command(w) and status(r) */
        short   sk_count;       /* bytes to be transferred */
        caddr_t sk_addr;        /* DMA address */
};
```

Next we assume another bit in the csr.

```
#define SK_DMA  0x10    /* Do DMA transfer */
```

And we must add another element in the `sk_device` structure for use by *msetup* and `mbdone`.

```
        int     sk_mbinfo;
```

Now we change the *skstrategy* routine to use the DMA feature.

```
skstrategy(bp)
        register struct buf *bp;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct struct sk_device *sk;
        int s;

        md = skdinfo[SKUNIT(bp->b_dev)]
        sk_reg = (struct sk_reg *)md->md_addr;
        sk = &sk_device[SKUNIT(dev)];
        s = splx(pritospl(md->md_intpri));
        while (sk->sk_busy)
                sleep((caddr_t) sk, SKPRI);
        sk->sk_busy = 1;
        sk->sk_bp = bp;
        /* this is the part that is changed */
        sk->sk_mbinfo = mbsetup(md->md_hd, bp, 0);
        sk_reg->sk_count = bp->b_count;
        sk_reg->sk_addr = MBI_ADDR(sc->sc_mbinfo);
        sk_reg->sk_csr = SK_ENABLE | SK_DMA;
        /* end of changes */
        iowait(bp);
        sk->sk_busy = 0;
        wakeup((caddr_t) sk);
        splx(s);
}
```

The need for the `skstart` routine is completely gone and thus we will delete it. All the I/O now is started by *skstrategy* and continues until `skpoll` is called. Thus we can delete the `sk_cp` and *sc_count* variables from the `sk_device` structure.

`skintr` is also simplified. There is no longer any need to check the count since all the data goes out through DMA. Therefore `iodone` will always be called. Also, we need to free up the Main Bus resources, so we will call the `mbrelse` routine. Here is the new `skpoll` and `skintr` routines:

```
skpoll()
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;
        int serviced;

        serviced = 0;
        for (i = 0; i < NSK; i++) {
                md = &skdinfo[i];
                sk_reg = (struct sk_reg *)md->md_addr;
                if (sk_reg->sk_csr & SK_INTR) {
                        serviced = 1;
                        skintr(i);
                }
        }
        return (serviced);
}

skintr(i)
        int     i;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;
        int serviced;

        md = &skdinfo[i];
        sk = &sk_device[i];
        sk_reg = (struct sk_reg *)md->md_addr;
                                                  /*  This is the part that changed */
        sk_reg->sk_csr = 0;       /*  Clear interrupt  */
        mbrelse(md->md_hd, &sk->sk_mbinfo);
        iodone(sk->sk_bp);
}
```

3.22. Errors

We have been pretty casual about errors up till now. Most devices have at least an error bit in the csr, and usually more detailed error information is available. Also, we should check whether the DMA count is exhausted.

Detection and treatment of errors varies greatly from device to device and is not very generalisable, so it wouldn't add much to this tutorial to show some elaborate error checking. Nonetheless, error checking is important because if you don't check for errors and they do happen your users will be very unhappy.

You should read the Product Specification manual for your device very carefully to determine what error indications can be given and what you should do when they do come up. At the very least, check for errors and if you can't figure out what to do about them, use the kernel `printf` function to display a message to the console just to let the world know that everything is not perfectly OK.

3.23. Memory Mapped Devices

Devices such as frame buffers are frequently accessed by mapping the buffer into the user address space and allowing the user to update them at will. The user accomplishes this through a `mmap`(2) system call. This call is translated by the kernel into a call to the driver's `mmap` routine. The call has three parameters, ***dev, off*** and ***prot***. ***Dev*** is of course the device major and minor number, ***off*** is the offset into the frame buffer from the user's `mmap` system call, and ***prot*** is a flag indicating whether write protection applies to the page(s). The constants PROT_READ, PROT_WRITE and PROT_EXEC are defined in the header file ***mman.h***. Each constant is a bit turned on to indicate that the appropriate access is allowed.

Here is the `mmap` routine from the Sun Color Graphics driver.

```
cgmmap(dev, off, prot)
        dev_t dev;
        off_t off;
        int prot;
{
        register caddr_t addr;
        register int page, uc;

        addr = cginfo[minor(dev)]->md_addr;
        if (off >= CGSIZE)
                return (-1);
        page = getkpgmap(addr + off) & PG_PFNUM;
        return (page);
}
```

The PG_PFNUM constant gets rid of extraneous bits that `getkpgmap` returns and just leaves the page number plus page type, which is what we have to return.

The routine first gets the address of the frame buffer from the Main Bus device structure. Remember that this is generated by `config` based on the user's input as to where devices are configured. Next the offset is checked to be sure the user isn't mapping beyond the end of the frame buffer. Next comes a call to `getkpgmap` to do the actual mapping. The page number returned by `getkpgmap` is then returned by `cgmap`. In this case, `prot` is not checked since the driver permits open to succeed only if the user is opening for both read and write, thus all access are permitted.

Chapter 4

Configuring the System to Add Skeleton Driver

Now we've written the Skeleton driver, we'll go through the steps required to add it to the system. A detailed description of how to configure and build a kernel is in the document *Building UNIX Systems With Config* in the *System Manager's Manual.* Here we just cover what is needed to add a new driver.

New device drivers require entries in */sys/sun/conf.c* and in */sys/conf/files.sun*. They are included by mentioning the device name in the configuration file.

The examples to follow assume that you are adding a driver for the Skeleton board (sk) to the system. The new system will be called *SKELETON*. Here is a representative section from *sun/conf.c*:

```
#include "sk.h"
#if NSK > 0
int     skopen(), skclose(), skread(), skwrite(), skmmap();
#else
#define skopen  nodev
#define skclose nodev
#define skread  nodev
#define skwrite nodev
#define skmap   nodev
#endif
        .
        .
        .
    {
skopen,  skclose, skread, skwrite,        /* 40 */
nodev,   nodev,   nodev,  0,
seltrue, skmmap,
    },
```

If NSK is greater than 0, this will add the driver routines into the `cdevsw` table so the kernel knows where they are. (NSK is set by the `config` program based on the kernel configuration file discussed below.) The entres added are, in order, the `open`, `close`, `read`, `write`, `ioctl`, `stop` and `reset` routines, a *tty* structure address and finally the `select` and `mmap` routines. We do not have an `ioctl` routine so this entry calls `nodev` which is a special routine that always returns an error. Since we are not a tty we do not have a `stop` routine which would be used for flow control, nor do we have a `tty` structure. The `reset` routine is not used so all devices use `nodev` for this one. The `select` routine is called when a user process does a `select`(2) system call; it returns true if the device can be immediately selected. Since our sk device is write only and fast, it is always selectable so we use the default `seltrue` routine which always returns true.

Here is the line you must add to *files.sun*:

```
        sundev/sk.c      optional sk device-driver
```

This says that the file *sundev/sk.c* contains the source code for the optional *sk* device and that it is a device driver.

Now, you can go through the process of building the system just as described in the chapter on configuration: Choose a name for your configuration of the system — in our case it will be called *SKELETON*. Then create the configuration file and directory:

```
        gaia#  cp GENERIC SKELETON
        gaia#  mkdir ../SKELETON
```

Edit *SKELETON* to reflect your system — you must add a description of the device to the *SKELETON* file:

```
        device  sk0 at mb0 csr all virt 0xeb0600 priority 2 [vector skintr 200]
```

This entry says we have an sk device (the first device is always number 0) on the Multibus, the control/status register (device register) is at Multibus address 0x600 (this is passed to out ***probe*** routine at boot time), this device will interrupt at level 2, and that if we are on a system that supports vectored interrupts, vector number 200 is set up to call the `skintr` routine.

Note the `all virt` keys and the address given as `0xeb0600` instead of simply as `0x0600`: this means that the system will handle this device both on the Multibus, or in systems with a VMEbus with a Multibus adaptor, since the monitor maps the equivalent of the Multibus I/O space (the 16-bit VMEbus space that the adapter listens to) to `0xeb0600`.

Then you can run */etc/config* to make the configuration files for the new device driver:

```
        gaia#  /etc/config SKELETON
```

/etc/config uses *SKELETON*, *files*, and *files.sun* as input, and generates a number of files in the *../SKELETON* directory.

Now you can change directory to the new configuration directory, *../SKELETON* in this case, and make the new system:

```
        gaia#  cd ../SKELETON
        gaia#  make depend
        gaia#  make
```

The **make depend** command creates the dependency tree for any new C source files you might have created during the process of adding new drivers or whatever to the system.

Now you must add a new device entry to the */dev* directory. The connections between the UNIX operating system kernel and the device driver is established through the entries in the */dev* directory. Using the example above as our model, we want to install the device for the Skeleton driver.

Making new device entries is done via a shell script called *MAKEDEV* in the */dev* directory. It is worth while looking inside *MAKEDEV* to find out the kinds of things that go on in there. The lines of shell script below reflect what you would add to *MAKEDEV* for the new Skeleton device. First, there are lines of commentary at the start of the *MAKEDEV* file:

```
#! /bin/sh
#         MAKEDEV 4.3     83/03/31
# Graphics
#         sk*       Skeleton Board
```

Then there is the actual shell 'code' which makes the device entries:

```
skeleton|sk|sk0)
          /etc/mknod sk0 c 40 0              ; chmod 666 sk0
          ;;
```

This makes the special inode **/dev/sk0** as a character special device with major device number 40 and minor device number 0, and then sets the mode of the file so that anyone can read or write the device.

Having added the new device entry, you can install the new system and try it out.

```
gaia#  cp vmunix /vmunix+
gaia#  /etc/halt
          The system goes through the halt sequence, then
       the monitor displays its prompt, at which point you
       can boot the system in single-user state
>   b vmunix+ -s
          The system boots up in single user state and
       then you can try things out
gaia#
```

If the system appears to work, save the old kernel under a different name and install the new one in /vmunix:

```
gaia#  cd /
gaia#  mv vmunix ovmunix
gaia#  mv vmunix+ vmunix
gaia#
```

Make sure that the new version of the kernel is actually called *vmunix* — because programs such as *ps* and *netstat* use that exact name to look for things, and if the running version of the kernel is called something other than *vmunix* the results from such programs will be wrong.

Appendix A

Summary of Functions

A.1. Standard Error Numbers

The system has a collection of standard error numbers that a driver can return to its callers. These numbers are described in detail in *intro*(2), the introductory pages of the *System Interface Manual*. A complete listing of the error numbers appears in <*sys/errno.h*>.

A.2. Device Driver Routines

A.2.1. Autoconfiguration Routines

A.2.1.1. `probe` — *Determine if Hardware is There*

```
probe(reg, unit)
     caddr_t reg;
     int     unit;
```

`probe` determines whether the device at address *reg* actually exists and is the correct device for this driver. If the device exists and is correct, `probe` returns

```
return (sizeof (struct device));
```

If the device does not exist, or is the wrong device for this driver, `probe` returns 0 (zero).

A.2.2. Open and Close Routines

A.2.2.1. open — *Open a Device for Data Transfers*

```
open(dev, flags)
    dev_t dev;
    int flags;
```

open checks that the minor device number passed in the *dev* argument is in range. The integer argument *flags* contains bits telling whether the open is for reading, writing, or both. The constants *FREAD* and *FWRITE* are available to be and'ed with *flags*. open returns:

```
return (ENXIO);
```

(meaning a non-existent device) if the minor device number is out of range. Then open attempts to initialize the device, and if there are any errors, open returns:

```
return (EIO);
```

to mean an I/O error. If the open is successful, open returns 0 (zero).

A.2.2.2. close — *Close a Device*

```
close(dev, flags)
    dev_t dev;
    int flags;
```

Close does whatever it has to do to indicate that data transfers cannot be made on this device until it has been reopened. *Flags* is the same as for open.

A.2.3. Read, Write, and Strategy Routines

A.2.3.1. read — *Read Data from Device*

```
read(dev, uio)
    dev_t dev;
    struct uio *uio;
```

read is the high-level routine called to perform data transfers from the device. read must check that the minor device number passed to it is in range. If the minor device number is out of range, read returns:

```
return (ENXIO);
```

meaning that the device is non-existent. Subsequent actions of read differ depending on whether the device is a character-at-a-time device such as a teletype, or is a block transfer device.

For the block-transfer devices, read simply calls on the *strategy* function via *physio*:

```
return (physio(strategy, &rbuf[minor(dev)], dev, B_READ, minphys, uio));
```

A.2.3.2. `write` — *Write Data to Device*

```
write(dev, uio)
    dev_t dev;
    struct uio *uio;
```

`write` is the high-level routine called to perform data transfers to the device. `write` must check that the minor device number passed to it is in range. If the minor device number is out of range, `write` returns:

```
if (VPUNIT(dev) >= NVP)
    return (ENXIO);
```

Subsequent actions of `write` differ depending on whether the device is a character-at-a-time device such as a teletype, or is a block transfer device.

For the block-transfer devices, `write` simply calls on the ***strategy*** function via ***physio***:

```
return (physio(strategy, &rbuf[minor(dev)], dev, B_WRITE, minphys, uio));
```

A.2.3.3. `strategy` *Routine*

```
strategy(bp)
    register struct buf *bp;
```

Strategy is the high level routine responsible for getting the data to the actual device. For DMA devices, ***strategy*** calls on ***mbgo*** to schedule the Main Bus resources. ***strategy*** does not return any value.

A.2.3.4. `minphys` — *Determine Maximum Block Size*

```
int     block = some 'reasonable' block size for transfers
                      must be a multiple of 1024 bytes

unsigned minphys(bp)
    register struct buf *bp;
```

Minphys determines a 'reasonable' block size for transfers, so as to avoid tying up too many resources. ***Minphys*** is passed as an argument to ***physio***. In the absence of a ***minphys*** functions supplied by the device driver itself, a system supplied version of ***minphys*** is used instead. ***Minphys*** shoulld perform the calculation:

```
if (bp->b_bcount > block)
    bp->b_bcount = block;
```

A.2.4. `ioctl` — *Special Interface Function*

```
ioctl(dev, cmd, data, flag)
    dev_t dev;
    int cmd;
    caddr_t data;
    int flag;
```

`ioctl` differs for every device and covers the functions that aren't done by `read` and `write`. `ioctl` does whatever it has to do, then returns 0 (zero) if there were no errors, and returns:

```
return (ENOTTY);
```

in the case that the command requested did not apply to this device. Note that ENOTTY gives rise to the error message 'Not a typewriter', which may be misleading.

A.2.5. Low Level Routines

Routines in this area are low level and can potentially be called from the interrupt side of the driver. `sleep` calls may never be made from the routines described here.

A.2.5.1. `intr` — *Handle Vectored Interrupts*

```
intr(unit)
    int  unit;
```

`intr` is responsible for fielding vectored interrupts from the device. `unit` is the unit number of the device that interrupted.

A.2.5.2. `poll` — *Handle Polling Interrupts*

```
poll()
```

`poll` is responsible for fielding non-vectored interrupts from the device. In situations where more than one device share the same interrupt level, `poll` must determine if the interrupt was actually destined for this driver or not. `poll` returns 0 (zero) to indicate that the interrupt was not serviced by this driver, and non-zero to indicate that the interrupt was serviced. It is a gross error for `poll` to say that it serviced an interrupt when it really did not.

If a device driver handles both vectored interrupts and polling interrupts, the `poll` routine typically calls the `intr` routine with the proper arguments, normally the unit number of the device that interrupted.

A.3. Common Service Routines

A.3.1. `sleep` — *Sleep on an Event*

```
sleep(address, priority)
    caddr_t address;
    int priority;
```

`sleep` is called to put a process to sleep. The ***address*** argument is typically the address of a location in memory. ***Priority*** is the software priority the process will have after it is woken up. The process which has been put to sleep can be woken up again by issuing a `wakeup` call with the same ***address***. `sleep` should ***never*** be called from the low level side of a driver.

A.3.2. `wakeup` — *Wake Up a Process Sleeping on an Event*

```
wakeup(address)
    caddr_t address;
```

`wakeup` is called when a process waiting on an event must be awakened. ***Address*** is typically the address of a location in memory. `wakeup` is typically called from the low level side of a driver when (for instance) all data has been transferred to or from the user's buffer and the process waiting for the transfer to complete must be awakened.

A.3.3. `mbsetup` — *Set Up to Use Main Bus Resources*

```
mbsetup(md_hd, bp, flag)
    struct mb_hd *mb_hd;
    struct buf  *bp;
    int flag;
```

Mbsetup is called to set up the memory map for a Main Bus DVMA transfer. ***flag*** is *MB_CANTWAIT* if the caller desires not to wait for map resources if none are available. Normally this will be zero which means the driver will wait. ***Mbsetup*** returns an integer which must be saved for the call to ***mbrelse***.

A.3.4. `mbrelse` — *Free Main Bus Resources*

```
mbrelse(md_hd, mbinfop)
    struct mb_hd *mb_hd;
    int *mbinfop;
```

Mbrelse releases the Main Bus DVMA resources allocated by ***mbsetup***. Note that the second parameter is a ***pointer*** to the integer returned by ***mbsetup***.

A.3.5. `physio` — *Lock in User's Buffer Area*

```
physio(strat, buf, dev, flag, minphys, uio)
    void  (*strat) ();
    struct  buf  *buf;
    dev_t  dev;
    int  flag;
    void  (*minphys) ();
    struct  uio  *uio;
```

A.3.6. `iowait` — *Wait for I/O to Complete*

```
iowait(bp)
    struct  buf  *bp;
```

`iowait` waits on the buffer header addressed by ***bp*** for the DONE flag to be set. `iowait` actually does a `sleep` on the buffer header.

A.3.7. `iodone` — *Indicate I/O Complete*

```
iodone(bp)
    struct  buf  *bp;
```

`iodone` is called to indicate that I/O associated with the buffer header ***bp*** is complete. `iodone` sets the DONE flag in the buffer header, then does a `wakeup` call with the buffer pointer as argument.

A.3.8. `pritospl` — *Convert Priority Level*

```
pritospl(value)
    int  value;
```

`pritospl` is a macro that converts the hardware priority level given by ***value***, which is a Main Bus priority level, to a CPU hardware priority level used by `splx`. `pritospl` is used to parameterize the setting of priority levels.

A.3.9. `spl`*n* `()` — *Set Specific Priority Level*

The ***spln***( ) functions are available for setting the priority level to ***n***, where ***n*** ranges from 0 to 7. These routines should probably never be used in any device driver.

A.3.10. `splx` — *Reset Priority Level*

```
splx(s)
    int  s;
```

`splx` called with an argument *s* sets the priority level to *s*, which was returned from a previous `spl`*n*`()`, `pritospl()`, or `splx()` call. `splx` is typically used to restore the priority level to a previously remembered level. `splx()` returns the previous level.

A.3.11. `uiomove` — *move data to or from the uio structure*

```
uiomove(cp, n, rw, uio)
    register caddr_t  cp;
    register int n;
    enum uio_rw  rw;
    register struct *uio;
```

Device drivers use ***uiomove*** to move a specified number of bytes between an area defined by a ***uio*** structure (normally passed to the driver when it is called) and an area in the kernel's address space (where it can be used by the driver). ***Uiomove*** moves ***n*** bytes from or to the ***iovec*** pointed to by the ***uio*** structure out of or into the area specified by *cp*. The read/write flags (which specify the direction of the data transfer) are defined in <*uio.h*>. ***Uiomove*** replaces the older ***copyin*** and ***copyout*** routines which are no longer supported. ***Uiomove*** can also be used to copy kernel ***uio*** structures — it checks ***uio->uio_segflag***.

A.3.12. `ureadc` *and* `uwritec` — *transfer bytes to or from a uio structure*

```
ureadc(c, uio)
    int c;
    register struct *uio;
```

Ureadc transfers a character represented by *c* in the definition into the ***iovec*** pointed at by the ***uio*** structure (normally passed to the driver when it is called). *Ureadc* is normally used when 'reading' a character in from a device.

```
uwritec(uio)
    register struct *uio;
```

Uwritec returns the next character in the ***iovec*** pointed at by the ***uio*** structure (normally passed to the driver when it is called), or returns −1 on error. *Uwritec* is normally used when 'writing' a character out to a device.

Note that 'read' and 'write' are slightly confusing in the above contexts, since ***ureadc*** actually obtains a character from somewhere and places the character ***into*** the ***iovec*** pointed to by the ***uio*** structure, whereas ***uwritec*** obtains a character from the ***iovec*** and 'writes' the character somewhere.

Ureadc and ***uwritec*** replace the routines ***cpass*** and ***passc***, which are no longer supported.

A.3.13. `peek, peekc` — *Check Whether an Address Exists and Read*

```
peek(address)
    short *address;

peekc(address)
    char *address;
```

peek and ***peekc*** are called with an address from which you want to read. Both ***peek*** and ***peekc*** return −1 if the addressed location doesn't exist, otherwise they return the value which was fetched from that location.

A.3.14. `poke, pokec` — *Check Whether an Address Exists and Write*

```
poke(address, value)
    short *address;
    short value;

pokec(address, value)
    char *address;
    char value;
```

poke and ***pokec*** are called with an ***address*** you want to store into, and ***value*** is the value you want to store there. Both ***poke*** and ***pokec*** return 1 if the addressed location doesn't exist, and 0 if the addressed location does exist.

A.3.15. `geteblk` — *Allocate Dynamic Buffer*

```
struct  buf  *geteblk(size)
    int *size;
```

geteblk allocates a buffer dynamically. The ***size*** of the block is limited to a maximum of 8K bytes, and must be a multiple of 512 bytes.

A.3.16. `brelse` — *Free Dynamic Buffer*

```
brelse(bp)
struct  buf  bp;
```

brelse frees a buffer previously allocated by ***geteblk***.

A.3.17. `swab` — *Swap Bytes*

```
swab(from, to, nbytes)
caddr_t  from;
caddr_t  to;
int  nbytes;
```

swab swaps bytes within words. ***nbytes*** is the number of bytes to swap, and is rounded up to a multiple of two. The ***from*** and ***to*** areas can overlap each other since the bytes are swapped one at a time.

Appendix B

Sample Drivers

The C code listings supplied here are sample drivers for devices that the Sun system supports. There are three drivers listed here:

CGONE
: is a device driver for the Sun-1 color graphics board. It is one of the simplest drivers around, being memory mapped.

SKY
: is a programmed I/O driver for the SKY floating-point board, with both polling interrupts and vectored interrupts. However, the interrupt routines don't do a whole lot.

VP is a fairly good example of a DMA device driver.

```
/*      @(#)cgreg.h 1.3 84/12/22 SMI    */

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Register definitions for Sun Color Board
 */
#define CGSIZE  (16*1024)       /* 16K of address space */

# define GR_bd_sel   CGXBase    /* Select Color Board */

# define GR_x_select  0x0800    /* Access a column in the frame buffer */
# define GR_y_select  0x0000    /* Access a row in the frame buffer */
# define GR_y_fudge   0x0200    /* Bit 9 not used at all */
# define GR_update    0x2000    /* Update frame buffer if this bit set */
# define GR_x_rhaddr  0x1b80    /* Location to read X address bits A9-A8.
                                   Data put into D1-D0. */
# define GR_x_rladdr  0x1b00    /* Location to read X address bits A7-A0.
                                   Data put into D7-D0. */
# define GR_y_rhaddr  0x1bc0    /* Location to read Y address bits A9-A8. */
# define GR_y_rladdr  0x1b40    /* Location to read Y address bits A7-A0. */


# define GR_set0      0x0000    /* Address Register pair 0. */
# define GR_set1      0x0400    /* Address Register pair 1. */


# define GR_red_cmap  0x1000    /* Address to select Red Color Map */
# define GR_grn_cmap  0x1100    /* Addr for Green Color Map */
# define GR_blu_cmap  0x1200    /* Addr for Blue Color Map */

# define GR_sr_select 0x1800    /* Addr to select status register */
# define GR_cr_select 0x1900    /* Addr to select mask (color) register */
# define GR_fr_select 0x1a00    /* Addr to select function register */


/* The following are pointers to the mask(color), status, and function regs. */

# define GR_creg       (u_char *)(GR_bd_sel + GR_cr_select)
# define GR_mask       (u_char *)(GR_bd_sel + GR_cr_select)
# define GR_sreg       (u_char *)(GR_bd_sel + GR_sr_select)
# define GR_freg       (u_char *)(GR_bd_sel + GR_fr_select)


/* These assignments are for bits in the Status Register */
# define GRW0_cplane 0x00      /* Select CMap Plane number zero for R/W */
# define GRW1_cplane 0x01      /* Select CMap Plane number one for R/W */
# define GRW2_cplane 0x02      /* Select CMap Plane number two for R/W */
# define GRW3_cplane 0x03      /* Select CMap Plane number three for R/W */

# define GRV0_cplane 0x04      /* Select CMap Plane number zero for video */
# define GRV1_cplane 0x05      /* Select CMap Plane number one for video */
# define GRV2_cplane 0x06      /* Select CMap Plane number two for video */
# define GRV3_cplane 0x07      /* Select CMap Plane number three for video */
# define GR_inten     0x10      /* Enable Interrupt to start at start
                                   of next vertical retrace. Must clear bit to
                                   clear interrupts. */

# define GR_paint     0x20      /* Enable Writing five pixels in parallel */
# define GR_disp_on   0x40      /* Enable Video Display */

# define GR_vretrace 0x80       /* Unused on write. On read, true if monitor in
                                   vertical retrace. */

/* This define returns true if the board is in vertical retrace */
# define GR_retrace  (*GR_sreg & GR_vretrace)

/* The following are function register encodings */
# define GR_copy            0xCC  /* Copy data reg to Frame buffer */
# define GR_copy_invert     0x33  /* Copy inverted data reg to FB  */
# define GR_wr_creg         0xF0  /* Copy color reg to Frame buffer */
# define GR_wr_mask         0xF0  /* Copy mask to Frame buffer */
# define GRinv_wr_creg      0x0F  /* Copy inverted Creg to FB */
# define GRinv_wr_mask      0x0F  /* Copy inverted Mask to FB */
# define GR_ram_invert      0x55  /* 'Invert' color in Frame buffer */
# define GR_cr_and_dr       0xC0  /* Bitwise and of color and data regs */
# define GR_clear           0x00  /* Clear frame buffer */
# define GR_cr_xor_fb       0x5A  /* Xor frame buffer data and Creg */
```

```
#ifndef lint
static  char sccsid[] = "@(#)cgone.c 1.12 85/02/05 Copyr 1983 Sun Micro";
#endif

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

#include "cgone.h"
#include "win.h"
#if NCGONE > 0

/*
 * Sun One Color Graphics Board(s) Driver
 */

#include "../h/param.h"
#include "../h/systm.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/proc.h"
#include "../h/buf.h"
#include "../h/conf.h"
#include "../h/file.h"
#include "../h/uio.h"
#include "../h/ioctl.h"

#include "../machine/mmu.h"
#include "../machine/pte.h"

#include "../sun/fbio.h"

#include "../sundev/mbvar.h"
#include "../pixrect/pixrect.h"
#include "../pixrect/pr_util.h"
#include "../pixrect/cg1reg.h"
#include "../pixrect/cg1var.h"

#if NWIN > 0
#define CG1_OPS &cg1_ops
struct  pixrectops cg1_ops = {
        cg1_rop,
        cg1_putcolormap,
        cg1_putattributes,
};
#else
#define CG1_OPS (struct pixrectops *)0
#endif

#define CG1SIZE (sizeof (struct cg1fb))
struct  cg1pr cgoneprdatadefault =
    { 0, 0, 255, 0, 0 };
struct  pixrect cgonepixrectdefault =
    { CG1_OPS, { CG1_WIDTH, CG1_HEIGHT }, CG1_DEPTH, /* filled in later */ 0 };
```

```
/*
 * Driver information for auto-configuration stuff.
 */
int     cgoneprobe(), cgoneintr();
struct  pixrect cgonepixrect[NCGONE];
struct  cg1pr cgoneprdata[NCGONE];
struct  mb_device *cgoneinfo[NCGONE];
struct  mb_driver cgonedriver = {
        cgoneprobe, 0, 0, 0, 0, cgoneintr,
        CG1SIZE, "cgone", cgoneinfo, 0, 0, 0,
};

/*
 * Only allow opens for writing or reading and writing
 * because reading is nonsensical.
 */
cgoneopen(dev, flag)
        dev_t dev;
{
        return(fbopen(dev, flag, NCGONE, cgoneinfo));
}

/*
 * When close driver destroy pixrect.
 */
/*ARGSUSED*/
cgoneclose(dev, flag)
        dev_t dev;
{
        register int unit = minor(dev);

        if ((caddr_t)&cgoneprdata[unit] == cgonepixrect[unit].pr_data) {
                bzero((caddr_t)&cgoneprdata[unit], sizeof (struct cg1pr));
                bzero((caddr_t)&cgonepixrect[unit], sizeof (struct pixrect));
        }
}

/*ARGSUSED*/
cgoneioctl(dev, cmd, data, flag)
        dev_t dev;
        caddr_t data;
{
        register int unit = minor(dev);

        switch (cmd) {

        case FBIOGTYPE: {
                register struct fbtype *fb = (struct fbtype *)data;

                fb->fb_type = FBTYPE_SUN1COLOR;
                fb->fb_height = 480;
                fb->fb_width = 640;
                fb->fb_depth = 8;
                fb->fb_cmsize = 256;
                fb->fb_size = 512*640;
                break;
```

```
		}
	case FBIOGPIXRECT: {
		register struct fbpixrect *fbpr = (struct fbpixrect *)data;
		register struct cg1fb *cg1fb =
		    (struct cg1fb *)cgoneinfo[(unit)]->md_addr;

		/*
		 * "Allocate" and initialize pixrect data with default.
		 */
		fbpr->fbpr_pixrect = &cgonepixrect[unit];
		cgonepixrect[unit] = cgonepixrectdefault;
		fbpr->fbpr_pixrect->pr_data = (caddr_t) &cgoneprdata[unit];
		cgoneprdata[unit] = cgoneprdatadefault;
		/*
		 * Fixup pixrect data.
		 */
		cgoneprdata[unit].cgpr_va = cg1fb;
		/*
		 * Enable video
		 */
		cg1_setreg(cg1fb, CG_FUNCREG, CG_VIDEOENABLE);
		/*
		 * Clear interrupt
		 */
		cg1_intclear(cg1fb);
		break;
		}

	default:
		return (ENOTTY);
	}
	return (0);
}

/*
 * We need to handle vertical retrace interrupts here.
 * The color map(s) can only be loaded during vertical
 * retrace; we should put in ioctls for this to synchronize
 * with the interrupts.
 * FOR NOW, see comments in the code.
 */
cgoneintclear(cg1fb)
	struct  cg1fb *cg1fb;
{
	/*
	 * The Sun 1 color frame buffer doesn't indicate that an
	 * interrupt is pending on itself so we don't know if the interrupt
	 * is for our device.  So, just turn off interrupts on the cgone board.
	 * This routine can be called from any level.
	 */
	cg1_intclear(cg1fb);
	/*
	 * We return 0 so that if the interrupt is for some other device
	 * then that device will have a chance at it.
	 */
	return(0);
```

```
}

int
cgoneintr()
{
	return(fbintr(NCGONE, cgoneinfo, cgoneintclear));
}

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

#include "../sundev/cgreg.h"
	/*
	 * Note: using old cgreg.h to peek and poke for now.
	 */
/*
 * We determine that the thing we're addressing is a color
 * board by setting it up to invert the bits we write and then writing
 * and reading back DATA1, making sure to deal with FIFOs going and coming.
 */
#define DATA1 0x5C
#define DATA2 0x33
/*ARGSUSED*/
cgoneprobe(reg, unit)
	caddr_t reg;
	int	unit;
{
	register caddr_t CGXBase;
	register u_char *xaddr, *yaddr;

	CGXBase = reg;
	if (pokec((caddr_t)GR_freg, GR_copy_invert))
		return (0);
	if (pokec((caddr_t)GR_mask, 0))
		return (0);
	xaddr = (u_char *)(CGXBase + GR_x_select + GR_update + GR_set0);
	yaddr = (u_char *)(CGXBase + GR_y_select + GR_set0);
	if (pokec((caddr_t)yaddr, 0))
		return (0);
	if (pokec((caddr_t)xaddr, DATA1))
		return (0);
	(void) peekc((caddr_t)xaddr);
	(void) pokec((caddr_t)xaddr, DATA2);
	if (peekc((caddr_t)xaddr) == (~DATA1 & 0xFF)) {
		/*
		 * The Sun 1 color frame buffer doesn't indicate that an
		 * interrupt is pending on itself.
		 * Also, the interrupt level is user program changable.
		 * Thus, the kernel never knows what level to expect an
		 * interrupt on this device and doesn't know is an interrupt
```

```
                 * is pending.
                 * So, we add the cgoneintr routine to a list of interrupt
                 * handlers that are called if no one handles an interrupt.
                 * Add_default_intr screens out multiple calls with the same
                 * interrupt procedure.
                 */
                add_default_intr(cgoneintr);
                return (CG1SIZE);
        }
        return (0);
}

#endif
```

```
/*      @(#)skyreg.h 1.3 84/12/22 SMI   */

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Sky FFP
 */

struct  skyreg {
        u_short sky_command;
        u_short sky_status;
        union {
                short   skyu_dword[2];
                long    skyu_dlong;
        } skyu;
#define sky_data        skyu.skyu_dlong
#define sky_direg       skyu.skyu_dword[0]
        long    sky_ucode;
        u_short sky_vector;     /* VME: interrupt vector number */
};

/* commands */
#define SKY_SAVE        0x1040
#define SKY_RESTOR      0x1041
#define SKY_NOP         0x1063
#define SKY_START0      0x1000
#define SKY_START1      0x1001

/* status bits */
#define SKY_IHALT       0x0000
#define SKY_INTRPT      0x0003
#define SKY_INTENB      0x0010
#define SKY_RUNENB      0x0040
#define SKY_SNGRUN      0x0060
#define SKY_RESET       0x0080
#define SKY_IODIR       0x2000
#define SKY_IDLE        0x4000
#define SKY_IORDY       0x8000
```

```
#ifndef lint
static  char sccsid[] = "@(#)sky.c 1.13 85/03/02 Copyr 1983 Sun Micro";
#endif

/*
 * Copyright (c) 1985 by Sun Microsystems, Inc.
 */

/*
 *  Sky FFP
 */
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/file.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../machine/pte.h"
#include "../machine/mmu.h"
#include "../machine/cpu.h"
#include "../machine/scb.h"
#include "../sundev/mbvar.h"
#include "../sundev/skyreg.h"

/*
 * "page" size for VME sky board
 * user page (0) doesn't allow access to nasty registers
 * supervisor page (1) does
 */
#define SKYPGSIZE       0x800

/*
 * Driver information for auto-configuration stuff.
 */
int     skyprobe(), skyattach(), skyintr();
struct  mb_device *skyinfo[1];  /* XXX only supports 1 board */
struct  mb_driver skydriver = {
        skyprobe, 0, skyattach, 0, 0, skyintr,
        2 * SKYPGSIZE, "sky", skyinfo, 0, 0, 0,
};

struct  skyreg *skyaddr;
int skyinit = 0, skyisnew = 0;

/*ARGSUSED*/
skyprobe(reg, unit)
        caddr_t reg;
        int unit;
{
        register struct skyreg *skybase = (struct skyreg *)reg;

        if (peek((short *)skybase) == -1)
                return (0);
        if (poke((short *)&skybase->sky_status, SKY_IHALT))
                return (0);
        skyaddr = (struct skyreg *)(SKYPGSIZE + reg);
        if (cpu == SUN2_120 || poke((short *)&skyaddr->sky_status, SKY_IHALT)) {
```

```
		/* old VMEbus or Multibus */
		skyisnew = 0;
		skyaddr = (struct skyreg *)reg;
	} else
		skyisnew = 1;
	return (sizeof (struct skyreg));
}

/*
 * Initialize the VME interrupt vector to be identical to
 * the 68000 auto-vector for the appropriate interrupt level
 * unless vectored interrupts have been specified.
 */
skyattach(md)
	struct mb_device *md;
{

	if (skyisnew) {
		if (!md->md_intr) {
			/* use auto-vectoring */
			(void) poke((short *)&skyaddr->sky_vector,
			    AUTOBASE + md->md_intpri);
		} else {
			/* use vectored interrupts */
			(void) poke((short *)&skyaddr->sky_vector,
			    md->md_intr->v_vec);
		}
	}
}

/*ARGSUSED*/
skyopen(dev, flag)
	dev_t dev;
	int flag;
{
	int i;
	register struct skyreg *s = skyaddr;

	if (skyaddr == 0)
		return (ENXIO);
	if (skyinit == 2) {
		/*
		 * Initialize the FFP.
		 * VME users can't do this themselves;
		 * since the status isn't writeable
		 */
		s->sky_status = SKY_RESET;
		s->sky_command = SKY_START0;
		s->sky_command = SKY_START0;
		s->sky_command = SKY_START1;
		s->sky_status = SKY_RUNENB;
		u.u_skyctx.usc_used = 1;
		u.u_skyctx.usc_cmd = SKY_NOP;
		for (i=0; i<8; i++)
			u.u_skyctx.usc_regs[i] = 0;
		skyrestore();
	} else if (flag & FNDELAY)
		skyinit = 1;
	else
		return (ENXIO);
	return (0);
}

/*ARGSUSED*/
skyclose(dev, flag)
	dev_t dev;
	int flag;
{

	/*
	 * We have to save context here in case the user aborted
	 * and left the board in an unclean state.
	 */
	if (skyinit == 2)
		skysave();
	if (skyinit == 1)
		skyinit = 2;
	u.u_skyctx.usc_used = 0;
	return (0);
}

/*ARGSUSED*/
skymmap(dev, off, prot)
	dev_t dev;
	off_t off;
	int prot;
{

	if (off)
		return (-1);
	off = (off_t)skyaddr;
	if (skyisnew && skyinit == 2)	/* use user page */
		off -= SKYPGSIZE;
	off =  getkpgmap((caddr_t)off) & PG_PFNUM;
	return (off);
}

/*ARGSUSED*/
skyintr(n)
	int n;
{
	static u_short  skybooboo = 0;

	if (skyaddr && (skyaddr->sky_status & (SKY_INTENB|SKY_INTRPT))) {
		if (skyaddr->sky_status & SKY_INTENB) {
			printf("skyintr: sky board interrupt enabled, status = 0x%x\n",
				skyaddr->sky_status);
			skyaddr->sky_status &= ~(SKY_INTENB|SKY_INTRPT);
			return (1);
		}
		if (!skybooboo && (skyaddr->sky_status & SKY_INTRPT)) {
			printf("skyintr: sky board unrecognized status, status = 0x%x\
                                        skybooboo = skyaddr->sky_status);
                                return (0);
                        }
                }
                return (0);
}

skysave()
{
        register short i;
        register struct skyreg *s = skyaddr;
        register u_short stat;

        for (i = 0; i < 100; i++) {
                stat = s->sky_status;
                if (stat & SKY_IDLE) {
                        u.u_skyctx.usc_cmd = SKY_NOP;
                        goto sky_save;
                }
                if (stat & SKY_IORDY)
                        goto sky_ioready;
        }
        printf("sky0: hung\n");
        skyinit = 0;
        u.u_skyctx.usc_used = 0;
        return;

        /*
         * I/O is ready, is it a read or write?
         */
sky_ioready:
        s->sky_status = SKY_SNGRUN;     /* set single step mode */
        if (stat & SKY_IODIR)
                i = s->sky_direg;
        else
                s->sky_direg = i;

        /*
         * Check again since data may have been a long word.
         */
        stat = s->sky_status;
        if (stat & SKY_IORDY)
                if (stat & SKY_IODIR)
                        i = s->sky_direg;
                else
                        s->sky_direg = i;

        /*
         * Read and save the command register.
         * Decrement by 1 since command register
         * is actually FFP program counter and we
         * want to back it up.
         */
        u.u_skyctx.usc_cmd = s->sky_command - 1;

        /*
```

```
         * Reinitialize the FFP.
         */
        s->sky_status = SKY_RESET;
        s->sky_command = SKY_START0;
        s->sky_command = SKY_START0;
        s->sky_command = SKY_START1;
        s->sky_status = SKY_RUNENB;

        /*
         * Finally, actually do the context save function.
         * (Unrolled loop for efficiency.)
         */
sky_save:
        s->sky_command = SKY_NOP;       /* set FFP in a clean mode */
        s->sky_command = SKY_SAVE;
        u.u_skyctx.usc_regs[0] = s->sky_data;
        u.u_skyctx.usc_regs[1] = s->sky_data;
        u.u_skyctx.usc_regs[2] = s->sky_data;
        u.u_skyctx.usc_regs[3] = s->sky_data;
        u.u_skyctx.usc_regs[4] = s->sky_data;
        u.u_skyctx.usc_regs[5] = s->sky_data;
        u.u_skyctx.usc_regs[6] = s->sky_data;
        u.u_skyctx.usc_regs[7] = s->sky_data;
}

skyrestore()
{
        register struct skyreg *s = skyaddr;

        if (skyinit != 2) {
                u.u_skyctx.usc_used = 0;
                return;
        }
        s->sky_command = SKY_NOP;       /* set FFP in a clean mode */

        /*
         * Do the context restore function.
         */
        s->sky_command = SKY_RESTOR;
        s->sky_data = u.u_skyctx.usc_regs[0];
        s->sky_data = u.u_skyctx.usc_regs[1];
        s->sky_data = u.u_skyctx.usc_regs[2];
        s->sky_data = u.u_skyctx.usc_regs[3];
        s->sky_data = u.u_skyctx.usc_regs[4];
        s->sky_data = u.u_skyctx.usc_regs[5];
        s->sky_data = u.u_skyctx.usc_regs[6];
        s->sky_data = u.u_skyctx.usc_regs[7];
        s->sky_command = u.u_skyctx.usc_cmd;
}
```

```
/*      @(#)vpreg.h 1.4 84/12/22 SMI    */

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Registers for Ikon 10071-5 Multibus/Versatec interface
 * Only low byte of each word is used. (16 words total)
 * Warning - read bits are not identical to written bits.
 */
struct vpdevice {
        u_short vp_status;      /* 00: mode(w) and status(r) */
        u_short vp_cmd;         /* 02: special command bits (w) */
        u_short vp_pioout;      /* 04: PIO output data (w) */
        u_short vp_hiaddr;      /* 06: hi word of Multibus DMA address (w) */
        u_short vp_icad0;       /* 08: ad0 of 8259 interrupt controller */
        u_short vp_icad1;       /* 0A: ad1 of 8259 interrupt controller */
        /* The rest of the fields are for the 8237 DMA controller */
        u_short vp_addr;        /* 0C: DMA word address */
        u_short vp_wc;          /* 0E: DMA word count */
        u_short vp_dmacsr;      /* 10: command and status */
        u_short vp_dmareq;      /* 12: request */
        u_short vp_smb;         /* 14: single mask bit */
        u_short vp_mode;        /* 16: dma mode */
        u_short vp_clrff;       /* 18: clear first/last flip-flop */
        u_short vp_clear;       /* 1A: DMA master clear */
        u_short vp_clrmask;     /* 1C: clear mask register */
        u_short vp_allmask;     /* 1E: all mask bits */
};
/* vp_status bits (read) */
#define VP_IS8237       0x80    /* 1 if 8237 (sanity checker) */
#define VP_REDY         0x40    /* printer ready */
#define VP_DRDY         0x20    /* printer and interface ready */
#define VP_IRDY         0x10    /* interface ready */
#define VP_PRINT        0x08    /* print mode */
#define VP_NOSPP        0x04    /* not in SPP mode */
#define VP_ONLINE       0x02    /* printer online */
#define VP_NOPAPER      0x01    /* printer out of paper */
/* vp_status bits (written) */
#define VP_PLOT         0x02    /* enter plot mode */
#define VP_SPP          0x01    /* enter SPP mode */

/* vp_cmd bits */
#define VP_RESET        0x10    /* reset the plotter and interface */
#define VP_CLEAR        0x08    /* clear the plotter */
#define VP_FF           0x04    /* form feed to plotter */
#define VP_EOT          0x02    /* EOT to plotter */
#define VP_TERM         0x01    /* line terminate to plotter */

#define VP_DMAMODE      0x47    /* magic for vp_mode */

#define VP_ICPOLL       0x0C
#define VP_ICEOI        0x20
```

```
#ifndef lint
static  char sccsid[] = "@(#)vp.c 1.13 85/02/05 Copyr 1983 Sun Micro";
#endif

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

#include "vp.h"
#if NVP > 0
/*
 * Versatec matrix printer/plotter
 * dma interface driver for Ikon 10071-5 Multibus/Versatec interface
 */
#include "../h/param.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/buf.h"
#include "../h/systm.h"
#include "../h/kernel.h"
#include "../h/map.h"
#include "../h/ioctl.h"
#include "../h/vcmd.h"
#include "../h/uio.h"

#include "../machine/psl.h"
#include "../machine/mmu.h"
#include "../sundev/vpreg.h"
#include "../sundev/mbvar.h"

#define VPPRI   (PZERO-1)

struct vp_softc {
        int     sc_state;
        struct buf *sc_bp;
        int     sc_mbinfo;
} vp_softc[NVP];

#define VPSC_BUSY       0400000
/* sc_state bits - passed in VGETSTATE and VSETSTATE ioctls */
#define VPSC_MODE       0000700
#define VPSC_SPP        0000400
#define VPSC_PLOT       0000200
#define VPSC_PRINT      0000100
#define VPSC_CMNDS      0000076
#define VPSC_OPEN       0000001

#define VPUNIT(dev)     (minor(dev))

struct  buf rvpbuf[NVP];

int vpprobe(), vpintr();

struct  mb_device *vpdinfo[NVP];
struct mb_driver vpdriver = {
        vpprobe, 0, 0, 0, 0, vpintr,
	sizeof (struct vpdevice), "vp", vpdinfo, 0, 0, 0,
};

vpprobe(reg)
	caddr_t reg;
{
	register struct vpdevice *vpaddr = (struct vpdevice *)reg;
	register int x;

	x = peek((short *)&vpaddr->vp_status);
	if (x == -1 || (x & VP_IS8237) == 0)
		return (0);
	if (poke((short *)&vpaddr->vp_cmd, VP_RESET))
		return (0);
	/* initialize 8259 so we don't get constant interrupts */
	vpaddr->vp_icad0 = 0x12;	/* ICW1, edge-trigger */
	DELAY(1);
	vpaddr->vp_icad1 = 0xFF;	/* ICW2 - don't care (non-zero) */
	DELAY(1);
	vpaddr->vp_icad1 = 0xFE;	/* IR0 - interrupt on DRDY edge */
	/* reset 8237 */
	vpaddr->vp_clear = 1;

	return (sizeof (struct vpdevice));
}

vpopen(dev)
	dev_t dev;
{
	register struct vp_softc *sc;
	register struct mb_device *md;
	register int s;
	static int vpwatch = 0;

	if (VPUNIT(dev) >= NVP ||
	    ((sc = &vp_softc[minor(dev)])->sc_state&VPSC_OPEN) ||
	    (md = vpdinfo[VPUNIT(dev)]) == 0 || md->md_alive == 0)
		return (ENXIO);
	if (!vpwatch) {
		vpwatch = 1;
		vptimo();
	}
	sc->sc_state = VPSC_OPEN|VPSC_PRINT | VPC_CLRCOM|VPC_RESET;
	while (sc->sc_state & VPSC_CMNDS) {
		s = splx(pritospl(md->md_intpri));
		if (vpwait(dev)) {
			vpclose(dev);
			return (EIO);
		}
		vpcmd(dev);
		(void) splx(s);
	}
	return (0);
}

vpclose(dev)
```

```
        dev_t dev;
{
        register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];

        sc->sc_state = 0;
}

vpstrategy(bp)
        register struct buf *bp;
{
        register struct vp_softc *sc = &vp_softc[VPUNIT(bp->b_dev)];
        register struct mb_device *md = vpdinfo[VPUNIT(bp->b_dev)];
        register struct vpdevice *vpaddr = (struct vpdevice *)md->md_addr;
        int s;
        int pa, wc;

        if (((int)bp->b_un.b_addr & 1) || bp->b_bcount < 2) {
                bp->b_flags |= B_ERROR;
                iodone(bp);
                return;
        }
        s = splx(pritospl(md->md_intpri));
        while (sc->sc_bp != NULL)               /* single thread */
                sleep((caddr_t)sc, VPPRI);
        sc->sc_bp = bp;
        (void) vpwait(bp->b_dev);
        sc->sc_mbinfo = mbsetup(md->md_hd, bp, 0);
        vpaddr->vp_clear = 1;
        pa = MBI_ADDR(sc->sc_mbinfo);
        vpaddr->vp_hiaddr = (pa >> 16) & 0xF;
        pa = (pa >> 1) & 0x7FFF;
        wc = (bp->b_bcount >> 1) - 1;
        bp->b_resid = 0;
        vpaddr->vp_addr = pa & 0xFF;
        vpaddr->vp_addr = pa >> 8;
        vpaddr->vp_wc = wc & 0xFF;
        vpaddr->vp_wc = wc >> 8;
        vpaddr->vp_mode = VP_DMAMODE;
        vpaddr->vp_clrmask = 1;
        sc->sc_state |= VPSC_BUSY;
        (void) splx(s);
}

/*ARGSUSED*/
vpwrite(dev, uio)
        dev_t dev;
        struct uio *uio;
{

        if (VPUNIT(dev) >= NVP)
                return (ENXIO);
        return (physio(vpstrategy, &rvpbuf[VPUNIT(dev)], dev, B_WRITE,
                    minphys, uio));
}

vpwait(dev)
```

```
	dev_t dev;
{
	register struct vpdevice *vpaddr =
	    (struct vpdevice *)vpdinfo[VPUNIT(dev)]->md_addr;
	register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];

	for (;;) {
		if ((sc->sc_state & VPSC_BUSY) == 0 &&
		    vpaddr->vp_status & VP_DRDY)
			break;
		sleep((caddr_t)sc, VPPRI);
	}
	return (0);	/* NO ERRORS YET */
}

struct pair {
	char	soft;		/* software bit */
	char	hard;		/* hardware bit */
} vpbits[] = {
	VPC_RESET,	VP_RESET,
	VPC_CLRCOM,	VP_CLEAR,
	VPC_EOTCOM,	VP_EOT,
	VPC_FFCOM,	VP_FF,
	VPC_TERMCOM,	VP_TERM,
	0,		0,
};

vpcmd(dev)
	dev_t;
{
	register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];
	register struct vpdevice *vpaddr =
	    (struct vpdevice *)vpdinfo[VPUNIT(dev)]->md_addr;
	register struct pair *bit;

	for (bit = vpbits; bit->soft != 0; bit++) {
		if (sc->sc_state & bit->soft) {
			vpaddr->vp_cmd = bit->hard;
			sc->sc_state &= ~bit->soft;
			DELAY(100);	/* time for DRDY to drop */
			return;
		}
	}
}

/*ARGSUSED*/
vpioctl(dev, cmd, data, flag)
	dev_t dev;
	int cmd;
	caddr_t data;
	int flag;
{
	register int m;
	register struct mb_device *md = vpdinfo[VPUNIT(dev)];
	register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];
	register struct vpdevice *vpaddr = (struct vpdevice *)md->md_addr;
```

```
        int s;

        switch (cmd) {

        case VGETSTATE:
                *(int *)data = sc->sc_state;
                break;

        case VSETSTATE:
                m = *(int *)data;
                sc->sc_state =
                    (sc->sc_state & ~VPSC_MODE) | (m&(VPSC_MODE|VPSC_CMNDS));
                break;

        default:
                return (ENOTTY);
        }
        s = splx(pritospl(md->md_intpri));
        (void) vpwait(dev);
        if (sc->sc_state&VPSC_SPP)
                vpaddr->vp_status = VP_SPP|VP_PLOT;
        else if (sc->sc_state&VPSC_PLOT)
                vpaddr->vp_status = VP_PLOT;
        else
                vpaddr->vp_status = 0;
        while (sc->sc_state & VPSC_CMNDS) {
                (void) vpwait(dev);
                vpcmd(dev);
        }
        (void) splx(s);
        return (0);
}

vpintr()
{
        register int dev;
        register struct mb_device *md;
        register struct vpdevice *vpaddr;
        register struct vp_softc *sc;
        register int found = 0;

        for (dev = 0; dev < NVP; dev++) {
                if ((md = vpdinfo[dev]) == NULL)
                        continue;
                vpaddr = (struct vpdevice *)md->md_addr;
                vpaddr->vp_icad0 = VP_ICPOLL;
                DELAY(1);
                if (vpaddr->vp_icad0 & 0x80) {
                        found = 1;
                        DELAY(1);
                        vpaddr->vp_icad0 = VP_ICEOI;
                }
                sc = &vp_softc[dev];
                if ((sc->sc_state&VPSC_BUSY) && (vpaddr->vp_status & VP_DRDY)) {
                        sc->sc_state &= ~VPSC_BUSY;
                        if (sc->sc_state & VPSC_SPP) {
```

```
                                        sc->sc_state &= ~VPSC_SPP;
                                        sc->sc_state |=  VPSC_PLOT;
                                        vpaddr->vp_status = VP_PLOT;
                                }
                                iodone(sc->sc_bp);
                                sc->sc_bp = NULL;
                                mbrelse(md->md_hd, &sc->sc_mbinfo);
                        }
                        wakeup((caddr_t)sc);
                }
                return (found);
}

vptimo()
{
                int s;
                register struct mb_device *md = vpdinfo[0];

                s = splx(pritospl(md->md_intpri));
                (void) vpintr();
                (void) splx(s);
                timeout(vptimo, (caddr_t)0, hz);
}
#endif NVP > 0
```

Index

B

bdevsw, 2-2
block devices, 2-1
bottom half of driver, 3-1
building a kernel, 4-1
byte order, 2-5

C

cdevsw, 2-2
character devices, 2-1
configuring the system, 4-1

D

/dev directory, 2-1
device class, 2-1
device names, 2-1
device numbers, 2-1
devices numbers
    macros to manipulate, 3-8
DMA devices, 2-8

I

I/O mapped devices, 2-5
interrupt context, 3-1
interrupt levels, 3-3
interrupt priorities
    raising and lowering, 3-7
interrupts, 3-2
    polling, 3-3
    vectored, 3-3

K

kernel address space, 3-1
kernel `printf` function, 3-8

M

major device number, 2-1
major macro, 3-8
makedev macro, 3-8
MAKEDEV shell script, 4-2
memory mapped devices, 2-5
minor device number, 2-1
minor macro, 3-8
Multibus, 2-5
Multibus I/O address space, 2-5
Multibus memory address space, 2-5
multibus resource management, 3-8

P

polling interrupts, 3-3
`printf` function in kernel, 3-8

R

raising and lowering interrupt priorities, 3-7

S

service functions, 3-6
    multibus resource management, 3-8
    raise and lower interrupt priorities, 3-7
    sleep and wakeup, 3-6
    timeout, 3-6
sleep and wakeup mechanism, 3-6
Sun Main Bus DVMA, 2-8
system configuration, 4-1

T

timeout mechanisms, 3-6
top half of driver, 3-1

U

user address space, 3-1
user context, 3-1

V

vectored interrupts, 3-3
VMEbus, 2-4
VMEbus address space, 2-12